Arrays and range types: Dynamically checked range types

Compiler Construction: Final Report

Maxime Schoemans
EPFL
maxime.schoemans@epfl.ch

1. Introduction

The goal of the initial project was to build a compiler for the Amy language, a simplified version of the Scala language. For this, the different parts of the compiler have all been implemented in Scala during the previous labs.

The aim of this compiler extension project is to add two features to the Amy language and to modify the compiler to take these features into account. The two added features are range types and fixed-size arrays of integers. An important part of the project is to make sure that the bounds on the ranges and arrays are respected, and for this, range checks have to be added during code generation. These range checks, not only have to be made at runtime (dynamically) but also only when necessary.

In this report, a few examples of the use of the features will be given. Then, the changes in the different steps of the compiler will be described and explained. Finally, possible further extensions to this project will be presented.

2. Examples

Adding arrays to the Amy language allows for easier storage of integer lists than by using an endless amount of variables. The following example uses an array to store student SCIPER numbers and prints them.

```
def printFrom(arr: array[15], i: [0 .. 14]): Unit = {
    Std.printInt(arr[i]);
    if(i < arr.length − 1) {
        printFrom(arr, i + 1)
    } else {
        ()
    }
}

val scipers: array[15] = [123456, 654321, 234567, 765432,
345678, 876543, 456789, 987654, 567890, 678901, 109876,
789012, 210987, 890123, 321098];

printFrom(scipers, 0)
```

The range type, that has been added together with the arrays can also be used on its own, for example for limiting the range of values that an integer can take. In the following example, this range type is used to limit the values of the days and months in dates.

```
abstract class Date
case class Day(day: [1 .. 31], month: [1 .. 12], year: Int)
                                                    extends Date
case class Month(month: [1 .. 12], year: Int) extends Date
case class Year(year: Int) extends Date
```

3. Implementation

To implement the previously described features, all the parts of the compiler had to be more or less modified and a new part was added: the range checker. This new addition to the pipeline lies between the type checking and the code generation. It determines whether a runtime check has to be added during the code generation at the places where it might be needed and is thoroughly described in a further section.

3.1 Lexer, Parser and Name analyzer

Changes were made on the first three parts of the compiler to allow for the new constructions that are made possible with the addition of arrays and range types. For the lexer, this concretely means that it has to take two new tokens into account: the left bracket '[' and the right bracket ']', and two new keywords: 'array' and 'length'.

In the parser, the changes were more significant, because of the changes to the grammar. The following

examples are new expressions that are possible in the new grammar, but not in the old one.

```
val x: array[5] = [1, 2, 3, 4, 5];
val i: [0 .. 4] = 4;
x[i]; // array accessing
x.length; // array length call
```

The grammar has thus been modified to take these new possible codes into account, and the AST, together with its constructor has been modified accordingly. Aside from range types and array types, having each respectively a start and end value and a length, a generic array type has also been added to the AST. This is used in the type checking to represent an array of arbitrary size. The arrays are considered to be array literals, this means that they are, just as strings, immutable and compared by reference. Lastly, expressions for array accessing and array length calls were also added in the AST. In the rest of the report, array length calls will be called ALC's.

3.2 Type checking

In the type checking, the constraint generation has been modified to handle array literals, array accessing and ALC's but works the same as before. For array accessing and ALC's, the generic array type is used as the expected type of the variable to denote an array of unspecified length. For array literals and variable definitions, on the other hand, array types of specified length are used. This means that the type checker differentiates between arrays of different length. Errors such as calling a function with an array of incorrect length or defining an array variable with an array of incorrect length, are thus already thrown here.

In the constraint solving, a new function has been added. This function is used to decide whether an expected type and a found type are compatible or not. For arrays, for example, generic array types are compatible with array types of all lengths. Array types of specific length are compatible with generic array types and array types of the same length, but not with array types of different lengths. In the type checking, all range types are also compatible with int type. It is in the next phase, the range checking, that range types and int types will be handled differently. For the rest of the types, the equality is used to determine compatibility, just as previously.

