James Madison University

JMU Scholarly Commons

Senior Honors Projects, 2020-current

Honors College

5-9-2020

Less-Java, more type safety: Type inference and static analysis in Less-Java

Charles Hines

Follow this and additional works at: https://commons.lib.jmu.edu/honors202029

Part of the Other Computer Engineering Commons

Recommended Citation

Hines, Charles, "Less-Java, more type safety: Type inference and static analysis in Less-Java" (2020). *Senior Honors Projects, 2020-current*. 27.
https://commons.lib.jmu.edu/honors202029/27

This Thesis is brought to you for free and open access by the Honors College at JMU Scholarly Commons. It has been accepted for inclusion in Senior Honors Projects, 2020-current by an authorized administrator of JMU Scholarly Commons. For more information, please contact dc_admin@jmu.edu.

Less-Java, More Type Safety: Type Inference and Static Analysis in Less-Java

---

An Honors College Project Presented to

the Faculty of the Undergraduate

College of Integrated Science and Engineering

James Madison University

---

by Charles David Hines

April 2020

Accepted by the faculty of the Department of Computer Science, James Madison University, in partial fulfillment of the requirements for the Honors College.

FACULTY COMMITTEE:

____
Project Advisor: Michael O. Lam, Ph.D.
Assistant Professor, Computer Science

____
Reader: Christopher J. Fox, Ph.D.
Professor, Computer Science

____
Reader: David H. Bernstein, Ph.D.
Professor, Computer Science

HONORS COLLEGE APPROVAL:

____
Bradley R. Newcomer, Ph.D.,
Dean, Honors College

Contents

Acknowledgements **4**

Abstract **5**

1 Introduction **6**

2 Related Work **8**

3 Objectives **10**

3.1 Type Rules for Less-Java 10

3.2 Unit Tests 10

3.3 Static Analysis 10

3.4 Object-Oriented Type Inference 11

3.5 Constructor Generation 12

3.6 Function Instantiation 13

4 Results **16**

4.1 Type Rules for Less-Java 16

4.2 Unit Tests 17

4.3 Static Analysis 18

4.4 Object-Oriented Type Inference 18

4.5 Constructor Generation 19

4.6 Function Instantiation 20

5 Future Work **22**

6 Conclusion **23**

A Less-Java Type Rules **24**

A.1 Expressions 24

A.2 Statements 24

List of Figures

1 A basic hello world program in Java 6
2 A basic hello world program in Less-Java 6
3 The expression tree for `x+1.5` where `x` is known to be of type `Double` 8
4 An invalid Less-Java program 11
5 Output of compiling figure 4 before the project 11
6 A valid Less-Java program that failed to compile at the start of the project 12
7 Java output of running the previous version of the compiler on Figure 6 14
8 A Less-Java program demonstrating multiple parameter bindings 15
9 An incorrect AST for a function with multiple bindings 15
10 The TInt type rule 16
11 The TIIAdd type rule 16
12 The TIf type rule 16
13 Invalid (left) and valid (right) uses of a `break` statement tested as part of unit tests 17
14 Invalid (left) and valid (right) `if` conditions tested as part of unit tests 17
15 Invalid (left) and valid (right) functions 18
16 Output from compiling figure 4 18
17 Less-Java code demonstrating use of a superclass's constructor 20
18 The correct AST for a function with multiple bindings as in Figure 8 21

Acknowledgements

Thank you to Dr. Lam for guiding the author through work on this project. Without your assistance this project would not have turned out the same way. Additional thanks to the readers for the project, Dr. Fox and Dr. Bernstein, who helped look over project work and ensure that adequate progress was being made throughout the project. A thank you to Zamua Nasrawt, the original designer of the Less-Java language and developer of the Less-Java compiler, as well as the rest of the Less-Java research team at JMU. Your help on this project was invaluable.

Abstract

Less-Java is an object-oriented programming language whose primary goal is to help new programmers learn programming. Some of the features of Less-Java that might make it better for beginners are static typing, implicit typing, low verbosity, and built-in support for unit testing. The primary focus of this project is on improving type inference (especially with regards to object-oriented programming) and adding static analysis in the Less-Java compiler.

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

Figure 1: A basic hello world program in Java

```
main() {
    println("Hello, world!")
}
```

Figure 2: A basic hello world program in Less-Java

1 Introduction

Less-Java is a programming language originally designed by Zamua Nasrawt [1, 2]. The goal of Less-Java is to be easier for new programmers to learn than Java. There are several core ideas behind Less-Java that aim to help accomplish this goal, each of them inspired by shortcomings of other popular introductory languages. The core ideas behind Less-Java are minimal verbosity, built-in support for unit testing, and a strong and implicit typing system.

Minimal verbosity makes Less-Java more beginner-friendly because it doesn't force the programmer to confront more advanced concepts that would not be important in an introductory class. Brushing against these advanced concepts for the sake of explaining the syntax could confuse the programmer, and ignoring them and instructing the students to blindly write these "magic" words is not a good teaching practice. Consider the basic hello world example in Java (Figure 1). For a new programmer to understand this code completely, they would need to understand concepts such as classes, functions, static functions, and function visibility, when all they need to do is write a message to the screen. The equivalent program in Less-Java (Figure 2) only requires the programmer to understand functions and output. Other popular introductory languages like Python and JavaScript address this complexity, but they have their own issues discussed later.

