Improved Parameters

Compiler Construction Final Report

Marc Egli Louis Perrier
EPFL

1. Introduction

In the first part of this project, we designed a compiler for a simple language called Amy. This compiler was built as a pipeline, consisting of:

- A lexer, which takes the whole program as a text file, removes white spaces and comments and produces token for each component of the program (operators, keywords, delimiters...)
- A parser which, given the list of tokens, checks that this list of tokens indeed forms a valid Amy program, and builds an Abstract Syntax Tree of the structure of the program (variable definitions, functions...)
- A name analyzer which, given the nominal AST, returns a symbolic AST where names have been resolved to unique identifiers according to the scoping rules, and rejects programs that violate the Amy naming rules.
- A type checker which check if the program follows the Amy typing rules.
- A code generator which generates Web Assembly code corresponding to the program.

The goal of our extension is to help an Amy programmer to write a program by giving him more flexibility over class fields and function parameters.
We have introduced two types of parameter improvement:

-Default parameters: We want the programmer to be able to give values to some parameters upon class/-function declaration. This way, when a function takes an argument that will have a fixed value in most cases, the programmer won't have to put this value as an argument at each call of the function.
For example, a function printing some string to an output stream could take as a default argument the standard output stream (the user's screen).

-Named parameters: We also want the programmer to be able to specify the names of the arguments upon function call / object creation.
This allows to make a program clearer by naming all arguments, to help someone reading the program understand what a function call corresponds to without reading the function definition.
This can also be combined with default parameters to give the programmer the possibility to specify which default parameter he's overwriting.

2. Examples

Example of an use of default parameters:

```
def foo(a: Int, b: Boolean = false)

//...

foo(2)
```

Since b has false as a default value, we don't need to specify it once calling the function, unless we want b to be true. Lots of programs have function that takes a boolean value that, when true, adds a secondary effect to the function, but that is false most of the time. So putting false as a default value simplifies it.

Example of an use of named parameters:

```
abstract class Shape
case class Rectangle(length: Int, height, Int, width: Int,
    red: Int, green: Int, blue: Int)

//...

val redRectangle: Shape = Rectangle(
```

```
    length = 10, height = 5, width = 5,
    red = 255, green = 0, blue = 0)
```

Creating redRectangle without naming the fields would have make the program difficult to read, now it's easy to understand which value corresponds to which field. Of course we could also have reorder the arguments, since the fact that they are named prevails over their ordering.

Example of an use of both default and named parameters:

```
def bar(a: Int, b: Boolean, c: Int = 2,
    d: Int = 10, e: Boolean = false)

//...

bar(10, false, d = 5, e = true)
```

Here, d and e are overwritten but c keeps its default value. Without naming d and e when calling bar, 5 would have been assigned to c and true would have been assigned to d, resulting in a typing error since d is an Int. The only way to make it work would have been to specify that c is equal to 2, which would have been useless because it's its default value, and would break all the interest of putting default values to parameters.

3. Implementation

To make our extension work, we've had to modify two stages of the pipeline: The parser and the name analyzer.

3.1 Theoretical Background

When modifying our parser, we had to make sure we were keeping our parser LL(1).
A LL(1) parser parses the input from left to right, without backtracking.
It consists of a set of rules over tokens. When parsing a program, a rule is chosen if:
- the rule can start with the next token in the program, or if
- the rule is nullable (can contain no token at all), and can be followed (in some other rule) by a rule that can start with the next token.

Having an LL(1) parser is important because it allows not to have ambiguities in our grammar (our set of rules), because a sequence of tokens cannot correspond to two different rules.
Note that a non-LL(1) grammar is not necessarily ambiguous, for example:
A ::= B | C
B ::= xy
C ::= xz
This set of rules is not ambiguous, but it does not form an LL(1) grammar since B and C both start with token x, so if we're parsing a token x and are considering rule A, we cannot know if we have to chose rule B or C with our LL(1) parser.

3.2 Implementation Details

3.2.1 Parser

First of all, we've had to modify some of our nominal AST nodes to take our extension into account (note that we also modified the given Printer to make it print our extension and be able to debug more easily). The modified nodes are the followings:
- a **ParamDef** (node used for the parameters of a function definition) now has a third field, in addition to its name and type, which is its default value. We've made this field optional, since we do not want all our parameters to have a default value. Note that ParamDef are also used for variable definition (node Let). To keep it simple (and not having to create a new AST node), we just ignored this field in the case of a Let in our pipeline.
- a **CaseClassDef**, which formely took a list of types for its fields, now takes a list of ParamDef, since we now want to keep track of the names and possible default values of the fields.
- a **Call** formerly took just a list of Expr for its arguments. To be able to name the arguments when calling a function, now the arguments are a list of pairs consisting of an optional Name (for the argument) and an Expr (for its value).

