Friendly Documentation

Erdem Yildirim

Alles sollte so einfach wie möglich gemacht werden, aber nicht einfacher.
Albert Einstein (1879-1955)

1 Introduction

Here I will documentate the overall structure of this compiler and explain the *analysis* and *translation* phases in detail. The emphasis will lie on compiling object-oriented programming elements; compiling arrays are not included. Since this compiler is an educational one, it forgoes some complexity. This enables me to make my design decisions understandable within a brief document[1].

This implementation consists of only compiler front-end, which has 4 phases:

- *Lexical Analysis*
- *Syntax Analysis*
- *Name and Type Analysis*
- *Intermedate Representation*

[1] The control-flow sensitive analysis is not fully implemented. In class **Analysis** you will see three methods labeled with *CFS*, you can ignore them. This document will not describe those methods.

Lexical and syntax analysis are not the main subject of this documentation, so I will skip their explanation. You can find their implementation in package **frontend**.

2 Name and Type Analysis

A compiler uses a type representation to determine whether program expressions satisfy the given type rules. Furthermore, we want compiler to show us all possible type inconsistencies. To this end, compiler presents types for correctly-typed as well as ill-typed program elements.

First I will explain how types and classes are represented, then proceed to analysis.

Representing Types

```
Type =
    |int
    |bool
    |null
    |any
    |ArrayType{type} // has a base element type
    |ClassType
```

As seen above, we have an abstract super class **Type**. For all the types we have a subtyping relation, therefore

Type provides the method **isSubTypeOf()**. All types must implement this method.

The types **int, bool, invalid, null, any** are represented by exactly one object to allow compalrison via "==". Furthermore, they are instantiated as static fields inside the abstract class **Type**, so that accessing them via **Type.INT, Type.BOOL ...** is possible.

We use the type **any**, when the type of an expression unsonund declared. By doing this we secure the continuation of type analysis; leaving that expression typeless will be error prone, since it might be present in other parts of the program.

Each instance of **ArrayType** has a base type. This base type is again an instance of **Type** except **ArrayType**.

Representing Classes

We have three different models for representing declared NQJ classes. They serve as different abstraction levels.

- **ClassRef** gives information about declared class without referring to class variables or methods.
- **ClassType** provides typing relations for a particular class.
- **ClassContext** expresses the class variables and methods.

All declared classes has a **ClassRef** object. All **ClassRef** objects has a **ClassType** object as field. Class type is created in the constructor of the **ClassRef** object for once.

ClassType has two **ClassRef** objects as fields for some class and its direct super class. The method **classTypeEqualTo(other)** takes a **Type** object and checks for type equality. Since for each **ClassRef** object there is a unique **ClassType** object as a field, declared classes have their own **ClassType** objects. Because of that we cannot use "==" for equality checking between **ClassType** ojects, instead above method checks the name equality of classes.

ClassContext implements a class reference and two hash maps for variables and methods:

Map *String-Variable*

Map *String-Method*

These maps are useful during analysis, when we try to get the NQJ node of a variable from its name, or when we need the method variables only knowing the method name.

Analysis

```
// overall structure with pseude code
NQJFrontend frontend = new NQJFrontend();
```

```
NQJProgram javaProgram = frontend.parse(input);
// [..]
Analysis analysis = new Analysis(javaProgram);
analysis.check();

// pseudo code for analysis.check()
createClassTable();
createNameTable();
verifyMainMethod();
javaProgram.visit();
```

The class **Analysis** takes the AST representation of given NQJ program and performs name and type analysis with method **check()**. During the analysis, type errors are listed and can be obtained via **analysis.getTypeErrors()**. We are following these steps for analysing:

1. *Class Names:*

   There is a **ClassTable**, which represents consistent classes. The constructor of **ClassTable** checks all class names, extension names and cyclic dependencies. Checking these properties in the constructor enables us to create **ClassTable** object, without any further explicit modification of that object. There are two maps:

   helperMap: *String-Node*

   actualMap: *String-ClassRef*

We have two maps, since checking extension names demands all class names to be present. First we make class names present in the helperMap. To this end we use a map to easily reach the class node from class name.

2. *Function Names:*

   The class **NameTable** checks names of declared global functions in its constructor. This enables us creating a **NameTable** instance without any explicit modification.

3. *Verify Main:* This is made in class **Analysis** by **verifyMainMethod()**. Here we check whether method **main** is present.

4. *Visitor Run:*

   The class **AnalyseProgram** visits the given AST nodes and accompanied by **ExprChecker**, which implements a *matcher* design. We are matching the visited expressions with their correct appearance and checking their types. Further, the visitor checks method and field names, when class nodes are visited. (See the algorithms in section 4.)