Less-Java also includes support for built-in unit testing. Introductory programming courses often teach that writing a good set of unit tests is valuable for evaluating the correctness of a program, yet the languages they use do not ship with native support for unit testing. Again take Java as an example. If a programmer wants to test their program, they need to either test in `main` or use a third party library like JUnit. Testing in `main` is not ideal because this mixes testing logic with application logic, and integrating external libraries can be complicated for new programmers. This hindrance can discourage new programmers from writing unit tests for their code, which makes the practice more difficult to pick up later in their career. In Less-Java,

the programmer can write simple unit tests using the form `test [conditional]`. These tests can be run separately from the program's `main` function and do not require the programmer to use external libraries, thus making the testing process easier for the programmer. This way the programmer is encouraged early on to write unit tests for their code.

Another feature of Less-Java that is intended to make it more beginner friendly than other popular introductory languages is a strong and implicit type system. Implicit typing makes the language more user-friendly because the programmer does not need to manually declare the data types for the variables in their programs. Static typing makes the language more beginner-friendly than other implicitly typed languages like Python and JavaScript because it allows the compiler to catch type errors before execution.

For the Less-Java compiler to enforce strong typing, it needs to determine the types of variables and expressions. Because the language is implicitly typed, this requires type inference. The compiler for Less-Java developed by Nasrawt included type inference that worked for primitive data types, but was insufficient for object-oriented programming. Additionally, the compiler did not check the program for type errors. Any type errors present in the Less-Java program would be carried forward into the Java code generated by the compiler. When this Java code was then compiled, the type errors would be caught and the programmer would be informed of issues in code that they didn't write. This behavior was not beginner-friendly, as understanding the error messages would require the programmer to know Java and have an understanding of which parts of the generated Java program correspond to which parts of the Less-Java program.

This work contributes full object-oriented type inference and static type checking to the Less-Java compiler, along with associated bug fixes and formal specifications.

2 Related Work

The Hindley-Milner type inference algorithm [3] is a common basis for type inference in programming languages, and can be used to infer types in object-oriented languages where programs are contained in a single compilation unit [4]. This algorithm first assigns each expression to be a variable, or unknown, type. Expression trees are then traversed to infer the type of expressions. This traversal begins at the leaves and works up to the root of the tree. The leaves are either variable references or literals. The type of a variable reference is set to the type of the variable in the current environment and the type of a literal is simply the type of the value it represents. Moving from child vertices to parent vertices, the type of the parent vertex is inferred based upon inference rules and the types of the child vertices. An example of such a rule would be that an addition of a `Double` and a `Double` is of type `Double`. Applying this inference rule, if a vertex representing an addition has exactly two children, both of type `Double`, then the vertex is assigned the type `Double`. This is shown in figure 3 where the two children of the addition node are of type `Double`, causing the addition node to be of type `Double` as well.

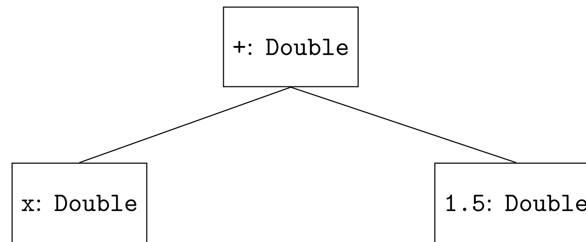

Figure 3: The expression tree for `x+1.5` where `x` is known to be of type `Double`

Upon assigning a value to a variable, the Hindley-Milner approach unifies the variable's type with the value's type. That is, the variable's type is modified to include the type of the value. Likewise when the type of the value of a return statement is inferred, the return type of the function is unified with the type of the return value. Many type inference implementations, such as the one developed by Johnson [5], describe a type as a set of classes. When types are unified here, the result of unification is simply the union of the two sets.

Graver [6] highlights type information flow as an important aspect of type inference algorithms, presenting two forms of type flow: data flow type information and requirement flow type information. Data flow type information is collected through definitions, such as assignments to variables, binding a function's parameter to an expression, or returning a value from a function. The type of the variable, parameter, or function must be compatible with the type of the expression. Data flow type information can be used to create a lower bound for a symbol's type. Requirement flow type information is gathered by inspecting the usage of a variable, function parameter, or return values from function calls. For instance in the expression `foo(2) + 1`

the return type of `foo` must be either `Integer` or `Double` since it is used in an addition. This creates an upper bound for a symbol's type.

Agesen [7] outlines an algorithm similar to Graver's approach. This algorithm is broken into three steps. The first step is to allocate memory for associating each symbol and expression with a type. Step two consists of identifying initial types for literals and variable declarations, similar to assigning types to leaf nodes in the Hindley-Milner algorithm. So `"hello"` is given the type `String` and if `x = "hello"` is encountered then `x` is assigned the type `String` as well. Step three identifies type constraints based on assignments, using data flow type information similar to Graver's approach. Note that this creates a lower bound on the type of symbols, but not an upper bound. In this step, encountering `x = y` causes `x`'s type to be unified with the type of `y`. Because the type of `x` has now changed, the types of other expressions that depend on `x` must be inspected again. This can be done iteratively until there are no more type changes. This is the approach most similar to the implementation in this project.