Next we've had to modify two rules in our parser: The rule defining parameters (which is used both for function and class definition), and the rule defining the args of a call.

- For parameters, we wanted to make sure that the parameters that do not have a default value are located before the one having a default value.

Implementing this rule while keeping our parser LL(1) was a bit tricky, since a parameter with a default value and a parameter without one both start with the same token, which is the name of the variable. We have created two syntaxes, one representing a coma ”,” followed by a list of parameters, and one representing an equals ”=” followed by an expression and a list of other default parameters. Both this syntaxes return a pair of an option of Expr and a list of ParamDef. The option is there to differentiate between the two cases, it will contain the value in case of a default parameter and None in case of a non-default parameter. This way, when parsing parameters, we can simply look at the value of this option to determine whether we are dealing with a default parameter or a non-default one.

-For the arguments of a call, we wanted to be sure that there are no ambiguities in the order of the arguments that the programmer wrote. To illustrate a possible ambiguity, let's consider this example:

```
def foo(a: Int, b: Int, c: Int,
    d: Int = 10, e: Boolean = false)

//...

foo(12, d=15, 18, 35)
```

How should we assign arguments to parameters? Of course the first one is assigned to a since they share the same location in the list, then the second argument is assigned to d since its name has been made explicit, but what about the third one? We could assign it to b since it's the next parameter that has not been assigned, or to c since it shares the same location in the list of parameters.
To solve this kind of ambiguity, we've chosen the following rule: arguments with explicit names should be located after the unnamed ones. Hence we parsed our arguments taking this rule into account.
To keep our parser LL(1), we used a similar trick to the one used for parameters. There is a little difference though, because an unnamed argument and a named one don't start with the same token (an Expr for an unnamed argument, and a Variable for a named one). However, this does not make parsing easier, because an Expr can be a variable, or can start with a variable. What we did was handling separately the case where the first token of the argument is a variable, and then check whether there is an ”=” and a value after (the same way we did for parameters). If it's the case, then the variable is the name of an argument. Else, the variable is passed as an argument. If we have an expression that is not a variable followed by an ”=”, we report an error because this would'nt mean anything.

3.2.2 Name Analyzer

Before all, we had to make a decision about when the default values of parameters should be evaluated. A first option would be to evaluate them at the function/-class definition. A second option would be to evaluated them only when the function is called, or once an object is created. We chose the second option, since it allows to evaluate only the default values we do not overwrite, and not all of them.
Another reason for this choice was that it allows us to perform the whole overwriting process during the name analysis phase, by simply taking the nominal node for a call, keeping the unnamed arguments as they are, reorder the named arguments to put them in the same order as their corresponding parameters by looking at their names (and report an error if the name of an argument doesn't match the name of a parameter) and replace the unspecified arguments by their default value (and if an argument is not specified and has no default value we report an error).
This allows us to keep the symbolic AST nodes as they were before the extension, and not having to rewrite the entirety of the pipeline.

Here is how me modified the name analyzer to make this work.

We first modified the discover process. When we discover a new CaseClassDef or FunDef we add them to a new Map which maps the pair of there name and owner to the corresponding nominal description. We have to create and fill this map such that when we transform a nominal call to a symbolic call, we first access the nominal description of class/function that is called. This gives us access to the complete list of parameters required (default and not default). Knowing the complete list of parameters helps us now determine if the call is valid or not, this is so because a call can now contain less arguments than the declaration specifies but can still be valid.

The validation happens in the following way.

We take the list of the complete number of arguments required from the nominal definition and create two maps. The first one maps the position of the arguments to their names and the second one maps their names to an optional expression. The default parameter will already have an expression mapped to it. After that we resolve from the call the corresponding expressions for the arguments. If after this we still have a name that maps to none we abort. This means that the call isn't valid.

Once the validation is complete we transform the nominal call into a symbolic call with the updated values from the default parameters and the resolved expression for every argument.

4. Possible Extensions

A way to further extend our compiler would be to implement lazy evaluation. A "lazy" variable is not evaluated when it's created, however it's evaluated only once (the first time it's used).
This would improve our extension indeed, because, as we said in the first paragraph of **3.2.2**, we've chosen to evaluate the default values of our parameters upon function call / object creation. However, if we call a function several times, this would mean that the default value is evaluated at each call, which is not optimal. With lazy evaluation, we could evaluate the parameter only at the first call, and we would not have to evaluate it again later.