3 Translation

```
// overall structure with pseudo code
```

```
Analysis analysis = new Analysis(javaProgram);
analysis.check();
// [..]
Translator translator = new Translator(javaProgram);
llvmProg = translator.translate();
```

Analysis takes the AST nodes and decorates them during type checking. Then this decorated syntax tree is given to translator. Particularly we added these informations to AST:

1. Proper **VarDecl** nodes are set to **FieldAccess** and **VarUse** expressions.
2. We have set class nodes their direct super class.
3. **MethodCall** nodes have now their proper **FunDecl**.
4. We can now obtain the class node from **NewObj** nodes.

Translation cannot be started, when the analysis reports errors. Translation has four steps as seen below. This sequence is chosen becauase of the cyclic dependency of the tasks:

> *Main method needs functions and classes to be present. Functions need classes to be present, and classes need functions.*

As a solution, we are separating class translation to initializing and translating; situating the function translation between them.

```
// pseudo code for translator.translate()
initClasses();
translateFunctions();
translateClasses();
translateMain();
```

Initialising Classes

This step includes field, method and constructor inititialisation for declared classes. *ClassTranslator.initializeClasses()* realises this step. The following is the order we proceed. Items *1.* and *2.* can be combined together, but I choosed to separate them to get a better step-by-step implementation.

1. Creating the struct types of all classes without any fields, and adding them to llvm program.
2. Adding fields to class structs without inherited fields. Also creating procedures with empty basic blocks for all class methods, and adding them to llvm program.
3. Examining field inheritance between classes and adding inherited fields to class structures. Inheritance examined after the addition of class fields, since the implementation recursively checks over class structures to get the field sequences precisely.
4. Creating the constructor procedure for each class and adding to program.

In particular, we are not creating any virtual method tables for classes; since, while generating the llvm code, the compiler statically checks which method must be called. Overridden methods are created as new procedures and added to program. We statically check, if an overridden method procedure or normal procedure must be called in the program context. (See the algorithms in 4.)

Translating Functions

First, procedures with empty basic blocks are created from global functions. Then, function bodys are translated statement by statement. (Algorithms in 4.)

Translating Classes

Since class structures are already created with fields, we only translate the class methods. (See 4.)

Main Function translated as last, since it assumes all other declared classes and functions to be present.

4 Used Algorithms

This section describes some of the used algorithms. Only the following algorithms are documented, since we want to emphasize the algorithms, which are relevant to object orientation, e.g. visiting class declarations can be described here, but visiting if-statements not.

4.1 Analysis Phase Algorithms

1. **Class Names, Extensions, Cycles:**

   (See **ClassTable** constructor).

   We first captured all class names in a map_1 from name to class node. During this we found possible name duplication errors. Then we proceed to function **checkExtensions**, which takes the class node list. This function iterates the given list and finds error, when extension names are not present in map_1.

   After that the function **checkCycles** is called in the constructor. It takes the class nodes and iterates all nodes, if a class has an extension, then **chainDependency** function is called with the class node and a new set of nodes. Function's work is:

   (a) Adds the parameter node to the given set.

   (b) Recursively opens itself with super class node and given set, until it founds a cycle. (A cycle is found, when a class node cannot be added to the set.)

   (c) Reports the error.

2. **Visiting Class Declaration:**

   We have a **ClassTable** object at hand, which represents the consistent classes. This object was created before the visiting. Visiting a class is as follows:

(a) Lookup the class reference in **ClassTable** object. Note that the compiler also visits classes with duplicate names. For example, if there are two declared classes with name A, compiler visits both of them. However, class table contains only one class reference to A. This means, the fields and methods of both of the classes are added to that reference, hence the reference is overloaded.

(b) Creates a class context with empty field and method lists.

(c) Checks names of the fields and adds to context.

(d) Checks the names of the methods without adding to context. Also checks, if method is an overridden method.

(e) Sets current class context to this context.

(f) Visits all class methods.

(g) Sets current class to **null**, which means class context ends.

3. **Checking Method Call:**

   A method call has a *receiver* and a $function$. First check, if receiver is a class. If not, then returns the **any** type. Receiver is class, then:

   (a) Check, if *receiver* object has a declared class.

   (b) Search the $function$ in the *receiver* class including super classes, since function may be inherited. To this end, **examineClassesMethod** is called, which searches the method lists of given class node including its super classes.

(c) If no appropriate method found in the *receiver*, then reports error. Else checks the arguments for inconsistencies.