Palsberg and Schwartzbach take a different approach to type inference [8]. Rather than traversing expression trees, they create a graph representing constraints imposed by usages of symbols in a program. Vertices in the graph represent constraints imposed within a method and edges between vertices represent method calls. Method calls are used here to place constraints on edges relating formal and actual parameter types. The graph is used to generate an overall set of type constraints, to which a solution is found. This solution to the constraints effectively assigns types to the program's symbols.

3 Objectives

The Less-Java language design and a prototype compiler were contributed by Nasrawt [2]. However, there were several issues with the compiler at the beginning of this work. Specifically, there were no formal type rules for the language, there were no unit tests for testing the correctness of the compiler, there was no static analysis phase to ensure type safety of the program, type inference did not work with objects, object constructors were not behaving as expected when inheritance was used, and variables within functions with multiple parameter bindings were forced to have the same data type across bindings.

3.1 Type Rules for Less-Java

The Less-Java language as designed by Nasrawt did not include any formally written type rules for Less-Java. Static analysis and type inference, which are significant portions of the current project, both heavily rely on having a well-defined type system. This type system dictates which types can be used in different scenarios in a valid Less-Java program. Type inference uses this type system to determine the types of expressions. Static analysis implements the type rules to detect data type errors within a Less-Java program.

3.2 Unit Tests

Nasrawt provided sample programs that could be used to test that the compiler was working, but there were no unit tests within the compiler to verify that different parts of the compiler were working as intended. One of the goals for this project is to write unit tests to check for issues with the new and modified parts of the compiler.

3.3 Static Analysis

One of the bigger issues with the compiler before this work was a lack of error messages when the programmer presented the compiler with an invalid program. When an invalid program, such as in Figure 4, was compiled, the Less-Java compiler did not detect any errors. The compiler generated Java code that corresponded to the input Less-Java source, including the erroneous code. The errors were then detected and reported by the Java compiler (Figure 5). This is undesirable behavior because Less-Java is intended to be an introductory language. If errors are reported referencing the Java code rather than the Less-Java code that the programmer wrote, then the programmer is likely to be confused. The error messages reference a file that the programmer

didn't write, line numbers that don't correspond to the erring lines in the Less-Java file, and code that the programmer is unfamiliar with. To fix this issue and make the language more beginner friendly, this work added a static analysis phase to the compiler that detects errors in the Less-Java program before running the Java compiler.

```
1  main() {
2      for(i: 1 -> 2.5) {
3          a = 5
4          if (a + true) {
5              foo(true, a)
6          }
7      }
8      break
9  }
```

Figure 4: An invalid Less-Java program

```
generated/Main.java:18: error: bad operand types for binary operator '+'
            if (Boolean.valueOf((a+Boolean.valueOf(true))))
                                  ^
  first type:  Integer
  second type: Boolean
generated/Main.java:20: error: cannot find symbol
                foo(Boolean.valueOf(true), a);
                ^
  symbol:   method foo(Boolean,Integer)
  location: class Main
generated/Main.java:23: error: break outside switch or loop
        break;
        ^
3 errors
```

Figure 5: Output of compiling figure 4 before the project

3.4 Object-Oriented Type Inference

Less-Java is an object-oriented language, and inheritance is an important feature of object-oriented programming languages. If in a Less-Java program classes `Bike` and `Car` both inherit from `Vehicle`, then a variable of type `Vehicle` should be able to reference instances of either class `Bike` or `Car`. Additionally, functions should be able to accept objects as parameters. However, before this work, the compiler's implementation of type inference did not allow for either of these features. For example in the `main` function in Figure 6, the type of `var` is initially inferred to be `Car` on line 21. The assignment on line 22 should have changed the type of `var` to `Vehicle`, but it did not. The call to `doThing(trike)` on line 25 should have also created

a binding of `doThing` taking a `Bike` as a parameter, but it failed to do so. These two type inference issues meant that, before this project, the Less-Java compiler could not handle a single variable being assigned to instances of two different but related classes and did not allow functions to take objects as parameters. This work fixed the original type inference implementation to work with objects to allow both of these features.

```
1  Vehicle {
2      public numWheels = 0
3      Vehicle(numWheels) {
4          this.numWheels = numWheels
5      }
6  }
7
8  Car extends Vehicle {
9      Car() {
10         super(4)
11     }
12 }
13
14 Bike extends Vehicle {
15     Bike() {
16         super(2)
17     }
18 }
19
20 main() {
21     var = Car()
22     var = Bike()
23
24     trike = Bike(3)
25     doThing(trike)
26 }
27
28 doThing(param) {
29     // ...
30 }
```

Figure 6: A valid Less-Java program that failed to compile at the start of the project

3.5 Constructor Generation

One feature of Less-Java intended to cut down on verbosity is that constructors of superclasses are automatically pulled into subclasses. For instance in Figure 6, classes `Car` and `Bike` extend `Vehicle` and `Vehicle` has a constructor that takes a single parameter, so `Car` and `Bike` automatically inherit a constructor that takes one parameter and makes a call to `Vehicle`'s constructor with that parameter. In addition to this, classes also automatically get constructors that take no parameters. If the programmer defines any construc-

tors that clash with the automatically generated ones, such as the constructors in `Car` and `Bike` that take no parameters, then the conflicting generated constructors are discarded and the programmer's version is kept. The issue with the original implementation of this was that only one constructor from the superclass was copied down, and the superclass's no-parameter constructor would always be copied down even if the programmer had defined their own. This ended up producing invalid Java code that had two bodies for the constructor taking no parameters, as seen in Figure 7. This work fixes the issue.

