Improved parameters

Compiler Construction Final Report

Théo Abel   Antonio Jimenez

EPFL

theo.abel@epfl.ch
antonio.jimeneznieto@epfl.ch

1. Introduction

Amy is a functional and interpreted programming language. We started by creating the Interpreter. It is in charge of reading the code we write and interpreting its meaning.

After that, we worked on the Lexer, which reads the text of our code and converts it into a list of tokens, removing all unusable information (i.e. whitespaces, comments).

Then, we studied the Parser, which takes the sequence of tokens produced by the Lexer and transforms it into an Abstract Syntax Tree (AST). For this, we decided to use the LL(1) parsing algorithm as it is able to parse the inputs in linear time.

Subsequently, we implemented the Type Checker. Once our AST is successfully constructed, the compiler runs a type analysis to prevent errors based on the form of the values manipulated by the program (i.e. prevent an integer from being added to a boolean value).

Finally, we learned about Code Generation. Our target language is WebAssembly and we execute the bytecode output directly using Node.js.

Considering our extension, we have decided to work on Improved parameters. With this, we want to increase the readability and maintainability of the code written in the Amy programming language.

2. Examples

We define improved parameters by the capability to first of all to name the parameters and secondly to set a default value on the parameters in both functions and case classes.

Improved parameters allow us to increase the readability of our code, thus improving its maintainability.

```
    val acc1: Account = Account(0, 1000);
    val acc2: Account = Account(1, 0);

    transaction(first, second, 1000, 0.8);
```

Unfortunately, this piece of code is not easy to read. We do not know what the attributes of the Account class do, nor do we know what the attributes of the transaction function do. However, if we use the improved parameters:

```
    val acc1: Account = Account(id=0, balance=1000);
    val acc2: Account = Account(id=1, balance=0);

    transaction(sender = first, receiver = second,
                amount = 1000, fees = 0.8);
```

Now, we can read the code and understand what each parameter does without having to go to the Account class to understand it.

Here is another example that illustrates in more detail all the features of the improved parameters:

```
object example
    abstract class Foo
    case class Bar(i: Int(32), j: Int(32),
                   b: Boolean = false) extends Foo

    fn foobar(i: Int(32), j: Int(32),
              b: Boolean = false): Int(32) = {
        if (b == true) {42} else {i + j}
    }

    val v1: Foo = Bar(0, j = 1);
    v1 match {
    case Bar(x, y, b) => Std.printInt(foobar(x, y, b))
    }; // 1

    val v2: Foo = Bar(0, 0, b = true);
    v2 match {
```

```
    case Bar(x, y, b) => Std.printInt(foobar(x, y, b))
    }; // 42

    val v3: Foo = Bar(j = 1, i = 2, b = false);
    v3 match {
    case Bar(x, y, b) => Std.printInt(foobar(x, y, b))
    }; // 3

end example
```

3. Implementation

For the implementation of improved parameters, we modified two stages of the compiler pipeline: the parser and the name analyzer.

3.1 Theoretical Background

The LL(1) parsing algorithm is used by the Parser to transform a sequence of tokens, produced by the Lexer, into an Abstract Syntax Tree. The new grammar G for improved parameters will thus have to be LL(1), that is:
1) G is a context-free grammar.
2) For each nonterminal X, first sets of different alternatives of X are disjoint.
3) For each nonterminal X, if nullable(X) then first(X) must be disjoint from follow(X) and only one alternative of X may be nullable.
Having an LL(1) grammar enables the use of the LL(1) parsing algorithm to parse inputs from left to right without the need for backtracking.

3.2 Implementation Details

To develop our extension, we will need to first parse the new data contained in the file, store this data in the tree nodes, and finally utilize it to properly initiate the call. We will outline the steps we have modified in the pipeline, from beginning to end.

3.2.1 Parser

Our first modification to the compiler pipeline comes with the Parser. We decided to modify and expand 2 rules: the rule for parameters and the rule for arguments.
We introduced restrictions to the grammar to make it unambiguous. As specified in the extension, default parameters must be defined at the end. The new parser will accept the first definition in the example below, but reject the second definition.

```
// Definition accepted by the parser:
fn foo(i: Int, j: Int = 42): Int = {...}
// Definition rejected by the parser:
fn foo(i: Int = 42, j: Int): Int = {...}
```

Moreover, a similar specification is implemented for the placement of arguments in function calls (i.e., all named arguments must be placed at the end). This restriction was introduced to eliminate the following ambiguity:
Consider the function foo, defined as shown below. The first call is unambiguous: the value 5 is assigned to i, the value 12 is assigned to j, and the value 7 is explicitly assigned to k. However, in the second call, there is an ambiguity: k is explicitly assigned the value 7, but it is unclear whether i should be assigned the value 5 (as it is the first unnamed parameter after k) or whether j should be assigned the value 5 (as it is in the same position as in the definition). To prevent this type of ambiguity, the parser has been designed to reject calls that have named arguments that are not placed at the end, such as the second call shown below.

```
fn foo(i: Int, j: Int, k: Int = 42): Int = {...}
// Call accepted by the parser:
foo(5, 12, k = 7)
// Call rejected by the parser:
foo(k = 7, 5, 12)
```

