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

Ayush Pandey

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

1 Analysis of Object Oriented Constructs

1.1 Fields used for analysis

Field	Description
`NQJProgram prog;`	AST representation of the program
`List<TypeError> typeErrors`	List of type errors against the elements which lead to the errors
`NameTable nameTable;`	Table containing the type information used by the analysis phase
`boolean inheritanceCycle`	Used to check if the class hierarchy contains a cycle
`LinkedList<TypeContext> ctxt`	List of contexts, used as a stack. The last element is the current context.
`NQJClassDecl currentClass`	Declaration of the current class in the context.

1.2 Analysis steps.

The analysis phase performs the type checking of the components of NotQuiteJava program. The Type checking begins from the `Analysis` class which extends the `DefaultVisitor` class to visit the components of the program based on an abstract syntax tree using the visitor pattern [1]. The execution begins with the function `check` as shown in Listing 1.

```
public void check(){
   // Initialise a new name table
   // Verify the main method
   // Visit the children of the top level AST node (NQJProgram).
}
```

Listing 1: `check` method in `Analysis`

Function `check` initialises a new name table and verifies the main function by calling `verifyMainMethod` as shown in Listing 2. The visitor is then called on the top level declarations.

```
private void verifyMainMethod() {
// Check that main function is present
// Check that main returns int type
// Check that main does not accept any parameters
// Chack that main has a return statement as the last stament in the body
}
```

Listing 2: `verifyMainMethod` method in `Analysis`

1.3 Visiting Top Level Declarations

Top level declaration for an `NQJProgram` can be a list of global static function declarations or a list of class declarations. The visitor is called on the functions implicitly as explained in section 1.3.1, but for the class declaration list, we need to check for an inheritance cycle. This is done by the visitor function for `NQJClassDeclList` as shown in Listing 3

```
public void visit(NQJClassDeclList classDeclList) {
   boolean inheritanceCycle = checkInheritanceCycle();
   if (inheritanceCycle) {
      return;
   }
   super.visit(classDeclList);
}
```

Listing 3: class declaration list visitor in `Analysis`

The function `checkInheritanceCycle` calls a recursive function on the classes in the declaration list. The recursive function sets a `step` variable to 0 and then finds the parent class of the current class in the recursion step. At every recursive call, `step` increases by 1. If there is no parent class for the current class, then the recursion terminates and a cycle is not present. However, if the value of `step` is greater than the size of the `classDeclList` then the recursion is stuck in a loop and a cycle is present. The pseudo code for this is shown in Listing 4.

```
private boolean checkInheritanceCycle() {
   boolean foundCycle = false;
   // For every class in the class declaration, call the recursive check.
   for(classDeclaration in classDeclList)   {
      foundCycle = foundCycle || recursiveCheck(classDeclaration, 0);
      // We don't break the loop here so that we can detect all the possible cycles.
   }
}

private boolean recursiveCheck(classDecl, step){
   if step > classDeclList.size
      // Report presence of inheritance cycle.
      return true
   else
      extendedClass = classDecl.getExtended()
      return recursiveCheck(extendedClass, step+1)
}
```

Listing 4: Checking inheritance cycle in the class declaration list

After the check for inheritance cycles is complete, the visitor is called on the individual classes in the class declaration list. This is explained in section 1.3.2

1.3.1 Global static function

For a global static function, the active class context should be `null`. In such a case, a new context is started for the function body. Function declaration should not contain duplicate parameters and the return type is added to the context for checking against the return statement in the function body. The visitor is then called on the method body. The method body is a block and is visited by the `NQJBlock` visitor as explained in section 1.5.

1.3.2 Class Declaration

For a class declaration, we first access the parent classes recursively and fetch the fields from them. These fields are added to the extended scope of the current class. For the current class definition, we start a new context. Then perform a check for duplicate method declarations. If no duplicate methods are present, then the visitor is called on the class methods.