3.6 Function Instantiation

Another feature of the Less-Java programming language is that functions with multiple possible parameter bindings only need to be defined once. When the compiler encounters a function call, it first checks the types of the arguments. If the types of all arguments are known and there is not already a binding of the function for those types, then a new instance of the function is created with the given parameter binding.

At the beginning of this project, multiple bindings of the same function would reference the same function block in the abstract syntax tree. This caused issues because it meant that every expression within the function had to have the same type across all bindings, which defeated the purpose of having multiple bindings. Given the code in Figure 8, the old version of the compiler produced an AST similar to that in Figure 9. This AST is no longer a tree since the ASTBlock node now has two parents. The main issue here is that the type of `succ` cannot be inferred. Type inference sees that `succ`'s type depends on `a`'s type, but tracing up the AST doesn't reveal what that type is. It could either be `Integer` or `Double`, depending on which binding is used.

```
1  import static org.junit.jupiter.api.Assertions.*;
2  import static wrappers.LJString.*;
3  import static wrappers.LJIO.*;
4
5  import org.junit.jupiter.api.Test;
6  import java.util.*;
7  import java.io.*;
8
9  import wrappers.*;
10 public class Main
11 {
12     public static void main(String[] args)
13     {
14         Bike trike;
15         Car var;
16         var = new Car();
17         var = new Bike();
18         trike = new Bike(Integer.valueOf(3));
19     }
20     private static class Vehicle
21     {
22         public Integer numWheels = Integer.valueOf(0);
23         public  Vehicle(Integer numWheels)
24         {
25             this.numWheels = numWheels;
26         }
27         public  Vehicle()
28         {
29         }
30     }
31     private static class Car extends Vehicle
32     {
33         public  Car()
34         {
35             super(Integer.valueOf(4));
36         }
37         {
38             super();
39         }
40     }
41     private static class Bike extends Vehicle
42     {
43         public  Bike()
44         {
45             super();
46         }
47         {
48             super(Integer.valueOf(2));
49         }
50     }
51 }
```

Figure 7: Java output of running the previous version of the compiler on Figure 6

```
1  successor(a) {
2      succ = a + 1
3      return succ
4  }
5
6  main() {
7      successor(1)
8      successor(1.0)
9  }
```