4. **Checking Function Call:**

   There are global functions and local functions of classes, which are methods. See the following code written in NQJ:

   ```
   int fun() { return 0; }

   class A {
       int fun() { return fun(); }
   }
   ```

   The local function will return the local **fun()**, not 0. We don't need to write **this.fun()** to refer to the local one. This is the language specification.

   The algorithm starts with checking if the function call exists in a class context. If there is a class context, checks the function names without considering inherited functions. If there is no method found in the class, then **globalOrInheritedFunCall** is called. This determines whether the function is an inherited or global one, also checking first the inheritance case. Appropriate errors are obtained in the meanwhile.

5. **Method and Field Inheritance:**

   Two methods are taking this task: **examineClassesMethod** and **examineClassesVar** in **ExprChecker**. They have the same logic and it is:

   (a) Taking the method/variable name and the class type.

   (b) Class type has two references: *class* and its *super*. Search the given name in *class* method/variable list.

   (c) If nothing found, get the class type of *super* (All classes has a unique object for their type.) and recursively open the function with given name and class type of *super*. Appropriate errors are obtained.

6. **Checking Override:**

   With function **searchOverride** in class **AnalyseProgram**, we try to find out whether a method is an overridden method. **searchOverride** takes the class reference and the *method* declaration and searches the super classes for the same method name. If method with same name is found, which means *method* overrides the found method, checks the return type and arguments, which must be compatible with the overridden method.

7. **Checking Variable Use:**

   Note that variable usages appear only in global functions and methods. We first check, whether the variable was declared in the function. Assume there is no such declaration. Then we check, whether there is a class wraping the function. If no such class, then this variable use appears in a global function without any declaration. If such class exists, we check the fields including inherited fields.

8. **Checking Field Access:**

   This is done by checking the followings:

   (a) Receiver must have a class type.
   (b) Class type must be a declared class.
   (c) Check the field in the class including inherited fields.

4.2 Translation Phase Algorithms

1. **Class Constructor:**

   In the **ClassTranslator**, the function **initConstructors** takes the list of classes and creates a constructor procedure for each of them. These procedures sets the class fields to their default values, since there is no NQJ language specification for constructor methods. So, a constructor procedure:

(a) allocates space for class structure on the heap and saves that space in a local variable obj_1,

(b) casts obj_1 to class pointer,

(c) sets fields of obj_1 to their default value, and returns obj_1.

2. **Translating Functions and Methods:**

   We first initialise all global functions except main. This means creating procedures from functions with empty blocks and adding to program. First initialising was chosen, since we may need to call a not yet translated function, while we are translating some another function. After the initialisation, not yet translated functions can be called. And then we translate the functions one by one (See *FunTranslator.translateFunction(f)*). Method translation is the same.

   Method initialisation is the same as functions, but we are adding one more parameter to method arguments, and that is the class pointer.

3. **Translating Method Calls:**

   We consider this procedure with following steps. The implementation is in *ExprRValue.case_MethodCall()*. We are not using virtual method tables for classes. They can be used to retrive needed method in runtime. However this compiler checks this statically and choses which method to call[2].

(a) Method receiver cannot be null. Adding null pointer check. This is now a runtime check.

(b) There are two classes to be examined: The method's and the receiver's. We cast the receiver's class to method's, if necessary.

(c) Calling the method with arguments by casting the argument types to parameter types if necessary. As the first argument we add the (casted) class of receiver.

4. **Translating Field Access:**
   See *ExprLValue.case_FieldAccess()*:

   (a) Adding runtime null pointer check for receiver.

   (b) Retriving the needed field from the class structure.

   (c) Returning the field pointer without loading or storing into a temporary variable.

5. **Field Inheritance:**
   We have already class structures with fields created. Now we will add new fields to these structures if

[2] During my project I hadn't got a serious knowledge of virtual tables. So I didn't use it in my implementation.

necessary. To this end, the field sequence will be maintained. For example, when a class X extends some class Y, the structure of X appends its fields to fields of Y. So the fields sequence will be: first fields of Y then of X. This enables us accessing field indexes of Y from X without changing the index.

The algorithm of *ClassTranslator.examineInheritance()*. This function takes the class node and its structure type.

(a) If class node has a super class, then function opens itself recursively with the super class node and its structure. Because super class may inherit other fields from its super classes.

(b) Getting the inherited fields.

(c) Appending the class structure to inherited fields. We implemented this with lists, so at the end we had a list with inherited fields and class fields.

(d) With method *setFields()* setting the new fields of the class structure.