1.4 Visiting Class Methods

For a method declared as the member of a class, the method declaration is fetched from the class hierarchy. If a duplicate method is present in the class hierarchy, the number of parameters and the type of each parameter should match for the two methods. Along with this, the return type should follow the subtype relation. If either of these criteria is violated, a type error is reported indicating incorrect overriding of the method.

If the method is properly overridden, the parameters are checked for duplicates. If no duplicate is found, the visitor is called on the method body. The method body is a block and is visited by the `NQJBlock` visitor as described in section 1.5.

1.5 Visiting Blocks

After matching the top level constructs of the AST, we reach blocks of type `NQJBlock`. A block is a list of elements of type `NQJStatement`. For every new block, a scope is created. This is done to implement variable shadowing between nested blocks. The Block visitor then calls the visitor on the individual statements. For statements that declare a new variable, a check in the context is made. If a variable declaration already exists in the block context, a type error is reported. Otherwise, the statement is visited as described in section 1.6. For statements that access a variable, the type checker tries to find a definition by climbing the hierarchy of nested scopes. If a declaration is not found in the hierarchy, a type error is reported.

1.6 Visiting Block Statements

The `NQJStmtIf`, `NQJStmtWhile`, `NQJStmtReturn`, `NQJStmtAssign` contain Blocks and Expressions as components which may have references to members from other classes. These are matched by the `ExprChecker` class which implements the matchers for `NQJExpr` and `NQJExprL` as described in section 1.7. `Analysis` class methods access the `ExprChecker` matcher via `checkExpr` function. The current context and the expression which needs to be visited is passed as the parameters to the function. The function then creates a new instance of `ExprChecker` and calls it on the expression.

```
public Type checkExpr(LinkedList<TypeContext> ctxt, NQJExpr e) {
   return e.match(new ExprChecker(this, ctxt, nameTable));
}
```

Listing 5: checkExpr for matching components of statements

1.7 ExprChecker Class

The `ExprChecker` class implements the matchers for `NQJMethodCall`, `NQJExprThis`, `NQJNewObject`, `NQJFieldAccess` types. These are explained in the following sections.

1.7.1 NQJMethodCall

For a method call, the receiver is retrieved. If the receiver of the method call is not a of `ClassType`, then a type error is thrown since primitive types in Not Quite Java don't allow method calls.
If the receiver is a class and the called method is not present in the receiving class or the receiving class does not inherit that method, a type error is thrown.
If the name and type of the parameters supplied with the method call do not match the expected parameters, a type error is thrown.
If none of these cases hold true, the method call is type correct and is accepted.

1.7.2 NQJExprThis

`this` can only be referred from within a class context. If `currentClass` contains a class declaration, a `ClassType` referring to the current class is returned. Otherwise, a type error is thrown.

1.7.3 NQJNewObject

For an object declaration, if the class being instantiated exists, a `ClassType` is returned. Otherwise a type error is thrown.

1.7.4 NQJFieldAccess

For the `fieldAccess`, we get the receiving object. If the receiver is not a class, a type error is reported stating that primitive types do not have fields. If the receiver is a class but the field is not present in it, a type error is reported. Otherwise, the declaration type is returned from the class.

2 Class Type

`ClassType` is a type implementation for classes. The type is initialised with the declaration of the class which makes it easy to access the members from the type definition.

```
public ClassType(NQJClassDecl classDecl) {
   this.classDecl = classDecl;
}
```

Listing 6: Constructor for the `ClassType`

2.1 Subtype relation between classes

To check if a class is a subtype of another, i.e. A $\prec$ B, we check the following conditions.

- If A and B have the same class declaration, they adhere to the subtype relation.
- If the A is null type and B is a class, then the subtype relation is true.
- If A and B are of class type but the class declaration does not match, then we check in the class hierarchy and see if the child relation holds. If A is a child of B, then A is also a subtype of B.

