Compiler Extension Project

Dead code elimination

Hugo Decroix 259022

EPFL

hugo.decroix@epfl.ch

1. Introduction

During the first part of the Amy compiler project, I implemented a basic compiler for the Amy language. Starting with a basic Scala interpreter at first. Then, I implemented a Lexer. The goal was to split the user code into tokens eliminating the comments lines, white spaces and other useless stuff to the compiler. The challenge for this part was to tokenize and identify name variable from code names such as: “if”, “else”, “val”, “def”, etc. Then, I implemented a Parser. The goal of the Parser is to generate a tree for the previously computed list of tokens. For this I needed a LL1 grammar language that described and respected all the behaviours of the language and then generate the parse tree by reading the tokens and checking the language. Then I implemented an Analyser. The first part of this analyser is the name analyser which goal is to resolve every name into identifiers and check if the naming rules of the Amy language are respected. The second part of the analyser is the type checking since Amy is language with types. In this part, we check if the typing rules of the Amy languages are respected. All this analysing part is done by reading and modifying (name analyser) the previously build parse tree. Then at this point, we know that the code provided by the user is correct and respect Amy’s rules, the last part is hence machine language code generation, in this case we generate Web Assembly code. This part is done by reading the parse tree and generating the corresponding code.

The goal of this project is to add a dead code elimination extension. For this, we only need to modify the code generation part by adding some checking before the code generation for a new value. Since this extension is more an optimization than a new language functionality to the user, the Lexer, the Parser, and the Analyser need to behave in the exact same way and therefore no modifications are necessary for those parts of the compiler architecture.

2. Examples

The goal of dead code elimination is to eliminate useless val from code generation if the created val is not used in the user given code. This prevent expensive calculation for nothing that can be present in the code by mistake or because of bad deletion of code lines. But we

also need to make sure that the alone val was not placed intentionally by the user for its side effects on the program. Those side effects can be behaviours such as print statements, read statements, error statements. Those statements need to be included in the program even if they are generated during the computation of an unused val.

Here are some examples to demonstrate those cases:

```
object Example0 {
  val a = 1 + 1;
  a + a;
  Std.printString("a generated !")
}
```

Here the val “a” is generated since the val is used after its statement.

```
object Example1 {
  val a = 1 + 1;
  Std.printString("a not generated !")
}
```

Here the val “a” is not generated since the val is not used after its statement.

```
object Example2 {
  def withError(b : Int) : Int = {
    if(b < 0){
      error("less than 0")
    }
    else { b }
  }

  val a = withError(-1);
  val b = withError(2);
  Std.printString("a and b generated!")
}
```

Here the vals “a” and “b” are generated even if they are not used after there statements. This is because there is potential side effect for the program during the computation of there values. In the case the side effect is the potential throw of an error with the call of the function withError().

```
object Example3 {
  def withPrint(b : Int) : Int = {
    if(b < 0){
      Std.printString("less than 0")
      b
    }
    else { b }
  }

  val a = withPrint(-1);
  val b = withPrint(2)
  Std.printString("a and b generated!")
}
```

This example is similar to the previous one, the only difference is that the side effect is a print statement instead of an error statement. This will not stop the program, but it will still print a message in the console. Therefore “a” and “b” need to be generated.

3. Implementation

As stated previously to implement this optimization to the compiler, we only need to modify the code generation part. For this we need to add some checking before the generation of a new val.

The first checking to be done is the usefulness of the created val. To deal with this checking I implemented a helper function:

```
def isUsed(expr: Expr, name: Identifier): Boolean
```

The goal of this function is to make a big logical OR statement on the body expression coming after the val creation and verify if the provided name is used or not. For this, the implement match on the given expr, check the name correspondence if the expr is a variable call or simply call the function recursively on the sub-expressions of the given expression.

The second checking to be done is the checking of potential side effects on the program during the value computation. For this, I implemented another helper function:

```
def isClean(expr: Expr): Boolean
```

Similarly, the first helper function, the implementation match on the given expr and check if there exist side effects statements. The side effects are all print calls in the Std library and the errors statements. Therefore, this helper function corresponds to a big logical AND statement between all recursive call on each sub-expression of the given expression.

With those two helper functions, we can easily implement the val code generation:

```
case Let(df, value, body) =>
  if(isUsed(body, df.name) || !isClean(value)){
    // Case where the variable definition is not dead code
  }
  else {
    // Case where the variable definition is dead code, we simply generate
code for the body
  }
```

For this we perform a check of usage of the val in the body expression and a check of potential side effects in the value expression. If the constraints are respected, we generate the val and add it to the locals otherwise we are in the case of a dead code and we can simply generate the code for the body.

4. Possible Extensions

They are plenty of other final code optimizations that can be added to this compiler such as elimination of useless if statement or creation of local values for values computed multiples. The important part of those extension is checking all the potential side effect on the program before deleting modifying the code given by the user.