List Comprehensions

Computer Language Processing '18 Final Report

Samuel Chassot Julie Giunta
EPFL
samuel.chassot@epfl.ch julie.giunta@epfl.ch

1. Introduction

In this project, we wrote a compiler for the Amy language. The Amy language is Turing complete and our compiler handles it fine. The compiler lexes the program to obtain a list of tokens that are next parsed to produce an AST (Abstract Syntax Tree). We then do some name analysis and type check to ensure that variables, types and functions are well-defined and that everything is called accordingly. We then produce the web assembly code corresponding to the given program.

Even if every algorithm can be expressed in Amy, working with constructors, especially List's ones, is a bit boring. We always need to call *L.Cons* or *L.Nil* to construct lists and pattern match every time we want to work on them.

With this extension, we bring in Amy a construct that simplify the work on lists especially constructing other lists from already defined ones or combining them. The syntax we add is a sort of *for comprehension* we can find in languages like *Scala*.

2. Examples

You can find a runnable example of our extension in *examples/ListCompr.scala* that showcases what are the new features.

You can find some other small examples of what the extension can do here :

List comprehension on an empty list :

```
val xs: L.List = [ x for x in L.Nil()];
Std.printString(L.toString(xs)) // [ ]
```

List comprehension on a list without condition :

```
val xs: L.List = L.Cons(1, L.Cons(2, L.Cons(3, L.Nil())));
val ys: L.List = [ x for x in xs];
Std.printString(L.toString(ys)) // [1, 2, 3]
```

List comprehension on a list with a condition :

```
val xs: L.List = L.Cons(1, L.Cons(2, L.Cons(3, L.Nil())));
val ys: L.List = [ 2*x for x in xs if (!(x%2 == 0))];
Std.printString(L.toString(ys)) // [2, 6]
```

List comprehension on an expression without a condition :

```
val xs: L.List = L.Cons(1, L.Cons(2, L.Cons(3, L.Nil())));
val ys: L.List = [ 2*x for x in L.concat(xs, xs)];
Std.printString(L.toString(ys)) // [2, 4, 6, 2, 4, 6]
```

List comprehension on a list comprehension without a condition :

```
val xs: L.List = L.Cons(1, L.Cons(2, L.Cons(3, L.Nil())));
val ys: L.List = [ 2*x for x in [x for x in xs]];
Std.printString(L.toString(ys)) // [2, 4, 6]
```

List comprehension on multiple lists without condition :

```
val xs: L.List = L.Cons(1, L.Cons(2, L.Cons(3, L.Nil())));
val ys: L.List = [ x for x in xs if (!(x%2 == 0))];
val zs: L.List = [ x*y for x in xs for y in ys];
Std.printString(L.toString(zs)) // [1, 3, 2, 6, 3, 9]
```

List comprehension on multiple lists with a condition :

```
val xs: L.List = L.Cons(1, L.Cons(2, L.Cons(3, L.Nil())));
val ys: L.List = [ x for x in xs if (!(x%2 == 0))];
val zs: L.List = [ x*y for x in xs for y in ys if (x*y < 4)];
Std.printString(L.toString(zs)) // [1, 3, 2, 3]
```

List comprehension with external variables :

```
val xs: L.List = L.Cons(1, L.Cons(2, L.Cons(3, L.Nil())));
val t: Int = 3;
val a: Int = 2;
val ys: L.List = [ t*a*x for x in xs if (x < t)];
Std.printString(L.toString(ys)) // [6, 12]
```

3. Implementation

We chose to desugar the list comprehensions early in the pipeline, during parsing thus we have only modified the files *Tokens*, *Lexer*, *Parser*, *ASTConstructor* and *ASTConstructorLL1*.

3.1 Theoretical Background

All the theory we used was in the basis of the Computer Language Processing course.

3.2 Implementation Details

We defined first new tokens (*FOR*, *IN*, *LBRACK*, *RBRACK*) to be able to parse the list comprehensions.

We added some non-terminals to the LL1 grammar to construct the AST from the list comprehensions. The non-terminal *ListCompr* represents a list comprehension and is an *Expr* of the highest precedence (*lvl10*). It corresponds to *LBRACK - 'Expr - 'ForIn - 'OptionalForIns - 'OptionalIf - RBRACK*, which means that the list comprehension must have at least one *ForIn* and that the condition is optional (*OptionalIf*). The *OptionalForIns* represents the ability to add zero or multiple *ForIn*s to the first one, which is then handled like a *flatmap* in *Scala*. *ForIn* is defined by *FOR - 'Id - IN - 'Expr*, *OptionalForIns* by *'ForIn - 'OptionalForIns* $|\epsilon$ and *OptionalIf* by *IF - 'Expr* $|\epsilon$.

