CS444 Assignment 2 - 4 Report: Static Checking

Amrit Sehgal - a38sehga
Aryaman Dhingra - a23dhing
Jagvir Dehal - jdehal
Paranjay Agarwal - p38agarw

Building the compiler

Our compiler can be built with `make`, using either `make build` or just `make`. The system it's built on needs the following dependencies, all of which are available in the student linux environment: g++ 11.4.0, GNU Make 4.3, cmake 3.22.1, flex 2.6.4, and bison (GNU Bison) 3.8.2. After building, the executable `joosc` will be placed in the project root directory.

Design and Implementation

Our compiler is written in C++. The important components of our program that were added or modified in assignments 2-4 are detailed below.

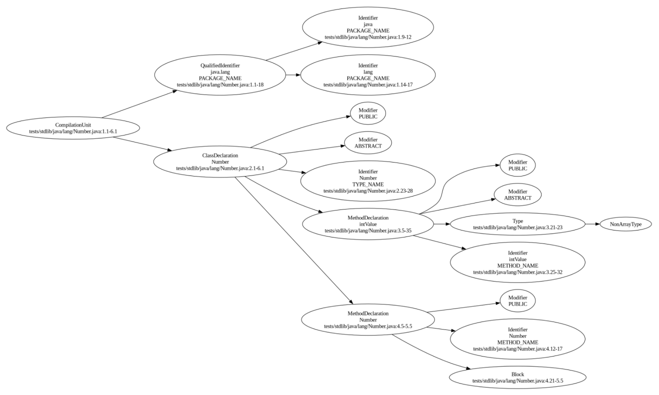
Build System

Due to the ever-increasing complexity of our compiler, we made the following changes to our build system from assignment 1.

By the time we got to assignment 3, we had very long compile-times. *Ninja* is a small build system focused on speed, and is used in conjunction with CMake to speed up the total compile time of the project (1m20s to 20s in a slow Docker container). Since the student environment does not have Ninja, the Makefile has a mechanism to detect and fallback to GNU Make if need-be.

`make_env.sh` is a utility with a plethora of commands to help with debugging, running and viewing test cases and has been critical in finding and resolving bugs quickly.

`rungraph <testname>` is a new command that was introduced to visualize our ASTs. The graphs are generated by visiting the nodes in the ASTs and outputting a DOT language file which converts our ASTs to viewable images. It optionally contains information about the nodes and their layout, and helps debug issues in AST generation as well as the rest of the compiler.


CompilationUnit
tests/stdlib/java/lang/Number.java:1.1-6.1
QualifiedIdentifier
java.lang
PACKAGE_NAME
tests/stdlib/java/lang/Number.java:1.1-18
Identifier
java
PACKAGE_NAME
tests/stdlib/java/lang/Number.java:1.9-12
Identifier
lang
PACKAGE_NAME
tests/stdlib/java/lang/Number.java:1.14-17
ClassDeclaration
Number
tests/stdlib/java/lang/Number.java:2.1-6.1
Modifier
PUBLIC
Modifier
ABSTRACT
Identifier
Number
TYPE_NAME
tests/stdlib/java/lang/Number.java:2.23-28
MethodDeclaration
intValue
tests/stdlib/java/lang/Number.java:3.5-35
Modifier
PUBLIC
Modifier
ABSTRACT
Type
tests/stdlib/java/lang/Number.java:3.21-23
NonArrayType
Identifier
intValue
METHOD_NAME
tests/stdlib/java/lang/Number.java:3.25-32
MethodDeclaration
Number
tests/stdlib/java/lang/Number.java:4.5-5.5
Modifier
PUBLIC
Identifier
Number
METHOD_NAME
tests/stdlib/java/lang/Number.java:4.12-17
Block
tests/stdlib/java/lang/Number.java:4.21-5.5

-

Abstract Syntax Tree
In Assignment 1, our parser generator outputted a generic, non-type safe parse tree which was sufficient for checking the first assignment's weeding conditions. For future compiler stages though, we needed a more robust intermediate representation of the input program.

