Lecture: Compiler and Language-Processing Tools
Winter 2020/2021
Dr. Annette Bieniusa
FB Informatik, TU Kaiserslautern

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Portfolio Exam

January 11 - February 22, 2021

Erdem Yildirim

Eigenständigkeitserklärung

Mit Einreichen des Portfolios versichere ich, dass ich das von mir vorgelegte Portfolio selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe. Mir ist bekannt, dass Plagiate einen Täuschungsversuch darstellen, der dem Prüfungsausschuss gemeldet wird und im wiederholten Fall zum Ausschluss von dieser und anderen Prüfungen führen kann.

Declaration of Academic Honesty

By submitting the portfolio, I confirm that the submitted portfolio is my own work, that I have fully indicated the sources and tools used, and that I have identified and referenced the passages in the work - including tables and figures - that are taken from other works or the Internet in terms of wording or meaning. I am aware of the fact that plagiarism is an attempt to deceit which will be reported to the examination board and, if repeated, can result in exclusion from this and other examinations.

Documentation

This documentation is about the implementation of in terms of object-orientation extended NotQuiteJava (NQJ) compiler. The content of this documentation consists of two sections: Name and Type Analysis, Translation.

1 Name and Type Analysis

The key factor of object-orientation is the concept "class". Because of this, it is necessary to give information about the class representation first. By saying "class representation" one means: the modelling of in an NQJ program declared classes in such a way that, necessary abstractions for name and type analysis are provided.

Representig Declared Classes

During the implementation, necessary abstractions are identified by asking these three questions:

a. How could it be possible to indicate that, without expressing the class body, some element in an NQJ program is a class?
b. How can one represent the relations between classes, the subtyping?
c. How is it possible considering classes as a whole, with their fields and methods and with their relations to their super classes?

To answer these questions, this means implementing the aspects covered

by these question, one created three different models of classes. They serve as three different abstractions, each of which captures the essence of one of the above listed questions. These models are implemented as java classes. They can be found in package **ClassRepresentations**.

To reduce complexity, the term "class", when it is necessary in the context, will be from now on mentioned as either "java class", which indicates a class in the implementation, or "NQJ class", which indicates a declared class in an NQJ program.

Java Class "ClassRef": This model is an answer to question "a". It gives informations about the NQJ class, without expressing or referring to class variables or methods.

```
class ClassRef {
    String name;   // name of the NQJ class
    String extendsSomething; // name of the NQJ class' direct super class
    ClassType type;   // type of the NQJ class
    NQJClassDecl NQJdecl;  // AST node of the class

    ClassRef(String className, NQJClassDecl decl) {
        // Constructor explanation will be given in
        // the explanation of java class "ClassType".
    }

    /** Some methods for getting above declared variables. */
}
```

Java Class "ClassType": This model is an answer to question "b".

```
class ClassType extends Type {
    ClassRef thisClass; // indicates a declared NQJ class
    ClassRef extClass; // indicates its direct super class
}
```

The subtyping between classes is checked via two methods. In particular, the operator "==" for equality checking between **ClassType** was not used, since, during the analysis more than one **ClassType** object can be created for the same NQJ class. Instead, the java method **instanceof** was used:

```
boolean classTypeEqualTo(Type other) {
    if (other instanceof ClassType) {
        // checks the name equality
    }
    else return false;
}

boolean isSubtypeOf(Type other) {
    // returns true, if other type
    // is equal to thisClass's type
    // or extClass's type is subtype of other type
}
```

As seen above, the subtyping relations of classes are determined by checking class names. There are two refereces of **ClassRef**, so that one can implement the above given methods and determine the class type relations of NQJ classes.