Figure 8: A Less-Java program demonstrating multiple parameter bindings

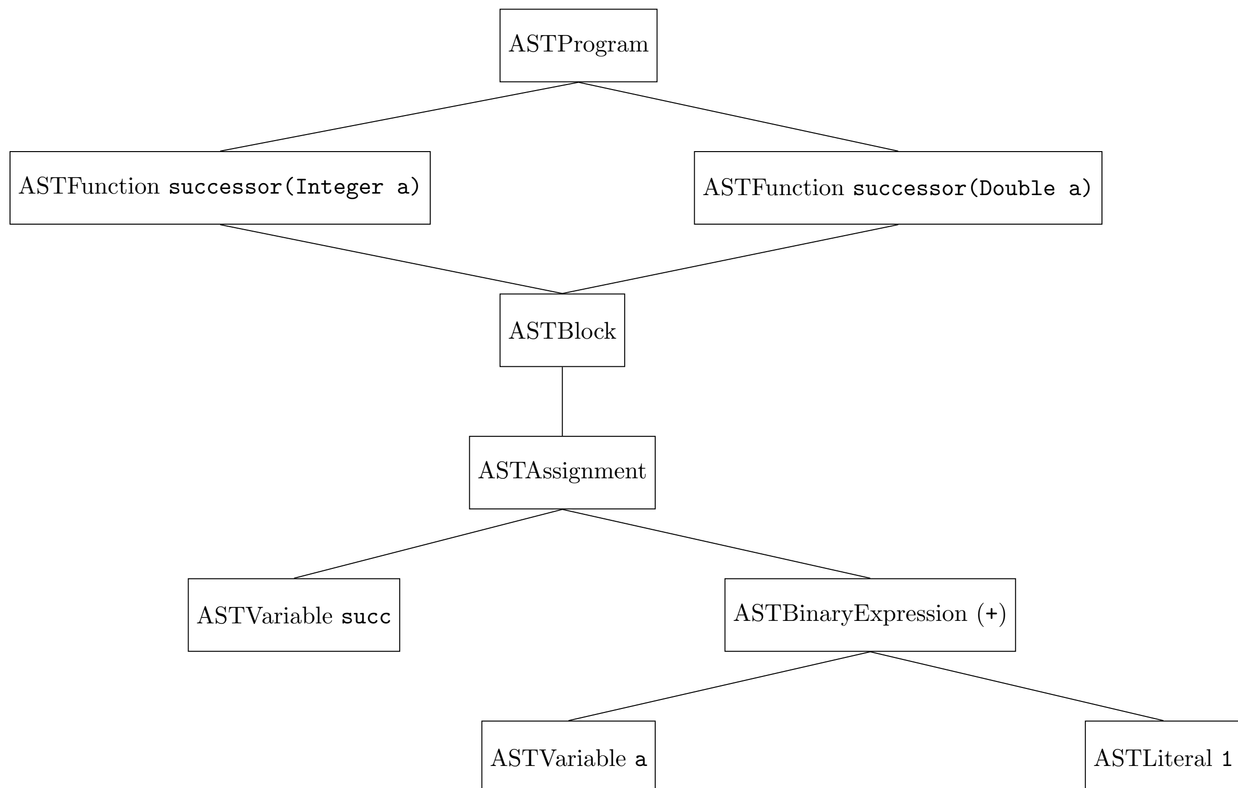

Figure 9: An incorrect AST for a function with multiple bindings

4 Results

This work addresses the issues with the Less-Java compiler and type system identified in section 3 by providing formalized type rules, unit tests for various components of the compiler, and a comprehensive static implementation of type checking. In addition, type inference works correctly in object-oriented programs, constructors are generated correctly, and functions with multiple parameter bindings are also working correctly.

4.1 Type Rules for Less-Java

The type rules formalized as part of this project determine validity of expressions and statements. Following the notation used in Pierce's well-known textbook [9], the type rules have a name to the side of the rule, premises above a line, and a conclusion below the line. Premises and conclusions are type judgements, asserting that an expression is of some type in some environment (often provided by symbol tables in a compiler). For example in TInt (Figure 10), there are no premises and the conclusion is that an `INT` token is of type `Integer`. As a slightly more complicated example, TIIAdd (Figure 11) has two premises: that both expressions e_1 and e_2 are of type `Integer` in environment Γ. The conclusion is that the sum of the two expressions within environment Γ is also of type `Integer`.

$$\text{TInt}\frac{}{\vdash \texttt{INT} : \textbf{int}}$$

Figure 10: The TInt type rule

$$\text{TIIAdd}\frac{\Gamma \vdash e_1 : \textbf{int} \quad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 \text{ `}\texttt{+}\text{` } e_2 : \textbf{int}}$$

Figure 11: The TIIAdd type rule

As an example of a type rule pertaining to a statement, TIf (Figure 12) shows that an `if` statement is well-typed in environment Γ if the condition is of type `Boolean` and the block is also well-typed in Γ.

$$\text{TIf}\frac{\Gamma \vdash e : \textbf{bool} \quad \Gamma \vdash b}{\Gamma \vdash \text{`}\texttt{if(}\text{` } e \text{ `}\texttt{)}\text{` } b}$$

Figure 12: The TIf type rule

For the rest of the type rules for Less-Java, see appendix A

4.2 Unit Tests

Unit tests for several parts of the Less-Java compiler were added as part of this project. Forty-six (46) new unit tests cover the static analysis, type inference, constructor generation, and function instantiation aspects of the compiler. The majority of the unit tests are focused on the static analysis and type inference portions of the compiler. These tests assert that a given Less-Java program is either valid or invalid. A test will attempt to compile a program through the static analysis phase of the compiler, and if no errors are found then it is marked as a valid program. Otherwise it is marked as invalid. If the compiler marks a program as valid and the test asserts that it is invalid, or vice-versa, then the test fails. Figure 13 shows an example of Less-Java programs used in unit tests. The program on the left must be asserted invalid since the `break` statement is not contained inside a loop body, while the program on the right must be asserted valid because the `break` statement is contained within a loop body.

```
main() {
    break
}
```

```
main() {
    while(true) {
        break
    }
}
```

Figure 13: Invalid (left) and valid (right) uses of a `break` statement tested as part of unit tests

Figure 14 shows programs used in unit tests that assert a program is invalid if an `if` statement does not have a `Boolean` condition. In the invalid program, the condition is an `Integer` rather than a `Boolean`.

```
main() {
    if(0) {}
}
```

```
main() {
    if(true) {}
}
```

Figure 14: Invalid (left) and valid (right) `if` conditions tested as part of unit tests

Figure 15 shows two more programs used in unit tests. The tests assert that a function can have different return types across different parameter bindings, but that a given parameter binding should only have one return type. Here the invalid program is invalid because the instance of `foo` taking an `Integer` parameter might return an `Integer` or a `Boolean`.

```
foo(a) {
    if(a == 0) {
        return true
    } else {
        return a
    }
}
main() {
    foo(true)
    foo(0)
}
```

```
foo(a) {
    return a
}
main() {
    foo(true)
    foo(0)
}
```

Figure 15: Invalid (left) and valid (right) functions

4.3 Static Analysis

Static analysis is the phase of compilation where the program is checked for correctness. This can be structural correctness or type correctness. An example of structural correctness is that `break` statements should only occur inside loop bodies. An example of type correctness is that if a variable's type is `Integer`, then it cannot be assigned a `String` value. Static analysis is implemented in this project using the visitor [10] design pattern. The visitor visits each node of the compiled program's AST. At each node, various conditions are checked to ensure no rules are being broken. At a node representing a `break` statement, for example, the visitor checks that the node has a loop node as an ancestor. At a node representing a function call, the visitor checks that the call is to a known function. Figure 16 shows the current output of the Less-Java compiler when run on the code in Figure 4 from earlier, whereas before this work the errors were not reported until the generated Java code was compiled by the Java compiler.

```
Line 2: For loops can only run through integers
Line 4: Cannot apply operator ADD with non-numeric right expression type Boolean
Line 4: Cannot unify types Integer and Boolean
Line 4: Integer is not a boolean expression
Line 5: Cannot find function foo with 2 arguments
Line 8: Break statement must be inside a loop
```