With these restrictions in mind and to ensure that the grammar remains LL(1), the following grammar has been derived:

```
parameters :=
    parameter ~
    opt(valueThenDefaultParameters | moreParameters)

valueThenDefaultParameters :=
    "=" ~ expr ~ opt("," ~ defaultParameters)

moreParameters :=
    recursive {
        "," ~ parameter ~
        opt(valueThenDefaultParameters | moreParameters)
    }

defaultParameters := repsep(defaultParameter, ",")

defaultParameter := identifier ~ ":" ~ type ~ "=" ~ expr

parameter := identifier ~ ":" ~ type
```

Note that the rewritten rule for arguments follows the same logic, and is not shown here for the sake of brevity.

3.2.2 Tree Module

We have parsed new information in the previous section, we now need to store this extra information in the tree nodes. We decided to modify 3 nodes of the nominal tree.

The first node ParamDef represents the parameters in a function definition.

Initially, all 3 nodes were implemented in the TreeModule trait. We decided to split the node implementation into two different trees: the NominaleTreeModule and the SymbolicTreeModule. This allows us to store the new parsed information in the Nominal Tree nodes. In the Nominal Tree, ParamDef now takes a new 3rd argument: an optional expression representing the value of the default parameter. This modification allows us to store the default parameters given in a function's definition. CaseClassDef sees its second argument fields modified. They now take a list of ParamDef instead of a list of TypeTrees. This now allows us to store the name, type and the optional default parameter of the fields. With this modification, case classes can now have default parameters. Finally, Call sees its second argument args modified to take a list of pairs. Each pair contains the expression (value) given to the call, and the new optional name of the argument. This last node modification lets us use named arguments during function and case class calls.

3.2.3 Name Analyzer

In order to minimize the modifications required to add support for named and default parameters, we decided to evaluate the arguments during the name analysis phase. At this point in the process, all necessary information is readily available to accurately transform the nominal tree into a correct symbolic tree. Thus, we do not need to modify any phase coming after the Name Analyzer. In previous sections, we discussed the storage of named arguments and default values. However, this alone is not sufficient for transforming the nominal tree of a call into a symbolic tree. It is also necessary to store the nominal description of a function or case class during the discovery process, so that it may be used during the transformation phase. This allows us to iterate over all parameters in the definition during the transformation, and apply either the unnamed argument, the named argument, or the default parameter as appropriate. It is also important to verify that all named arguments are used only once. The Name Analyzer will reject the example shown below.

```
fn foo(i: Int, j: Int, k: Int = 42): Int = {...}
// Rejected during name analysis:
foo(i = 5, k = 7, i = 12, j = 3)
```

The example below illustrates the transformation from a nominal call node to a symbolic call node. We first create a new array the same size as the parameters in the nominal description of the function. This new array will contain the arguments in the correct order for the symbolic tree. We begin by iterating over all unnamed arguments in the nominal tree and place them at the same index in the symbolic arguments array. We then loop over all named arguments given in the nominal tree and place them at the correct index in the symbolic arguments array. If a number is already present at the index, an error is thrown for having too many arguments. (See rejected example 1 below). Finally, we loop over the symbolic arguments array to verify all arguments have a value. If an index does not have a value, we check if we can apply a default parameter given in the parameters of the nominal tree. If none is found, we throw an error for not having enough arguments

```
fn foo(i: Int, j: Int, k: Int = 42): Int = {...}

// A function call:
foo(j = 12, i = 5)

// Nominal tree
// Arguments : [(j, 12), (i, 5)]
// Parameters : [(i, None), (j, None), (k, 42)]

// Symbolic tree
// Symbolic Arguments : [5, 12, 42]

// Rejected example 1
foo(1, i = 5, j = 12) // i was given 2 values : 1 and 5
```

3.2.4 Symbol Table

In order to improve usability, we have implemented a function that allows us to locate the module of a constructor based on its name within the Name Analyzer. This capability has been added specifically for function and case class definitions, enabling the user to omit the explicit specification of the module. This feature allows us to search for the owner of a constructor in cases where the owner is not explicitly provided.

```
object L
abstract class List
case class Nil() extends List
case class Cons(h: Int, t: List = Nil()) extends List

// Before the modification
fn sum(l: List = L.Nil()): Int = {...}

// After the modification
fn sum(l: List = Nil()): Int = {...}
```

4. Possible Extensions

A possible extension to 3.2.4 would be to add a compiler option to activate or deactivate this feature. In large programs, having to search the symbol table each time for the owner of a constructor is time consuming. Having the user explicitly name the module owner is a code optimization.