There were two main design options we considered for our AST as discussed in class. The first was a traditional inheritance-based AST, where the object-oriented visitor pattern would be used to make adding operations easy and adding node types hard. The second was using C++'s version of sum types, `std::variant`, and accomplishing the same visitor strategy with `std::visit`.

`std::variant` allows programmers to define types that can contain one of a set, as a more safe successor to unions from C. Calling `std::visit` on a variant object with a callable object that has an overload for each contained type automatically invokes the correct overload.

For example: with an inheritance design, Expression would be represented as an abstract class each sub-expression class inherits, whereas with variant design, it would be represented as a variant with the member types being sub-expression classes, i.e. `using Expression = std::variant<InfixExpression, PrefixExpression, CastExpression, …>`.

After some research and testing, we found that variants had the advantage of decoupling our visitor code from the ast class definitions, as `std::visit` didn't require an accept method and the callable class could have any return type on its operations. It also had a noticeable advantage of allowing less indirection, as polymorphism must be done through pointers in C++ and variants are stack-allocated. Inheritance had the advantage of cleaner nesting and allowing shared behaviour, for example each sub-expression inheriting a `type` field from `Expression`.

Weighing these tradeoffs, we went with variants. We decided more flexible operations were more important than more flexible classes, and we could alleviate the lack-of-common-fields drawback by using inheritance in a small amount, defining an `AstNodeCommon` class that each ast class inherits only so we can have shared fields without code duplication.

In fact, using this, we took advantage of bison's built in yy::location module to augment our AST with location information of the node's origins, as seen in the graph image shown previously. This gives us more detailed warning/error messages throughout the compiler and allows us to debug issues much faster.

Visitor Design
As described above, we knew one of the most important things going forward was having a good system for performing operations on an entire AST, as that was going to happen a lot. We wanted to do this in a way to eliminate as much boilerplate as possible.

We defined a class, `AstVisitor`, which had an `operator()` overload for every possible AST node variant member type. Each of those methods were abstract. This class also had an implemented `visit_children` method for every node type that would call `operator()` on the children ast nodes of that node.

Next, we defined `DefaultSkipVisitor` as a subclass of `AstVisitor` that overrides each `operator()` call to call `visit_children` on that node, but keeps the method overridable. The idea was to have the ability to simply continue traversing to child nodes without doing any operation on the current node as the default behaviour in our visitors, to reduce boilerplate.

Going forward, to make visitor classes, all we had to do was subclass `DefaultSkipVisitor`, define operations on the node types that particular visitor cares about, and call `visit_children` within those in whatever order we prefer (with the flexibility for preorder, inorder, or postorder traversals as appropriate), and ignore writing code for all other node types. This gave us a clean, reliable system for handling the next stages, detailed below.

Weeder Changes
As already mentioned in the Abstract Syntax Tree section, we rewrote our parser to generate ASTs directly from the grammar. This meant we would have to rewrite the weeder as well since our A1 weeder acted on our more simplified parse tree structure. So, instead of acting on our generic ASTNode pointers, the weeder was made type-safe and instead acted on more specific nodes, such as `InterfaceDeclaration`, `ClassDeclaration`, `MethodDeclaration`, `MethodInvocation`, `FieldDeclaration`, and `Literal`. The weeder's logic however remained untouched, and it still traversed these nodes, but in a type-safe manner. One future change could be to rewrite the weeder as a proper AST visitor, such as our other AST traversals, for consistency with the rest of our compiler.

Environment Building
Our environment building stage was handled in visitor class `EnvironmentBuilder`, the first of many that inherit from `DefaultSkipVisitor`.

An important part of environment building is the symbol tables. `SymbolTable` wraps a map from strings to lists of `SymbolTableEntry`, which is a variant of declaration object classes such as `ClassDeclarationObject`, `FieldDeclarationObject`, `MethodDeclarationObject`, etc. As with in lectures, these classes are distinct from AST classes and contain pointers to the AST node they were obtained from among other useful information. In particular, some have their own symbol tables, such as `MethodDeclarationObject` having a `parameters` symbol table exclusively containing mappings to `FormalParameterDeclarationObject`.