Figure 16: Output from compiling figure 4

4.4 Object-Oriented Type Inference

This project successfully modifies the previous implementation of type inference in the Less-Java compiler to work with objects in addition to the primitive data types it already supported. Allowing type inference for

objects is done by inspecting assignments to symbols. When a symbol is assigned a value of a given type, the type of the symbol is unified with the type of the value. In cases where the symbol and value are of primitive data types, the type of the symbol is unified with the type of the value in accordance with the original implementation. If the symbol and value are both objects, then the type of the symbol is unified with the type of the value by inspecting the class hierarchy. The unified type is the nearest common superclass of the types of the symbol and value. Consider again the program from Figure 6. Here `Car` and `Bike` both extend `Vehicle`. Because `var` is assigned values of type `Car` on line 21 and `Bike` on line 22, the type of `var` is inferred to be `Vehicle`, the nearest common superclass between `Car` and `Bike`. If there is no common superclass (this is possible because Less-Java does not have a rooted class hierarchy where all classes inherit implicitly from `Object` like in Java), then unification fails and a static analysis error is generated.

The original implementation of type inference was also changed to make use of data flow type information only, whereas before it also used requirement flow type information. Suppose in a Less-Java program, variables `a` and `b` are of type `Integer`. Now suppose the expression `a || b` is encountered. Using requirement flow type information here would attempt to unify the types of `a` and `b` with `Boolean`, since only `Boolean` types are compatible with the `||` operator. This would generate an error message that `Integer` and `Boolean` types cannot be unified, but this error message would not be very helpful to the programmer since the error message doesn't explain that the `||` operator requires `Boolean` operands. So instead of using requirement flow type information, type inference only uses data flow type information (information gathered by inspecting assignments). Errors such as using operands of incorrect types for a given operator are instead caught during static analysis, which allows for more specific and helpful error messages to be generated. In the previous example, the error message generated by static analysis explains that the `Integer` data type cannot be used with the `||` operator, which is a better explanation of the error.

4.5 Constructor Generation

This project also addresses the issue where only a single constructor from a superclass wasbeing copied into the subclass. To fix this problem, the AST nodes representing Less-Java classes now maintain a set of all of the class's constructors rather than a single constructor as in the previous implementation. Then when a subclass is defined, the set of the superclass's constructors is iterated over and each constructor is copied down into the subclass.

The issue that caused a superclass's zero-parameter constructor to always be copied into the subclass is also fixed by this project. This project modified the code to only copy the superclass's constructor into the

```
1  Vehicle {
2      private numWheels = 0
3      Vehicle(numWheels) {
4          this.numWheels = numWheels
5      }
6  }
7
8  Bike extends Vehicle {
9      Bike() {
10         super(2)
11     }
12 }
13
14 main() {
15     car = Vehicle(4)
16     bike = Bike()
17     trike = Bike(3)
18 }
```

Figure 17: Less-Java code demonstrating use of a superclass's constructor

subclass if the programmer hadn't already defined a constructor taking zero parameters.

Figure 17 demonstrates a sample program that takes advantage of having a superclass's constructor copied into the subclass. The call to `Bike(3)` on line 17 shows a call to a constructor defined in `Bike`'s superclass, `Vehicle`.

4.6 Function Instantiation

One final issue addressed by this project is that variables in a function were restricted to always being of the same type across each parameter binding of the function. The issue was caused because each AST node representing a binding of a function contained a reference to the same AST node representing the function's body. This made it so that each binding had identical implementations, including the types of variables.

This project addresses the issue by duplicating the AST node representing the function's body so that each binding can reference its own function body. This allows the types of symbols within each instance of the function to change independent of the types in other bindings of the same function. The function's body is duplicated using the visitor design pattern. The original function body is visited, building a stack of statements and a stack of expressions. When a node representing a new statement is visited, a node representing its copy is added to the top of the statements stack. After completing a visit to a statement node, the statement on top of the statements stack is popped off of the stack, populated with expressions from the top of the expressions stack (such as the condition for an `if` statement) and added to the copy

