# Generics: Type-Parametric Programming

Compiler Construction '15 Final Report

Johannes Heinemann Patrik Ackland

KTH - Royal Institute of Technology

{heinem,packland}@kth.se

## 1. Introduction

Typically, a compiler runs through several different kind of phases in order to transform the source code to bytecode [Appel 2002, pp 12-15]. In the previous part of our project did we implement those steps with the following structure:

- Lexer: Split up the source file into different tokens.
- Parser: Analyse the phrase structure and build an Abstract Syntax Tree (ABS).
- Name Analysis: Build a symbol tree as basis for type checking. In addition, object binding and definite assignment is performed.
- Type Checking: Checks the program for type errors.
- Code Generation: Convert the ABS into Java bytecode.

With this we built a fully functional compiler, that could compile a simple object oriented programming language called KOOL (see [Phaller 2015] for more details). In this part of the project, we introduce a simple form of generics as an extension for this language. Our assignment was then to add type variables to the KOOL compiler, which initiate only a generic class.

The purpose of generics is to provide classes that can be used with any type and enables type checking at compile-time. This is done by letting the class use an identifier that is only available in that class. When an object declaration of this class occurs, the user specifies which type it would like the object to use instead of this identifier. This enables the type checker to make sure that all methods which use this identifier, then use the type specified by the user. Without generics this would have to be done by creating a list which takes any type and then type-cast the value. This could fail at run time.

## 2. Examples

Generics is most commonly used for data structures such as Lists, Trees or Hashmaps, but can also be used for algorithms that should be used with multiple types. Probably, the most famous example for generics is the Linkedlist. Let's assume we want to have a list that allows all the possible classes as types so we do not have to implement the list several times. In order to do that we need to use the most general type for the element. Then this needs to be casted to the desired type. In KOOL that could look like the following example with a list of string elements:

```
var myList: LinkedList =

  new LinkedList();
myList.addFirst("Hello world!");
var str: String = (String) myList.getFirst();
```

For retrieving the desired string, we have to introduce a cast. However, this brings to problems with it. On the one side the cast is annoying, because it feels unnecessary and is ugly. On the other side run the third line the risk of resulting in a run time error. Although, the programmer should know which type is contained in the list, there mistakes are always possible. Therefore, it would be better to let already the compiler check if the types coincide, so the error is discovered during compile time. This is exactly where generic types come into play. Taking the same example as before, creating and using an object from a class with generics is depicted in listings 1.

**Listing 1.** LinkedList

```
var myList: LinkedList[String] = new LinkedList[String]();
myList.addElement("Hello world!");
var str: String = myList.getFirst();
```

The corresponding generic LinkedList class is then declared as follows:

```
class LinkedList[T1] {
  var next: LinkedList[T1];
  var elem: T1;

  def addFirst(elem: T1): Bool = { ... }
  def getFirst(): T1 = { ... }
}
```

## 3. Implementation

### 3.1 Theoretical Background

The approach of generic programming can was defined by two different directions. One is the gradual lifting of concrete algorithms [Reis 2005] and was brought up by [Musser and Stepanov 1989]. The idea is to take a pragmatic algorithm and transform it gradual into a more abstract version without losing efficiency or semantics. This methodology is coming from the imperative languages, where C++/Java are playing a dominant role nowadays.

The second school a calculational approach, often also referred to as Datatype-Generic programming [Gibbons 2007]. [Bird and Meertens 1998] were the ones who created the basis with their work. The focus is on regular data types by exploiting algebras generated by functors sum, product and unit. Thus, it is possible to operate any inductive datatype, if the algorithms are written for those functors. In contrary to the first approach has Datatype-Generic programming Haskell and its variants as the most exclusive tool.

Due to the object oriented language type from KOOL and the close relationship from this language to Java, we decided to use the first approach for our generics extension.

Our generics extension, like Java, uses type erasure. This means that the generic types are checked at compile time and then replaced with Object. This limits generics to classes. Primitive types like boolean and int are not allowed. This made the implementation easier but makes the language less powerful.

### 3.2 Implementation Details

**Lexer**

Our changes are stated in the table below. Most of the changes were made in the grammar. The lexer already supported tokens for identifiers and braces.

| | | |
|---|---|---|
| Type | ::= | Identifier [ GenericType ] ; |
| ClassDeclaration | ::= | class Identifier ([Identifier])? .... |
| GenericType | ::= | String \| Identifier |