We decided to transform a list comprehension in a call to a newly created function during the construction of the AST. We gave to the new functions names that could not be tokenized in the Amy language (*++listComprDesuggar#* where # is a fresh integer) thus a function with this name would not already exist. We had to pass these new functions as an *Option* of *List* of *ClassOrFunDef* when constructing the expressions of the program and then add them in the current module's functions.

For every expression that follows the *IN* token (the expression of the list we need to go through), we computed the AST for the expression and gave an unique identifier to the list (*−−ListId#* where # is a fresh integer) for the newly created function to refer to. For example :

```
val xs: L.List = L.Cons(1, 2, 3);
val ys: L.List = [x for x in L.concat(xs,xs)]
```

would translate to :

```
val xs: L.List =
  L.Cons(1, L.Cons(2, L.Cons(3, L.Nil())));
val ys: L.List =
  ListCompr.++listComprDesuggar1(L.concat(xs, xs), a)
```

and then in the function, L.concat(xs, xs) would be represented by *−−ListId1*.

For every list comprehension, we had to check if external variables were used so we went through the expression to output and through the condition expression if there was one to find all of them. We then passed these external variables as arguments of the new function. The name analyzer could then check automatically if the external variable was defined in the scope.

For example, if we write :

```
val t: Int = 1;
val a: Int = 2;
val xs: L.List = L.Cons(1, 2, 3);
val ys: L.List = L.Cons(1, 2);
val zs: L.List = [a*x*y for x in xs for y in ys if x*y <= t]
```

During the AST constuction, we detect that there are two *ForIn*, that there is a condition and that we have two external variables (a and t). We then create these two functions :

```
def ++listComprDesuggar1(−−ListId1: L.List,
        −−ListId2: L.List, a: Int, t: Int): L.List =
{
    −−ListId1 match {
      case L.Cons(x, tail) =>
        L.concat(ListCompr.++listComprDesuggar2(
                −−ListId2, x, a, t),
            ListCompr.++listComprDesuggar1(
                tail, −−ListId2, a, t))
      case L.Nil() => L.Nil()
    }
}

def ++listComprDesuggar2(−−ListId2: L.List,
        x: Int, a: Int, t: Int): L.List =
{
    −−ListId2 match {
      case L.Cons(y, tail) =>
        (if(((x * y) <= t)) {
          L.Cons(((a * x) * y),
            ListCompr.++listComprDesuggar2(
                tail, x, a, t))
        } else {
          ListCompr.++listComprDesuggar2(tail, x, a, t)
        })
      case L.Nil() => L.Nil()
    }
}
```

Here, *−−ListId1* represents the current part of *xs* and *−−ListId2* the current part of *ys*. For each element of *xs*, we then call *++listComprDesuggar2* which goes through *ys* with the current *x* and then concatenate this result and the result of the same process for the rest of *xs*.

We paid attention to the fact that the identifier given (in the example here *x* and *y*) were the same in the pattern matching and as argument to the functions in order to be sure that the expression to output and the condition would pass the name analyzer.

Then the value *zs* is a call to a function like you can see below :

```
val zs: L.List =
    ListCompr.++listComprDesuggar1(xs, ys, a, t);
```

If there was no condition, the *++listComprDesuggar2* would look like that :

```
def ++listComprDesuggar2(−−ListId2: L.List,
        x: Int, a: Int): L.List = {
    −−ListId2 match {
      case L.Cons(y, tail) =>
        L.Cons(((a * x) * y),
            ListCompr.++listComprDesuggar2(tail, x, a))
      case L.Nil() => L.Nil()
    }
}
```

4. Possible Extensions

We finished what was asked in the requirements for this extension.

If we wanted to extend it more, we could do list comprehensions over range of integers, like in Scala. For example, we could write things like :

[x for x in 1 to 3] to have 1, 2, 3,

[x for x in 1 until 3] to have 1, 2,

[x for x in 1 to 3 by 2] to have 1, 3.

We would need to add new tokens (*TO*, *UNTIL*, *BY*) and extend the current grammar with a new expression to handle this new extension. During the AST construction, we would then transform these ranges into the corresponding list of integers before applying the same transformations as we did for normal lists in our extension.

References

We did not use any references outside the theory of the course.