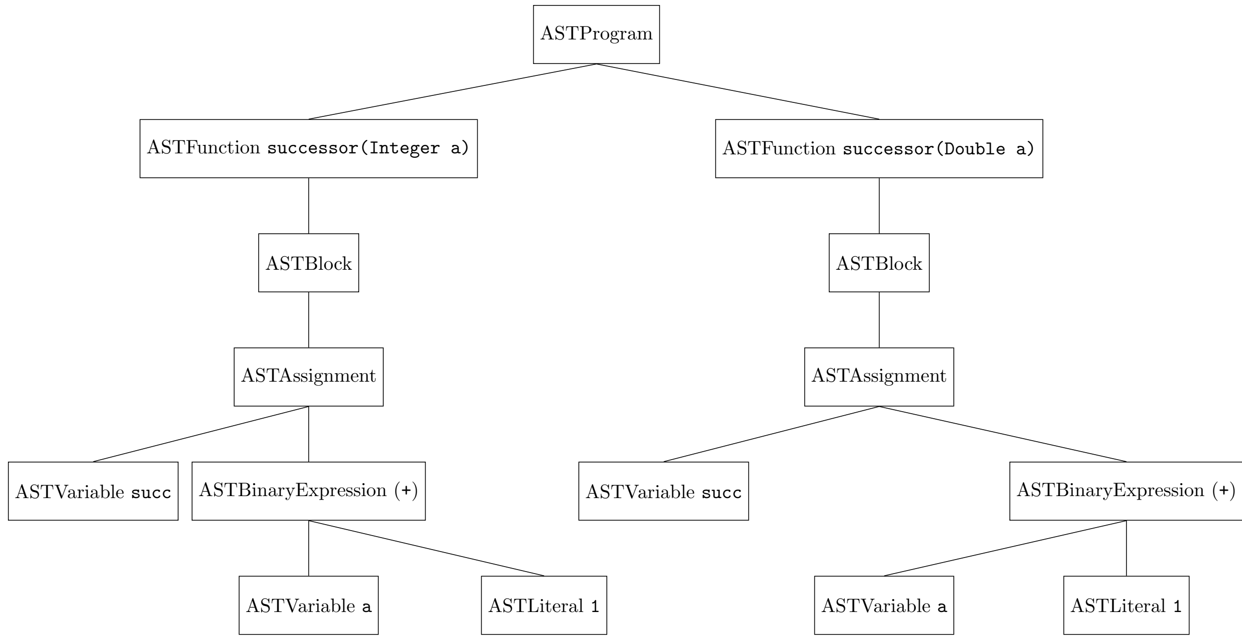

Figure 18: The correct AST for a function with multiple bindings as in Figure 8

of the function's implementation. When a new expression is visited, a copy of the expression is generated by popping expressions off of the top of the stack if needed (for example the left and right subexpressions if the expression being copied is a binary expression). This new expression is then pushed onto the stack. Duplicating an expression does not copy type information over to the copy, which ensures that types of expressions and symbols between copies are independent.

The end result of this duplication process is that a new function body is created that is identical to the original in every way except for type information. This allows the types of expressions in the duplicate to be changed without having side effects on the original implementation. The AST generated by the current implementation when compiling the code example from Figure 8 is illustrated in Figure 18. The ASTFunction nodes reference identical but independent implementations of the `successor` function, allowing the type of `succ` to be different in the two implementations.

5 Future Work

The Less-Java language and compiler are now much closer to a finished product as a result of this work. However, there are still several improvements to be made in future work.

One useful addition to the language would be to add file I/O capabilities. Currently the language is only capable of reading from and writing to standard output. File I/O is included in many introductory programming courses, so it is currently a notable exclusion from the Less-Java language. An ideal implementation would have safe ways of reading and writing files, but this would likely require some sort of exceptional control flow.

Adding basic exceptional control flow to Less-Java is itself another potential area for work in the future. The only way to achieve some form of exceptional control flow in Less-Java currently is through special return values, but this is limiting to the programmer because it removes a value from the possible outputs of a function. As an example demonstrating why this is not a good method of exceptional control flow, consider a function that finds and returns the key associated with some value in a map. If the function is called with a value that does not exist in the map, then the function should fail. Currently, the only way to indicate that the function failed is to have it return a special value, say `"FAILED"`, and have the caller check for this return value. But then there's the possibility that `"FAILED"` actually *was* the key to the specified value. So instead of just checking whether the function returned `"FAILED"`, the programmer would also have to check that `"FAILED"` doesn't map to the specified value. This would be much cleaner with some actual form of exceptional control flow such as the `try/catch` mechanism provided by Java.

Another improvement for Less-Java would be to have runtime errors reference lines in the Less-Java file, rather than lines in the Java file that the compiler produces. For example, if a number is divided by zero the Java runtime throws an exception and the program crashes, printing the exception to the screen. This exception references the line number where the division by zero occurred, but this line number does not correlate to the line in the Less-Java file. This can cause frustration when debugging a Less-Java program, because it requires looking into the generated Java code, finding the Less-Java code that corresponds to the Java code, and then fixing the issue. This process would be much simpler if the runtime could be made to reference the Less-Java code rather than the Java code.

Less-Java's ultimate goal is to be a better introductory programming languages than other commonly taught programming languages. However, this claim remains untested. Once the language and compiler have been more thoroughly refined, an empirical study should be conducted to test whether or not Less-Java is actually a more effective introductory programming language than other popular languages like Java.

6 Conclusion

Prior to this project, the Less-Java compiler had several major flaws. There was no formal type system available for the language, the compiler had no unit test coverage, Less-Java programs were not inspected for type errors, type inference with respect to objects was not working properly, constructors in subclasses were not being generated properly, and variables within functions were restricted to having the same type across different parameter bindings. This project has addressed all of these issues. Additionally, this is the first instance of type inference being used in an object-oriented language with strong static typing that the author is aware of.

The source code for the Less-Java compiler (including all improvements contributed in this project) is available at `https://www.github.com/JMU-CS/less-java`.