3 Translation of Object Oriented Constructs

3.1 Fields used for Translation

Field	Description
`StmtTranslator stmtTranslator`	Translator class for `NQJStatement` type.
`ExprLValue exprLValue`	Translator class for `NQJExprL` type.
`ExprRValue exprLValue`	Translator class for `NQJExprR` type.
`Map<NQJFunctionDecl, Proc> functionImpl`	List of function declarations with their generated procedures of `Proc` type.
`NQJProgram javaProg;`	AST representation of the `NQJProgram`.
`Prog prog`	LLVM IR representation of the `NQJProgram`.
`Map<NQJVarDecl, TemporaryVar> localVarLocation`	Mapping of variable declaration of `NQJVarDecl` type to `TemporaryVar`
`Map<analysis.Type, Type> translatedType`	Mapping between the Analysis types and LLVM IR Types.
`Map<String, TypeStruct> classTypeStructs`	Mapping of the class name against the translates `TypeStruct`
`Map<String, Proc> classConstructorProcs`	Mapping of class name against the implicit constructor
`Map<String, TypeStruct> classMemberStructs`	Mapping of the class name against the `TypeStruct` containing its members.
`Map<String, Global> classGlobalStructs`	Mapping of class name against the `Global` declarations in the class.
`Proc currentProcedure`	The procedure being translated.
`BasicBlock currentBlock`	The current `BasicBlock` to which the instructions are added.
`BasicBlockList currentBasicBlockList`	List of current `BasicBlocks`.
`Map<NQJVarDecl, Integer> fieldAddressOffset`	Mapping of the variable declaration against its address offset in the declaration `TypeStruct`.
`Map<String, Integer> methodAddressOffset`	Mapping of a method declaration against its address offset in the declaration `TypeStruct`.

4 Translation Steps

4.1 Generating class structs

Translation of the program begins by generating `TypeStructs` for the classes. For this, the visitor visits every class in the `NQJTopLevelDeclList` and generates a Struct. If the class has already been handled, an entry will be present in `classTypeStructs`. If not, a new `TypeStruct` is created with the name of the class and an empty `StructFieldList`. This is referred to as `classTypeStruct` This struct is then added to the list of handled classes and to the list of the Struct types of the program.

A reference to the implicit constructor is created next. This is done by the `Proc` constructor of the AST class. The constructor is given a name and a pointer to the Type struct of the class. An empty list of parameters and a new Basic block list is also added to the constructor. These will be populated later by the further steps.

Next, a list containing the members of the class is generated and added to the translated program struct types. Another list, pointing to the global definitions of these members is added to the list of program globals. These will be used later when methods and fields are accessed between classes that inherit each other.

As the last step of struct generation, a reference to the constructor is added to the list of procedures of the program.

4.2 Generating the Implicit Constructor Body

For every class in the program, a new `BasicBlock` is created which serves as the block containing the constructor body. The constructor proc which was generated in step 4.1 is fetched from the struct. The return type of the constructor then becomes a pointer to the class struct which will contain the initialised fields.

Next, a reference to the members of the class is added to the constructor. This reference contains a pointer to the member struct generated in step 4.1.

After the members are added to the class, the fields are initialised with their default values. This is done by matching the field type with the corresponding `minillvm.ast.Ast` type in the `InitialisationTranslator` as described in section 5.

Last, a return instruction, returning the reference to the initialised class is added to the constructor body. Whenever the class is instantiated, the constructor returns the reference.

4.3 Generate Procedures for Class Methods:

For the methods defined in a class, we need to consider overriding. So, first the methods of the parent class are handled..

For a parent class, the fields are copied to the field struct that the method accesses. This allows the methods of the child classes to access global fields from the parent class. The `Global`

definitions from the class are also added to the current class' global struct. These structs are then used by the method translator to translate member methods.

Next, a reference to the class in which the method is defined is added to the parameters of the method. We call this list `parameterList`. This allows every method context to be able to access the class. The reference to the class is always added as the first parameter of the method.

Now, we handle the formal parameters of the method. For this, we translate the `NQJVarDecl` type into a valid LLVM type and add it to the `parameterList`. After the parameters are translated, the reference to the method declaration is added to the type struct of the class.