**Parser**

The parser needs to recognize the new grammar and add the optional identifier to the ClassDecl object. The parser must also accept the old grammar, because the new grammar is a superset of the old one with the generic type being optional. Note that this grammar is, in contrast to java, limited to only one generic type per class. The parser checks that the generic type is either identifier or string since we do not allow primitive types. The changes we made looks like this:

```
def parseGenericIdentifier: Identifier = {
  val pos = currentPos
  val identifier: Identifier = parseIdentifier
  var genericType: Option[TypeTree] = None
  if (currentToken.kind == LBRACKET) {
    eat(LBRACKET)
    val pos2 = currentPos
    genericType = currentToken.kind match {
      case STRING =>
        eat(STRING)
        Some((new StringType).setPos(pos2))
      case IDKIND =>
        Some(parseGenericIdentifier)
      case _ =>
        error("Primitive types not allowed for generics", pos2)
        readToken
        None
    }
    eat(RBRACKET)
  }
  identifier.genericType = genericType
  identifier.setPos(pos)
}
```

Which can be explained as follows: If an identifier is followed by a left bracket. Parse the generic identifier that follows. As shown in the code, only STRING and Identifier are valid types to be used as generic types.

**Name Analyser**

The name analyzer has to make sure the generic class identifier is only used inside that class. When implementing the compiler we made sure every reference to the same class had the same symbol. This

meant we could not attach the generic type to the symbol. What we did instead was create a new symbol GenericSymbol which is attached to every reference to the generic identifier inside the class. We then added a field to the TObject type to allow it to take a type in addition to a class symbol. This means we could check the type of the object when doing method calls or return statements. The changes are illustrated below.

```
def handleGenerics(cd: ClassDecl, cs: ClassSymbol) =
cd.id.genericType match {
  case Some(t) =>
    t match {
      case Identifier(v,_) =>
        var gs = new GenericSymbol(v)
        gs.setType(TGeneric(gs))
        gs.setPos(cd.id)
        cs.genericSymbol = Some(gs)
      case _ =>
    }
  case None =>
}
```

In the code above, if we have a generic identifier for this class, create a generic symbol.

The generic symbol is attached like this:

```
def setTypeSymbol(tpe: TypeTree, cs: ClassSymbol) {
  tpe match {
    case id: Identifier => {
      // ...
        case None =>
          cs.genericSymbol match {
            case Some(gs) =>
              if (id.value == gs.name)
                id.setSymbol(gs)
            case None =>
              error("Class not declared", tpe)
          }
      }
    } case _ =>
  }
}
```

In the code above, if we haven't found a class with a matching name, we check if the identifier is the generic identifier. If it is, we assign the generic symbol.

This is how it is used in practice

```
class A[T] {
  var t: T;
  def meth(q: T): String = {
    // ...
  }
}
```

In the code above, T would get assigned a a GenericSymbol which has the type TGeneric. When using generics in an object the code looks like this:

```
println(new A[String]().meth("text"));
```

In this code, the identifier A is parsed as a generic identifier by the parser. This means it has the type TObject(classSymbol, StringType()). In the type checker, we'll use the two fields of TObject to type check the generic parameters.

### Type checker

In the type checker we use our extended TObject type to make sure that the second parameter in TObject (string in the example above), matches the argument passed to a method which takes a generic argument. Our code for doing that looks like this:

```
if (a(i).toString == "generic") {
  mc.obj.getType match {
    case TObject(_,Some(t)) =>
    if(mc.args(i).getType != t.getType) {
      var err: String = "Type error: expected: " +
      t.getType + " found: " + mc.args(i).getType
      error(err, mc.args(i))
    }
    case _ =>
  }
} else {
  mc.args(i).setType(tcExpr(mc.args(i), a(i)))
}
```

Which can be read like this: If the parameter is generic, see if the generic type of the object is the same as our argument. As shown in the listing above, the normal type checking procedure (tcExpr) is not used for generic arguments. In addition to checking method arguments we also have to check the methods return type whenever it is generic. This is done as follows (in tcExpr)

```
case mc: MethodCall =>
  mc.setType(tcMethCall(mc))
  mc.obj.getType match {
    case TObject(cs,Some(t)) =>
      generic = true
      if (!expected.isEmpty &&
```

```
        !expected.contains(typeTreeToType(t,cs)))
      error("Type error", mc)
    case _ =>
  }
  mc.getType
```

Which means that if the method return type is generic (indicated by the fact that it has a second argument in TObject), see if it is one of the expected types. $generic = true$ is used to bypass the regular typechecking which is done in tcExpr. This is because the type of the method is still TGeneric. An example of how this is used is given below. Keeping TGeneric here means we have to implement a few special cases. However, it makes the code generation much easier as we'll se later.

A more complete example:

```
class A[T] {
  def meth(t: T): T = {
    return t;
  }
}

//.... usage

println(new A[String]().meth("ok"));

//.. stdout: ok
```

In this code, t and T in A will have the type TGeneric, new A[String]() will have type TObject(ClassSymbol, StringType()) and "ok" will of course have type TString. This is assigned by the name analyzer. In the type checker we'll see that meth takes a generic parameter. When the type checker notices this it checks the method calling object for a generic type (stored in TObject). This is compared to the type of "ok". Since both are string we are OK. When type checking the argument to println we provide a number of types that are allowed (TString, TBoolean, TInt). The type checker will compare the expected types to the type stored in TObject. Since the generic type is String and this is allowed for println, the check goes through.