Now we return to explanation of the constructor of **ClassRef**. This constructor has a nontrivial task between trivial ones, that is genereting the class type. For this reason, we explain only this task of the constructor using semi-pseudo code:

```
class ClassRef {
    // ...
    ClassType type;
    // ...

    ClassRef(String className, NQJClassDecl decl) {
        if (decl extends some NQJ class A) {

            // New ClassRef for extended class
            ClassRef extRef = new ClassRef(A.getName(), A.getClassDecl());

            // Determining the class type with this ClassRef and extRef
            this.type = new ClassType(this, extRef);
        }

        else {
           // Determining the class type with this ClassRef and "null"
           this.type = new ClassType(this, null);
        }
    }
```

```
}
```

Java Class "ClassContext": This model is an answer to question "c".

```
class ClassContext {
    ClassRef thisClass;  // Reference to class
    HashMap<String, VarRef> variables;   // Class variables
    HashMap<String, MethodScope> methods;  // Class methods

    /** Some methods for manuplating the hash maps. */
}

class VarRef {
    // Represents a variable with its type
}

class MethodScope {
    // Represents a method with its variable environment
}
```

Since the parsing phase provides strings for variables and methods of NQJ classes, one uses hash maps for mapping these strings to semantically checkable units, to **VarRef**s and **MethodScope**s.

At the end, we have three java classes for representing NQJ classes. The representation is separated into three classes for the sake of modularity.

Representing Types

Now we will describe, how types are represented during type checking. Since by lecturer provided type representation is practical, same represenation is used during type checking. The provided representation was only extended with **ClassType**.

```
Type =
   |INT |BOOL |INVALID |NULL |ANY
   |ArrayType {baseType}  // has a base element type
```

```
|ClassType              // type for classes
```

We have an abtract superclass *Type*. The types *INT*, *BOOL*, *INVALID*, *NULL*, *ANY* are represented by exactly one object, to allow comparison via "==", furthermore they are instantiated as static fields inside the abstract class *Type*, so that one can access them via

Type.

- INT
- BOOL
- INVALID
- NULL
- ANY

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

The type *INVALID* was never used in this implementation.

The expression checker uses the type *ANY* for an NQJ element, when the type of that element is unsound declared. By doing this, one secures the continuation of type analysis; leaving that element typeless will be error prone, since that element might be present in other NQJ program elements.

Analysis

The overall structure of name and type analysis is as follows:

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

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 with method **analysis.getTypeErrors()**.

Step by step explanation of analysis:

1. Name checking declared classes:
   a) Checking class names.
   b) Checking extension names, also checking cyclic dependencies.
   c) Creating a class table from consisten classes.
2. Name checking global functions.
3. Creating a name table from global functions.
4. Verifying the **main** method.
5. Analysing global function parameters and bodys.
6. Analysing classes:
   a) Checking variable names.
   b) Checking method names and method overwriding.
   c) Creating class context from variables and methods.
   d) Analysing method parameters and bodys.

We have a field **programAnalyser** of type **AnalyseProgram**, which is the main analysing factor.

```
// code of analysis.check()
programAnalyser.createClassTable();
createNameTable();
verifyMainMethod();
program.accept(programAnalyser);
```

Step 1 is implemented by **programAnalyser** in the function **createClassTable()**. This function creates a new **ClassTable** object. The constructor of **ClassTable** performs the tasks of step 1.

The class implementation uses two hash maps **helperTable** and **table**, from identifiers to AST nodes, and from identifiers to **ClassRef** objects. There are two maps, since for checking extension names one needs all names to be present. A map was implemented instead of a name list, so that gaining nodes from names would be easier.

```
// pseudeo code for Step 1
classes = AST class nodes
for class in classes do
    helperTable.put(class.name, class)
checkExtensions(classes)
checkCycles(classes)
for class in classes do
    table.put(class.name, new class ref)
```

Steps 2, 3 and 4 are implemented by **Analysis** in functions **createNameTable()** and **verifyMainMethod()**. **Steps 5 and 6** are implemented by visiting classes and functions with **programAnalyser**.

Different Variable Scopes

Function Scopes: There is a **MethodScope** class for functions and methods. This class implements a map from variable names to **VarRef** objects, and so represents the variable environment. The parameters are also included in this environment. Environment changing is implemented with a list of **MethodScope**s. When a new context is entered, if there exists already an opened scope, then this scope is copied and pushed to the top of the list. When that context ends, then the top **MethodScope** object is popped, thus turning back to old scope. If there are no existing scopes in the list, then one is created and pushed to the list.

Method Scopes: As an addition to function scopes, we have a class context. If we encounter an identifier in the method body, then we lookup that identifier first in method's scope, then if needed, we consider the class context. This is because method variables can shadow class variables.