A `Proc` is created for the method which contains the return type of the method, the parameter list containing the reference to the class and an empty basic block list which will contain the method body.

After translating the method declaration, we check the overriding of the method. If the field struct of the class contains a method of the same name, the method is overridden. Since, the programs are expected to be type correct, we do not check if the overriding is valid again. The overridden method is removed from the list of global definitions of the class because at this point, two instances of the same method exist. One which was copied from the parent class and the other which is locally defined.

As the last step, the procedure is added to the global list, the class and the method procedure is added to the auxiliary maps used for translation. Along with this, the method address offset is added to the method offset map `methodAddressOffset`.

4.4 Translating Method Bodies:

For the method body, we load the the procedure generated in the last step corresponding to the method name and set it as the current procedure. This allows us to access the list of formal parameters. We create a `TemporaryVar` for every formal parameter of the procedure except the `this` reference. An `Alloca` instruction is added to allocate the space for the parameter and a reference is added to the procedure body. The temporary is also added to the current scope.

The variable `currentBasicBlockList` is initialised with the basic blocks of the procedure and the translator is then called on individual statements of the method. After the statements are translated, the scope is cleared.

4.5 Translating Statements containing Object Oriented constructs:

This is done using a matcher of type `NQJStatement`. It can match with one of the following cases:

- **Variable declaration:** For a variable declaration, the type of the variable is calculated and added to the `currentBasicBlock` which contains the body of the method being translated. The `InitialisationTranslator` 5 is then called to set the default value based on the type.
- **Assignment:** Assignment needs to consider subtyping relation as well as an access to the members of the classes. A separate translation matcher is called on the left and right sides of the assignment and a type cast is added if the types follow the subtyping relation.

- **Return:** In the return statement, the return type is typecast to the target return type of the current method in scope and a `ReturnExpr` is added to the block, returning the typecast result.

4.6 Translating the Left and side of an Assignment(ExprLValue):

The left hand side of an assignment can match with the following:

- **Field Access:** For a field access, the receiver is calculated using the `ExprRValue`4.7 translator. A pointer to the field is then added to the current basic block. The address offset of the field in the receiver is accessed from the `fieldAddressOffset` map. This gives a pointer to the field from the class in which it was originally declared.
- **Variable Use:** If a variable was declared in the current scope, a reference is returned. Otherwise, the variable is inherited from a parent class and a pointer to the field is returned. The address offset of the field in parent class is accessed from the `fieldAddressOffset` map.

4.7 Translating the Right and side of an Assignment(ExprRValue):

The right hand side of an assignment or a call from the `ExprLValue` can match with the following:

- **New Object Definition:** For a new object definition, we get a reference to the procedure of the implicit constructor for the class and add a `Call` to the constructor procedure. This call returns a reference to the class with properly initialised members.
- **This:** `this` is used to access the current class scope. We return reference to the first parameter of the method which contains the reference to the class.
- **Function Call:** A call to a global static function does not need access to class scopes so a reference to the procedure of the function is returned.
- **Method Call:** For a call made to a class method, We first load the member table of the receiving class. Then, the offset of the method is fetched from `methodAddressOffset` map. Using this offset, a `Load` instruction is added to the block to load the method at the pointer. Next, the return type of the method and the type of the parameters fetched from the method definition. `Bitcast` instructions are added for all such data types to allow subtypes. To call the method, a `Call` instruction is added to the block.

5 Initialisation Translator

For an object of a class, We need to initialise the fields in the struct type defined for the class with their type values. If the struct type contains a pointer, the fields from the target of that pointer are initialised too. An integer is initialised with `ConstInt(0)` and a boolean is initialised with `ConstBool(false)`. For a struct type, the initialiser is matched with the type of the field. If the fields are of primitive types, then the default values are stored. For user defined types like classes, the pointer to the class type gives the detail of the fields of the class. The matcher is called on the pointer target which can be a class or an array. This recursive matching is done

on the target until a primitive type is received for the components of the target, i.e. For an array, the elements match with the primitive type and for a class, the class fields match with a primitive type.

