Compiler Construction

Type Checker - Tutorial

Christopher Liebmann, Sebastian Puck
Practical Instructor: Alexander Perko

cc@ist.tugraz.at

Institute of Software Technology
Graz University of Technology
Austria

Summer Term 2024

Version: April 11, 2024

Outline

Task 2 - Overview

Coming from Task 1...

```
MyClass : MyClass2                           // class decl ✓
{
  int myField;                               // field decl ✓

  int myMethod(int param1, bool param2)      // method head ✓
  {
    ...                                      // TODO: method body
  }
}
```

But first...

- Make sure you fix errors from Task 1 (if you had any)
- Concentrate on positive cases, i.e., tests which conform to Jova-spec, but are falsely rejected by your implementation
- If you need help:
  $\Rightarrow$ Come to voluntary review meeting (date: TBA)
  $\Rightarrow$ Use the Task 1 test system

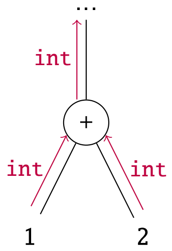
Task 2 - Type Checking

- Jova is statically typed
- We have to guarantee type safety at compile time...
- ... via verifying types of operations/expressions conform to (informally described) type system/rules

Task 2 - Implementation

Implementation - Type Checker

- Walking the method body subtree(s)
- Via visitor/listener implementation
- Utilize ANTLR4 labels to get more direct access to information in visitor/listener
- Synthesize types from leaves to root
- Check at every node (operation) whether types conform to type rules
- If not $\Rightarrow$ report error



Symbol Table - Implementation

- Extend symbol table
- Add new layer for local/method scope to support variable shadowing
- Create mapping from local variable names to types
- In the mehtod body you will need to access:
  - Class types/IDs (incl. super classes)
  - Members (fields/methods)
  - Method parameters
  - Local variables
  - Built-in functions

Implementation - Symbol Table

- In addition: Think about symbol table design with respect to Task 3
- You will need to map local variables to (JVM specific) *local variable array* indices for every method translation
- Most JVM instructions expect a specific type
  E.g., `iload 1` loads integer value from local array index 1
- You will also need access to the class name of members to generate correct instructions, e.g.:
  ```
  getfield MyClass/myField I
  invokevirtual MyClass/myMethod(I)I
  ```

References/Resources

- Wotawa, Franz. Compiler Construction Lecture Slides. TUGraz, 2024.
- Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. Compilers, Principles, Techniques. Boston: Addison wesley, 1986.
- JVM structure: `https://docs.oracle.com/javase/specs/jvms/se21/html/jvms-2.html`
- JVM instruction set: `https://docs.oracle.com/javase/specs/jvms/se21/html/jvms-6.html`

Happy Hacking!