Providing Information to Translation

During analysis phase, given AST nodes are decorated, so that:

- For each **FieldAccess** and **VarUse**, the variable declaration can be obtained by **getVariableDeclaration()**.
- For each **NQJClassDecl**, the extended class can be obtained by **getDirectSuperClass()**.
- For each **MethodCall** and **FunctionCall**, the function declaration can be obtained using **getFunctionDecleration()**.
- For each **NewObject**, the class declaration can be obtained using **getClassDeclaration()**.

Then, this decorated syntax tree is given to translation phase.

2 Translation

Overall structure of trasnlation:

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

Analysis takes the AST nodes and decorates them during type checking. Then, **Translator** takes the decorated syntax tree and translates it in the function **translate()**. Translating works as follows:

1. Initializing all classes.
2. Initializing fields for each class.
3. Initializing methods for each class.
4. Translating global functions, except main.

5. Translating classes and methods.
6. Translating main.

Translator has fields **funTr** and **classTr**.

```
// code of translator.translate()
classTr.initializeClasses();  // 1, 2, 3
funTr.translateFunctions(); // 4
classTr.translateClasses();   // 5
funTr.translateMainFunction(); // 6
return llvmProg;
```

There are two classes **FunTranslator** and **ClassTranslator**. With method **initializeClasses()** of class translator, struct types of all classes are created without fields and added to llvm program. Then, fields are added to each struct type and procedures are created with empty blocks, from class methods. Then we check the field inheritance.

```
// pseudo code for classTr.initializeClasses()
  initAllClasses();
  for class in classes do
      initFields(class);
      initMethods(class);
  for class in classes do
      examineInheritance(class); // for fields
```

Initializing Methods

For each method a procedure with empty block is created and as a first parameter, a class type pointer is given, so that one can obtain the current method's class with this first parameter and manuplate their fields. Afterwards the procedure is added to program procedures. For calling inherited methods, current class is casted to other class and then given to procedure of inherited method as first parameter.

Field Inheritance

Field inheritance examined after the initialisation of fields of all classes, since the implementation recursively checks over structures to get the field sequence precisely. When a class X extends some class Y, then the structure of X appends its fields to fields of Y. So the field sequence is: first fields of Y, then fields of X. This field sequence is chosen for implementing, so that one can access field indexes of Y from X without changing the index.

Translating Functions and Classes

There is a public class **CurrentStates**, in which the information about currently translated objects is captured. For example, current procedure or current class. Translation of functions made by **FunTranslator**. Classes and methods are translated in **ClassTranslator**. Function and method translation is as follows:

```
// with pseudo code
createInitializingBlock(); // initBlock
getProcedureOfFunction();  // proc
setCurrentProc(proc);
addBasicBlockToProc(initBlock);
setCurrentBlock(initBlock);
localVariableMap.clear();
storeFunctionParameters();
allocaSpaceForLocals();
translate(functionBody);
```

One first sets current procedure and block. Then we clear the variable map. There are two variable maps **localFunctionVars** and **localMethodVars**, from variable nodes to temporary variables, representing current function's or method's space allocated temporary variables. After that we allocate space for parameters and local variables, also adding them to map. Finally function body is translated with the visitor.

Class translations are made in **ClassTranslator** by only translating class methods, since structures and method procedures are already initial-

ized.

New Objects: There is a method **initNewObject(class, objVariable)** in **ClassTranslator**. It takes as parameters the class node and a temporary variable for storing the object. Then a constant structure was created with default field values. So, the class constructor was implemented in this method. Finally, a global is created for the object and added to llvm program.

Reflection

This compiler is my first programming project. Besides being proud of implementing a big structure like compiler in my early studies, I gained an insight view of a subject, which is now drawing my interest. Compiling is drawing my interest, because it is only be done, when one engineers the compiler. I used the word engineering because, first of all, compiler has to be designed incrementally like a building. One has to gain the insight of one step, before he/she proceeds to higher steps. Otherwise there will be a lack of insight knowledge, which leads someone to design some steps unconsciously. "Unconsciously designing" is the case, when code works but its designer doesn't know precisely how it works. So, knowing the essence of each step means your compiler lasts longer (in your mind), there will be no gaps in the building. During the portfolio I experienced that my implementations can pass the tests, even when I designed and implemented them unconsciously. I encountered with this situation at the very first stages of analysis and immediately deleted the unconscious designs. Because, if someone designs his compiler unconsciously, then he is not the engineer of it.