A design consideration we made was how much information to encode directly in these classes that could be obtained from the AST node anyway. In the end, we decided to be as minimal as possible with the fields in the symbol table entries to avoid duplication of state, and generally accessed the AST pointer, for example, to check what modifiers are associated with a `MethodDeclarationObject`. This was less verbose with helper methods on the AST nodes.

The environment building visitor's job, then, was to fill in the symbol tables. It was given an empty `PackageDeclarationObject` (representing the default package) and added the package declaration for each compilation unit to the package symbol table in that object, sometimes recursively if a package has subpackages. It kept track of the current package while traversing, and when visiting a `ClassDeclaration` AST node, added an object for that class to the current package's symbol table and set the current class to that object. There was also a current method for variables and formals which operated similarly. A two-way link between the AST node and the symbol table entry was always created.

Type Linking
Similar to environment building, the majority of work for type linking is done in a visitor class, this time called `TypeLinker`.

This visitor linked non-syntactically ambiguous uses of named types to their declaration symbol table entry objects. To make this robust, we added a `LinkedType` class that represented the compiler's representation of types. This class had a `NonArrayLinkedType` which could be a `PrimitiveType`, or `ClassDeclarationObject`/`InterfaceDeclarationObject` symbol table entry, as a pointer. It also had a boolean representing whether or not that `LinkedType` was representing an array of the type stored in the `NonArrayLinkedType`. This design lets us implement operations to compare type equality and perform queries such as checking if types are subtypes later.

In the visitor, the import and package statements were processed and used to create a `CompilationUnitNamespace` object, which was a helpful abstraction for looking up observable names in a compilation unit. This class contained the logic for resolving qualified and simple type names. After initially putting that logic in `TypeLinker`, we pulled it in a separate class so we could reuse the strategy for looking up type names in later phases after type linking had already done the job of resolving imports and package declarations once. The `CompilationUnit` AST node received ownership of this object for this purpose.

The visitor then simply uses the `CompilationUnitNamespace` to set each linked type.

Hierarchy Checking
The hierarchy checking phase of our compiler is also done using a visitor class.

The hierarchy checker is done in such a way that we maintain access to the inherited methods, overridden/hidden methods, as well as a list of overloaded methods for each identifier. This helps us in later stages of the compiler as we can easily connect method invocations from within each class to their corresponding declarations.

The overriding/hiding was done by getting a complete list of potentially inheritable methods using a DFS on a class's superclass and superinterfaces. Using this list, the classes were appropriately deemed inherited/overloaded, or overridden/hidden, and was respectively added to a list of overloaded methods, or removed from the overall list.

There was careful consideration with java.lang.Object in the special case when a class or interface has no direct superclass or superinterface respectively.

Disambiguation
Our disambiguation phase is implemented as another AST visitor traversal. This is the phase that classifies previously ambiguous names. All names in our compiler boil down to an `Identifier` node, so this node is where we placed the classification for each name. Our classification enum has four possibilities (`TYPE_NAME`, `PACKAGE_NAME`, `METHOD_NAME`, `EXPRESSION_NAME`).

We developed a main disambiguate function, which worked recursively. It implemented the steps defined in JLS 6.5.2. This meant that for each qualified name, the left-most name had to be classified first, and then based on the first classification, the next name could get classified, and so on for the rest of the names in the fully qualified name. This describes the basic recursive manner in which our disambiguation function worked.

The disambiguation step then involved visiting all `QualifiedIdentifiers` and disambiguating them based on the rules in JLS 6.5.2. We also had some special cases, such as single-type imports and on-demand imports, method invocations, and more, where we already knew the classification so we manually set those classifications instead of relying on the steps in JLS 6.5.2.

