Generics Programming in SLAC

Compiler Construction 2016 Final Report

Michael Meyer Trace Powers
KTH Royal Institute of Technology
{meye,trace}@kth.se

1. Introduction

In the required parts of the project, our task was to implement the entire pipeline of the compiler, which includes lexing, parsing, name analysis, type checking, and code generation.

Each of these specific phases transforms the current representation of the program into a more useful representation of the desired program, along with catching any errors that come along the way.

First, lexing converts plain text into a sequence of tokens. This is useful because we have an understanding of the meaning of a token as a part of the language. a '=' in plain text isn't meaningful because we don't know if it is part of an equals sign or an assignment. However, an EQSIGN or an EQUALS is much more meaningful.

Parsing converts the sequence of tokens into a tree. This tree is the organization of the program. It describes not just what is in the program, such as classes and identifiers, but rather how the parts come together to form a more complex meaning.

Name analysis attaches the concept of a specific variable to many identifiers so the compiler then understands what the identifier 'x' means in a specific context.

Type checking and code generation don't necessarily transform the representation of the program, but are needed to produce the end result of the binary.

Code reuse is an important feature of a language. Classes allow us to reuse many functions through inheritance. However, many clases might have to be written multiple times to support different types. Interfaces solve this problem, but in doing so produce unneeded code. Generics allow us to sidestep this issue by generating new versions of a general class for the specific type we need on the fly automatically in the compiler. This project aims to implement generics support for the SLAC programming language.

2. Examples

The biggest use of generics is in data structures and algorithms, where the data structure acts the same for all types.

Generics allow for the creation of a standard library, which can have all the useful data structures already implemented for the user.

The first data structure we looked into was the linked list.

```
class List[T] {

  var val : T;
  var next : List[T];
  var hasNext : Bool;

  method init(v : T): List[T] = {
    val = v;
    hasNext = false;
    self
  }

  method append(v : T): Unit = {
    if(hasNext) {
      next.append(v)
    } else {
      next = new List[T]().init(v);
      hasNext = true
    }
  }

  method attach(v : T): List[T] = {
    var tmp : List[T];
    tmp = new List[T]().init(v);
    tmp.set_next(self);
    tmp
  }
  method set_next(v : List[T]) : Unit = {
    next = v;
    hasNext = true
  }


  method set(n : Int, v : T) : Unit = {
    if(n == 0) {
      val = v
    } else {
      next.set(n−1,v)
    }
  }

  method get(n : Int) : T = {
    if(n == 0) {
      val
    } else {
      next.get(n−1)
    }
  }

  method next_pointer() : List[T] = {
    next
  }

  method len(): Int = {
    if(hasNext) {
      1 + next.len()
    } else {
      1
    }
  }

}
```

The code above creates a list class for any type. It supports appending to the list, getting and setting objects and getting the length. The user can then use this in their own methods shown below.

```
var m : List[Int];
m = new List[Int].init(5);
m.append(6);
m.append(104);
m.get(2)
```

With a list class defined for all objects, more complex data structures such as Stacks and Queues can be formed. They are defined below.

```
class Stack[T] {

  var lst : List[T];

  method init(v : T): Stack[T] = {
    lst = new List[T]().init(v);
    self
  }

  method push(v : T): Unit = {
    lst = lst.attach(v)
  }

  method pop(): T = {
    var tmp : T;
    tmp = lst.get(0);
    lst = lst.next_pointer();
    tmp
  }

}


class Queue[T] {

  var lst : List[T];

  method init(v : T): Queue[T] = {
    lst = new List[T]().init(v);
    self
  }
  method push(v : T): Unit = {
    lst.append(v)
  }
  method pop(): T = {
    var tmp: T;
    tmp = lst.get(0);
    lst = lst.next_pointer();
    tmp
  }

}
```

The stacks and queues can be run with the code below (results also shown).

```
stk = new Stack[Int]().init(10);
stk.push(20);
stk.push(30);
que = new Queue[Int]().init(10);
que.push(20);
que.push(30);

c = 3;
while(0 < c) {
  println("ITER");
  println("stack" + strOf(stk.pop()));
  println("queue" + strOf(que.pop()));
  c = c − 1
};

==== RESULTS ====
ITER
stack30
queue10
ITER
stack20
queue20
ITER
stack10
queue30
```

Furthermore, functional programming types can be constructed, such as the Option type.

```
class Option[T] {
  var value : T;
  var has : Bool;

  method some(v : T): Option[T] = {
    value = v;
    has = true;
    self
  }
  method none(): Option[T] = {
    has = false;
    self
  }
  method get(): T = {
    value
  }
  method contains(): Bool = {
    has
  }
}
```

3. Implementation

We implemented generics support by adding another stage to the pipeline. This stage is called the generics stage and fits in between the parsing stage and the name analysis, both of which had to be slightly modified, mostly to recognize identifiers that can have brackets next to them (aka List[T] or List[Int]).

The generics phase scans the entire program and searches for generic uses, based off variable delcarations and new object creation. It obtains a list for each generic type of which concrete versions to generate on the fly.

For example, if a List[Int] was called in the main method, the compiler would know to 'implement' the generic List class for type Int.

To 'implement' a class, it clones the classDecl tree and replaces any identifiers of type T (or whatever the user called the generic type), with the identifier or type of the concrete version, aka the Int.

Then all references of List[Int] are changed to the concrete class, which is named with a '#' to the end of the class. So the concrete List[Int] is converted into a List#Int. Then the type checker can just look up the List#Int class as normal.

The first big issue that came up when testing was that multiple casts of the generic caused the compiler to fail. Everything worked fine in a single cast case, but the second cast caused the type checking on the first one to fail. After debugging this for a few hours, we found that the issue was hidden not in the type checking or name analysis, but in the generic phase of the compiler. This was because the tree cloning process was sharing some Identifier tree nodes. This caused the setSymbol command to overwrite each other, thus making a String cast, which required a symbol with type String, to instead get the symbol for type Int.

Another issue that came up was the generic type dependencies. For example, if a Stack[T] requires a List[T] for the implementation, but a concrete List[T] is never created somewhere else in the program, the compiler originally failed because the Stack[T] called a class that didn't exist. To solve this, we had to use a queue to specify a work list of tuples of type (ClassDecl, Type). The compiler would generate this generic class of type Type, and then also add to the queue any types it depends on. This solved the issue because then since Stack[Int] requires a List[Int], the list would be created even though the user never wrote List[Int] in their code.

4. Possible Extensions

This system only supports one type generic classes. This makes it more difficult to implement data structures such as maps, although there probably is a special way to implement them using structures such as tuples.

However, this is still an inconvience for the programmer and an arbitrary amount of generic types should be supported in the future. Since generics are just one solution for the general issue of code reusability, certain features can be tacked on to a basic generics program.

In certain cases, most code of a class can be represented with generics, except for a single function, such as a specific library call. It would still be useful to have generics for this situation, but it can't be done using the type replacement mechansism. By implementing first class functions and then passing a function in to the type generator as well, this issue could be alieviated. Of course, a wrapper class could be written instead but this is uneeded code and what the generic is trying to replace.

References

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