The other aspect of engineering is, drawing the problem in a paper to get an embracive view. A conscious design first needs to be drawn. Drawing is the most valuable helping assistant, because one cannot say something like "I lost so much time with drawing". Indeed, drawing can consume much more time than one assumes, but at the end you are giving that time to understand the problem. Drawing helps, because in the end it always supplys the knowledge one wanted to require. During the portfolio I noticed that time pressure can lead someone to unconsciously design a part of the compiler. Since engineering a compiler was my aim, in such situations I choosed ignoring time and giving precedence to understanding of the part I'm implementing. For example, after I have finished the analysis phase, there were 10 days of portfolio time for the translation phase. But I have

chosen taking time and drawing a preliminary work flow of translation, e.g describing in clearest manner how field inheritance should work and after which algorithms should field inheritance be situated, etc. Another example: I started coding the analysis phase, after I satisfied with my parser knowledge. Parsers consumed 3-4 days from my portfolio time. But I think the time is not lost, when I sketched many states of parsing in my notebook. At the end I understood how parsing works.

Lastly, engineering is not only about designing and implementing. It has a more embracive nature: considering other people. One has to be disciplined for giving insights into what one creates and what one knows. So an engineer has the culture of explaining. The most valuable thing I have learned with my portfolio is, how hard the explanation of ones own work, stating what it is to contain, can be. Since this is my first project and so my first explanation, it seemed to be hard, but I asked "why is it hard". Because explaining demands being rigorous and accurate. After I finished the whole code writing phase, I started to document it. In a short period I noticed that my code is not consistent with my sentences. For example, while explaining the analysis phase, I saw many useless function names and package names, or many algorithms are situated in wrong classes. So, for a healthy documentation I checked my code first to make it understandable for me and others. In this way, I learned that unreadable code can work well, but if one wants to make his/her code readable, then he/she must try documenting it (not only with javaDoc) or explaining it.

In conclusion, I am interested in compiler writing, since it has to be engineered. But so far this fact is not the most exciting aspect for me. During the lecture I found the opportunity to see, how problems are analysed to drawable units: We have a problem, for example, counting the words in a program. The first abstraction is clear, finding all of the words. But for the sake of computer, we have to arrange some rules. So we create regular expressions for capturing word rules. Finally creating an automaton, which takes word by word the written program and capable, specifying

each word without any error. Now, if somene asks "Can you draw that problem?", I will draw that automaton. This is like generating new abstractions from older ones, until one founds a concrete ground. In this respect, the lecture, I think, consists of to drawable units analysed problems. So, I don't want to point out some specific topic and say, this one is better than others. However, for me always that topic is the most considerable one, which does not provide conventions. Lexers and parsers, for example, provide conventions, when one considers programming laguages. What happens, when one considers a natural language? During the portfolio time I was reading the book "The Time Regulation Institute", one of the major works of modern turkish literature. It is characterized by long, elegant and brillantly ironic sentences, in which old maxims are combined with new turkish words and old turkish syntax with new syntax[1], and this was made in such a way that, this does not bother the reader. I asked me during the portfolio, how this brilliant syntax can be analysed. Considering or studying this kind of sytanx further would be my area, or at least I think that for now. The context-dependent analysis and the translation provides not many conventions I think. If I had to remake this portfolio a second time, I would like to change my analysis and translation implementations, so I can think of another implementation designs. Lastly, I would prefer adding new features to the NQJ language and implementing the compiler of this changed language, if I had to do the portfolio task a second time.

[1]Turkish language has a distinguishable old-new syntaxes and old-new words, since turks have made a reform in letters in 1928 and a reform in the language in 1932. To understand the impact: Within a year, 2 millions of people learned reading and writing (the population was 13 million people, 11 millions of them were living in primitive villages).