After our disambiguation step, our goal was to have every single name in the program to be classified. So, to enforce this, we wrote a visitor that checks that each Identifier in the program is not unclassified, and in the case that it is, throws an instance of `CompilerError`. This helped us catch bugs in our implementation since there shouldn't ever be ambiguous names after this step.

Type Checking
As expected, type checking is done in a visitor class.

Each subexpression AST node is given a `link` field, and the visitor operations assign them the correct `LinkedType` object based on the `link` values of the children nodes and generally by a straight following of the JLS specification. Using the work done in disambiguation, any

`QualifiedIdentifier` that was labeled as an `ExpressionName` is searched as a field and is given the type of the field that was assigned in type linking, while also checking access rules.

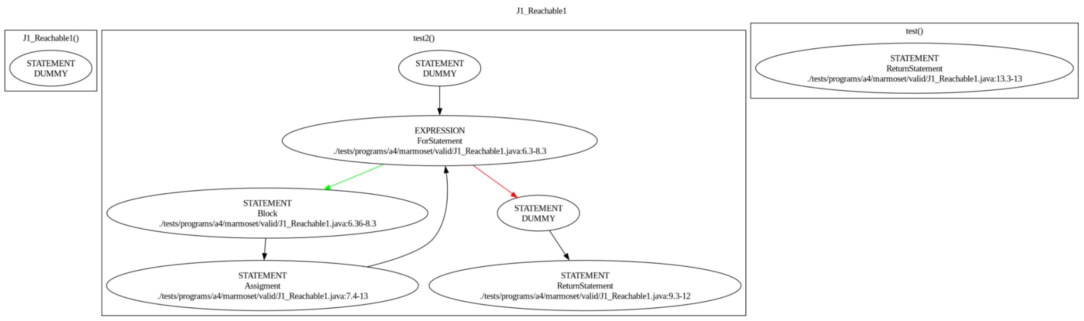
MethodInvocation calls also follow the simplified `joosc` method overload rules and applicable JLS specification, and compile-time access rules are checked within this visitor.

An interesting note about this visitor is that since the `link` of a parent node depends on the `link`s of the children, this is the first visitor where `visit_children` is called at the start of the visit logic instead of at the end, in a postorder traversal. This highlights the usefulness of our flexible visitor design that allows us to call or not call `visit_children` at any point.

Control Flow Graph Building
Control flow graph building was yet another visitor operation.

We visited each statement within a method declaration and recursively generated a sub-CFG depending on the statement seen. Conditional statements (eg. ifthen, while) were created from CfgExpression nodes, which had true/false branches, while the remaining statements were created from CfgStatement nodes, which simply had a next field pointing towards the next statement in the control flow. The CFG for Block statements (as well as the overall Block within the MethodDeclaration) were created by linking all of the sub-CFGs in order, and creating a larger overall CFG.

We also created a graph generator for the CFGs, again to help debugging and identifying errors.


J1_Reachable1
J1_Reachable1()
STATEMENT
DUMMY
test2()
STATEMENT
DUMMY
EXPRESSION
ForStatement
./tests/programs/a4/marmoset/valid/J1_Reachable1.java:6.3-8.3
STATEMENT
Block
./tests/programs/a4/marmoset/valid/J1_Reachable1.java:6.36-8.3
STATEMENT
DUMMY
STATEMENT
Assigment
./tests/programs/a4/marmoset/valid/J1_Reachable1.java:7.4-13
STATEMENT
ReturnStatement
./tests/programs/a4/marmoset/valid/J1_Reachable1.java:9.3-12
test()
STATEMENT
ReturnStatement
./tests/programs/a4/marmoset/valid/J1_Reachable1.java:13.3-13

-

Reachability Analysis
In reachability analysis, we visit the control flow graphs built as described above.

Essentially, we created a visitor for CFGs which would traverse the nodes, but only visit children which were reachable, and mark them so. Afterwards, we would look at the statements, and check if they were reached - if not, throw an error.