Reflection

- What was the most interesting thing that you learned while working on the portfolio? What aspects did you find interesting or surprising?

  After having implemented the concepts of object oriented programming, I can safely say that i understand more about them. The most interesting part about the implementation was the Visitor pattern. I had never encountered it before. After having read a bit about it, what surprised me the most was how class hierarchies can be employed for some very interesting applications. Another aspect which is very surprising is that object orientation in languages now appears to me as syntactic sugar. I do understand the ease that it gives for programming but i also know that it is nothing but pointer manipulation for the most part.

- Which part of the portfolio are you (most) proud of? Why?

  Personally, I find the TypeChecking part relatively better than the rest of the implementation. The reason is that i think I used the data structures well. I also made good use of the visitor pattern and also understand about types and subtypes. The reason i don't consider translation as the best part even though it has more a complicated implementation is because i still don't understand things in their entirety. There are a few things about LLVM which i definitely overlooked because of time constraints and need more in depth understanding.

- What adjustments to your design and implementation were necessary during the implementation phase? What would you change or do different if you had to do the portfolio task a second time?

  If i had to implement things a again, I would definitely change the number of data structures being used for both translation and type checking. I certainly would try to make them into a unified store so that everything can be organised better. I tried doing such things by making all the fields private and introducing getters and setters for them in the translation implementation but i am not absolutely happy about the way the code is organised. I did not have to make any major changes to my implementation plan. There were minor things like the use of `instanceof` which i used heavily in the beginning but as i understood the matchers and visitors better, i replaced `instanceof` and loops as much as i could for places where the visitor pattern could be used.

- From the lecture/course, what topic did excite you most? Why? What would you like to learn more about and why?

  From the lecture, the most interesting part was definitely the SSA and the optimisations based on SSA. I had always learned about the problems in data structures and what the optimal ways of solving them could be. I never found a way to use them at the level

that i learned them. But with things like the use of Graph Colouring to find out spills and minimise the use of temporaries and registers, I was fascinated to finally see an implementation where there was a direct mapping between 2 problems. It also gave me a sense of how solving a single problem in a complexity class could lead to multiple problems being solved. Another aspect of the lecture that i found very interesting was the parser generators. It was hinted multiple times that I would have to use it in the future and I did. I had to implement a template engine where i worked with leex and yecc and used the concepts of grammars and generators.

- Things that i would like to learn more about in the domain of formal languages and their processing.

  I would definitely learn more about intermediate languages. I believe that we have barely scratched the surface of how LLVM is used and there is more to learn. I would also look more into how the different paradigms of programming translate to low level languages. We tackled Object oriented languages but there are functional languages, Languages based on pure Lambda calculus which should have interesting ways of handling things. I am also now going to look into depth into TypeScript which i use almost everyday.

- Which resources did i find the most helpful throughout the course and which resources did i use most heavily?

  During the initial phase of the lecture, I heavily used the lecture script as well as the book on compilers by Aho et. al. [2]. After we moved from the parsing to the intermediate representation and Abstract syntax trees as well as during the implementation of my portfolio exam, i referred a lot to the book on compiler implementation by Andrew W. Appel [3]. It gave me some interesting ideas towards implementation of struct types for classes and how scopes can be managed for class hierarchies. Another course that i used for some ideas is Compilers(SOE-YCSCS1) by Stanford university [4].

References

[1] D. Szczukocki, "Visitor design pattern in java," Sep 2019. [Online]. Available: https://www.baeldung.com/java-visitor-pattern

[2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, ser. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986. [Online]. Available: https://www.worldcat.org/oclc/12285707

[3] A. W. Appel, *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

[4] A. Aiken, "Compilers(SOE-YCSCS1)." [Online]. Available: https://online.stanford.edu/courses/soe-ycscs1-compilers

[5] L. Project, "LLVM language reference manual," Feb 2021. [Online]. Available: https://llvm.org/docs/LangRef.html