Another special case is the plus operator. It is only allowed with int or string. This is solved by telling tcExpr to expect Int or String. It doesn't matter if the objects involved are generic or not as our code uses the expected values for method calls in both cases. However, since Int is not allowed. If generic types are involved the result must be string. This is what our code looks like:

```
def tcPlus(p: Plus): Type = {
  p.lhs.setType(tcExpr(p.lhs, TString, TInt))
  p.rhs.setType(tcExpr(p.rhs, TString, TInt))
  if (p.lhs.getType == TInt &&
      p.rhs.getType == TInt) TInt
  else if (p.lhs.getType == TInt &&
           p.rhs.getType == TString) TString
  else if (p.lhs.getType == TString &&
           p.rhs.getType == TInt) TString
  else if (p.lhs.getType == TString &&
           p.rhs.getType == TString) TString
  else if (p.lhs.getType.toString == "generic" ||
           p.rhs.getType.toString == "generic") {
    TString
  } else {
    error("Cannot use + on " + p.lhs.getType +
          " and " + p.rhs.getType, p)
    TError
  }
}
```

### Code Generation

Since we only allow strings and objects the code generation has a really easy job. It needs to convert all the TGenerics to Object. This is done with the following code:

```
def typeToByteCode(tpe: Type): String = tpe match {
  case TString => "Ljava/lang/String;"
  case TInt => "I"
  case TBoolean => "Z"
  case TIntArray => "[I"
  case TObject(v,_) => "L" + v.name + ";"
  case TGeneric(_) => "Ljava/lang/Object;";
  case TError | TUntyped => ""
}
```

## 4. Possible Extensions

There are three ways that came to our mind on how to extend the simple generic programming approach we defined here. Although, also other extension could have been possible, we chose those, because our language is strongly influenced on the programming language Java, it makes sense to chose their concepts for the next implementation step.

Instead of just allowing type variables for classes, one could also allow these for methods. This is especially the case, when one does not want to have

a generic type for the whole class, but only for one method. Accordingly, a generic method could be look like the following:

```
def print(something: T): Bool [T] { println(something); .... }
```

It is often the case that we would like to specify the generic type that is given in a method or a class. Lets assume we want to have a generic function max that gives us the largest element in a list of elements. How can we implement that without additional knowledge about the element type? Therefore, we constrain the type to extend the Comparable class in order to allow a comparison between two elements (illustrated in Listing 2).

**Listing 2.** Extended LinkedList

```
def max( list: LinkedList[T]): T [T extends Comparable] {
  var largest: T = list.getFirst();
  for (int i = 1; i < list.length; i++){
    if (list.at(i).compareTo(smallest) > 0) smallest = list.at(i);
  }
  return smallest;
}
```

Another idea for an extension is the wildcard ?, which can be found in Java. It represents an unknown type and can be bound to be all classes which extends a certain class. Examples are shown below:

1.
```
var a: Array[?] = new Array[String];
```

2.
```
def meth(list: List[? extends Shape]) = {
  list.add(...
}
```

## 5. How to compile the code

We have only added parts to the existing code and the sbt structure is the same. This means the code is run just like labs 1-6. When standing in the provided folder *koolc*. This is how you would run our provided example program which demonstrates how to use generics with our compiler.

```
koolc> sbt
sbt> compile
sbt> run programs/example/lab7.koolc
sbt> exit
koolc> java Example
```

## References


Appel, A. W. [2002], *Modern Compiler Implementation in Java*, 2nd edn, Cambridge University Press.

Bird, R. S. and Meertens, L. G. L. T. [1998], Nested datatypes, *in* 'Proceedings of the Mathematics of Program Construction', MPC '98, Springer-Verlag, London, UK, UK, pp. 52–67.

Gibbons, J. [2007], Datatype-generic programming, *in* 'Proceedings of the 2006 International Conference on Datatype-generic Programming', SSDGP'06, Springer-Verlag, Berlin, Heidelberg, pp. 1–71.

Musser, D. R. and Stepanov, A. A. [1989], Generic programming, *in* 'Proceedings of the International Symposium ISSAC'88 on Symbolic and Algebraic Computation', ISAAC '88, Springer-Verlag, London, UK, UK, pp. 13–25.

Phaller, P. [2015], 'Dd2488 - compiler construction', online. Available at: `http://www.csc.kth.se/~phaller/compilers/` [Accessed 27th May 2015].

Reis, G. D. [2005], What is generic programming, *in* 'In LibraryCentric Software Design, OOPSLA workshop'.