3.3 Range checking

The range checking, is a new phase in the pipeline, between the type checking and the code generation. It is used to determine if a dynamic check has to be added during the code generation. A range error can happen only in a few cases: during array accessing, when defining a variable of range type or when calling functions or constructors with array type or range type parameters. A new boolean attribute 'rangeCheck' has thus been added in the tree module for array accesses and for variable definitions. For function and constructor calls, the same boolean attribute has been added to each argument.

The range checker takes as input a program and symbol table and returns a new program, with the modified 'rangeCheck' attributes, and the same symbol table. To modify the 'rangeCheck' attribute, it goes through the whole program recursively, while keeping track of an environment variable. This environment maps every variable to its corresponding type. When the range checker encounters either array accesses, variable definitions or function/constructor calls, it calls the function 'rangeCheckExpr' to verify if a dynamic range check will be needed or not.

'rangeCheckExpr' takes an expression, an expected range type and an environment as parameters and returns *true* if a dynamic range check is needed and *false* when it is certain that the expression will result in an integer in the expected range. It also throws an error if it is certain that the expression will be outside the expected range. To know the range of integers that the expression could take, it calls the function 'getType'.

The function 'getType' takes an expression and an environment and computes the result type of the expression. Since only the array types of specific lengths and range types are important for the range checks, the rest of the types are denoted with a generic type defined in the range checker. Integer literals are here considered as having a range type starting and ending at their integer value and the same applies for ALC's, which are range types with start and end value equal to the length of the array. The function also handles basic math with the plus and minus operations. This allows for expressions like the following to be accepted without dynamic range check.

```
x[x.length − 1]; // no dynamic check
x[2 + 2]; // no dynamic check if x.length >= 5
```

3.4 Code generation

The way array literals are stored in memory is comparable to a constructor, with each field being an element of the array. The counterpart of the index of the constructor is the length of the array. The representation on the stack of an array literal is thus a pointer to the address containing the length of the array. The integer elements are then in order at the subsequent addresses. Range types are basically integers and are thus stored as such.

The rest of the code generation is essentially the same, except for the fact that dynamic range checks are added where the range checker estimated it to be necessary. These range checks compute the value of the expression and compare the result with the start and end value of the expected range. If the result is in bounds, it will just continue, and it will throw a range error if this the result is out of bounds. The expected range for arrays is from zero to the length of the array minus one, and for specific range types is just the specified range itself.

4. Possible Extensions

Even with the addition of arrays and range types, the possibilities of the Amy language are still very limited and a lot of improvements could be made to these arrays to make them more practical.

On another note, the 'getRange' function could also be improved, to take more complicated expressions into account, like divisions, multiplications or even function calls, and thus possibly lowering the number of dynamic checks in the generated code. This, however, could increase the compiling time drastically and this has thus not been done in the current state of the project.

For the improvements to the arrays, there are many that we can think of, but only a few important ones will be discussed below. The first example is allowing the user to use the generic array type in the code, and thus to be able to write functions that take arrays of any size as a parameter. With this in mind, we could rewrite the function above that prints an array of length 15 to be able to print arrays of any size. This function could look like this.

```
def printFrom(arr: array, i: Int): Unit = {
    if(0 <= i && i < arr.length) {
        Std.printInt(arr[i]);
        printFrom(arr, i + 1)}}
```

Another improvement would be to allow the creation of arrays of any type. This would mean that not only integer arrays are possible, but also string arrays, boolean arrays or possibly even abstract type arrays, like an array of students or courses.

A limiting factor in the use of arrays is the fact that they are considered literals and that they thus have to be defined statically. Arrays could be drastically improved by adding array concatenation, array indexing and dynamic array definition or modification. This, together with the previous improvements, would allow for arrays of any type, any length and greatly improved potential, similar to arrays in common languages like java. [1]

[1] Of course, loops would be nice too, since going through a 3D-array with recursion is not the easiest idea.