In the case of a For/While loop, we would evaluate whether the expression is a constant expression, and if so evaluate it statically. If the expression is then a bool, only one branch of the expression will be traversed depending on the value of the expression.

In the case of a ReturnStatement, we would cut off traversal so any statements proceeding it will not be marked as reachable.

Finally, if we reach a leaf node of the Cfg and we have a non-void return type, we simply throw an error if there is no return statement.

Testing and Known Issues

The primary way we have been testing our compiler is with integration tests, sample programs that are expected to be valid or invalid. We originally set out to use unit tests but found this unsustainable due to the sheer amount of code and the complexity of unit testing compiler stages which are very dependent on earlier ones. Instead, we have focused on maximizing the features we test in our sample programs.

One known issue is our dead assignments visitor. Currently, our visitor only detects one of the dead assignment test cases - the simplest one. This is because we don't properly handle variables going out of scope, and the set of unused assignments when these variables go out of scope. We are still working on ironing out the detection of dead assignments, and hopefully, we are able to properly implement the worklist or iterative algorithm to help us do this.

Work Split

Amrit

- Researched C++ AST design and potential of std::variant
- Worked on types/names/packages portion of AST class definitions
- Set up AstVisitor and DefaultSkipVisitor classes which are inherited by all other visitors and serve as the framework for visitor operations
- Implemented environment building stage
- Worked on SymbolTable and symbol table entry classes that represent all the declarations in the program
- Created scope manager for managing nested scopes in a method and available declarations
- Worked on type linking with Aryaman, and implemented the `CompilationUnitNamespace` class, to allow names to be searched with regards to the rules for checking imports in later compiler stages.
- Implemented type checking for qualified identifier's representing expression names, method invocation calls, and field accesses

Aryaman

- Rewrote the AST weeder after the structure of AST changed
- Defined some of the classes for our AST structure, mainly expression classes
- Worked with Amrit to implement the type-linking phase to properly link named usages of a type to the type declaration
- Pair programmed with Jagvir to implement disambiguation of names.
- Implement a forward declaration traversal phase to catch forward declaration of fields and methods within type declarations

- Pair programmed with Jagvir to build control flow graphs, as well as reachable statement analysis.
- Built a local variable visitor to ensure variables are initialized when declared, as well as enforcing that variables do not refer to themselves in the initializer.
- Improved our test script to give better feedback on success and failure with an assignment breakdown

Jagvir

- Rewrote parts of the scanner to handle comments, string and char literals better (more accurate location information)
- Rewrote the parser to directly generate ASTs rather than parse trees
- Created the graph generators for both the overall ASTs as well as the CFG
- Created the make_env.sh file containing helper functions to run, debug, and view tests
- Added autocomplete to make_env.sh commands
- Changed build system to Ninja to improve build speed
- Worked on handling inheritance, overriding, hiding and overloads within hierarchy checking
- Pair programmed with Aryaman for disambiguation, control flow graph building, and reachability
- Created a function to check for and evaluate constant expressions (utilized in the reachability of for/while loops)
- Revamped the parser to include the location information of each node to help debugging

Paranjay

- Updated scanner to correctly interpret comments and unicode characters
- Defined the classes for the AST, namely the statement classes
- Worked on handling extends, implements, and potential cyclic dependencies, using DFS, for classes and interfaces
- Implemented type checking while resolving expressions to have the appropriate type based on its children for most of the expression classes, including cast expression, instanceof, infix, and assignment expressions
- Wrote a function that handled the castability check for both cast expression and instanceof expressions

Thoughts

Compared to Assignment 1, Assignments 2-4 is when we really hit the ground running, and had to face the consequences of the advantages and disadvantages of our chosen language.

The lack of memory safety in C++ caused a bit of annoyance, as we had a few Segmentation Faults occur in each assignment which took time to step through and debug. Overall, we got better at solving these over time though.

The lack of polymorphism with `std::variant` caused some annoyance too. If a type was a variant of two types which both had a field of the same name and type, it still needed to be destructured to access the field, which created additional boilerplate in some cases.