A Less-Java Type Rules

A.1 Expressions

$$\text{TInt}\,\frac{}{\vdash \texttt{INT} : \textbf{int}} \qquad \text{TDouble}\,\frac{}{\vdash \texttt{REAL} : \textbf{double}} \qquad \text{TBool}\,\frac{}{\vdash \texttt{BOOL} : \textbf{bool}} \qquad \text{TStr}\,\frac{}{\vdash \texttt{STR} : \textbf{string}}$$

$$\text{TList}\,\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad ... \quad \Gamma \vdash e_n : \tau}{\Gamma \vdash \text{ ‘}\texttt{[}\text{‘ } e_1, e_2, ..., e_n \text{ ‘}\texttt{]}\text{‘} : List(\tau)}$$

(Similar for TSet)

$$\text{TMap}\,\frac{\Gamma \vdash k_1 : \tau_k \quad \Gamma \vdash k_2 : \tau_k \quad ... \quad \Gamma \vdash k_n : \tau_k \quad \Gamma \vdash v_1 : \tau_v \quad \Gamma \vdash v_2 : \tau_v \quad ... \quad \Gamma \vdash v_n : \tau_v}{\Gamma \vdash \text{ ‘}\texttt{<}\text{‘ } k_1 : v_1, k_2 : v_2, ..., k_n : v_n \text{ ‘}\texttt{>}\text{‘} : Map(\tau_k, \tau_v)}$$

$$\text{TMapAccess}\,\frac{\texttt{ID} : \texttt{Map}(\tau_k \rightarrow \tau_v) \in \Gamma \quad \Gamma \vdash e : \tau_k}{\Gamma \vdash \texttt{ID} \text{ ‘}\texttt{[}\text{‘ } e \text{ ‘}\texttt{]}\text{‘} : \tau_v} \qquad \text{TListAccess}\,\frac{\texttt{ID} : \texttt{List}(\tau) \in \Gamma \quad \Gamma \vdash e : \textbf{int}}{\Gamma \vdash \texttt{ID} \text{ ‘}\texttt{[}\text{‘ } e \text{ ‘}\texttt{]}\text{‘} : \tau}$$

$$\text{TSubExpr}\,\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{‘}\texttt{(}\text{‘ } e \text{ ‘}\texttt{)}\text{‘} : \tau} \qquad \text{TVar}\,\frac{\texttt{ID} : \tau \in \Gamma}{\Gamma \vdash \texttt{ID} : \tau}$$

$$\text{TMember}\,\frac{\texttt{ID}_o : \tau_o \in \Gamma \quad \tau_o <: \{\texttt{ID}_m : \tau\}}{\Gamma \vdash \texttt{ID}_o\texttt{.}\texttt{ID}_m : \tau}$$

$$\text{TIIAdd}\,\frac{\Gamma \vdash e_1 : \textbf{int} \quad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 \text{ ‘}\texttt{+}\text{‘ } e_2 : \textbf{int}}$$

(Similar for TIIMul (*), TIISub (-), TIIDiv (/), and TMod (%))

$$\text{TIDAdd}\,\frac{\Gamma \vdash e_1 : \textbf{int} \quad \Gamma \vdash e_2 : \textbf{double}}{\Gamma \vdash e_1 \text{ ‘}\texttt{+}\text{‘ } e_2 : \textbf{double}}$$

(Similar for TIDMul (*), TIDSub (-), TIDDiv (/), TDIAdd (+), TDIMul (*), TDISub (-), and TDIDiv (/))

$$\text{TEq}\,\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \text{ ‘}\texttt{==}\text{‘ } e_2 : \textbf{bool}}$$

(Similar for TNEq(!=))

$$\text{TFuncCall}\,\frac{\texttt{ID} : (\tau_1, \tau_2, ..., \tau_n) \rightarrow \tau \in \Gamma \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad ... \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \texttt{ID} \text{ ‘}\texttt{(}\text{‘ } e_1, e_2, ..., e_n \text{ ‘}\texttt{)}\text{‘} : \tau}$$

(Similar for TMethodCall)

A.2 Statements

$$\text{TIf}\,\frac{\Gamma \vdash e : \textbf{bool} \quad \Gamma \vdash b}{\Gamma \vdash \text{‘}\texttt{if(}\text{‘ } e \text{ ‘}\texttt{)}\text{‘ } b} \qquad \text{TIfElse}\,\frac{\Gamma \vdash e : \textbf{bool} \quad \Gamma \vdash b_1 \quad \Gamma \vdash b_2}{\Gamma \vdash \text{‘}\texttt{if(}\text{‘ } e \text{ ‘}\texttt{)}\text{‘ } b_1 \text{ ‘}\texttt{else}\text{‘ } b_2}$$

$$\text{TWhile}\,\frac{\Gamma \vdash e : \textbf{bool} \quad \Gamma \vdash b}{\Gamma \vdash \text{‘}\texttt{while(}\text{‘ } e \text{ ‘}\texttt{)}\text{‘ } b} \qquad \text{TFor}\,\frac{\texttt{ID} : \textbf{int} \in \Gamma \quad \Gamma \vdash e_1 : \textbf{int} \quad \Gamma \vdash e_2 : \textbf{int} \quad \Gamma \vdash b}{\Gamma \vdash \text{‘}\texttt{for(}\text{‘ } ID \text{ ‘}\texttt{:}\text{‘ } e_1 \text{ ‘}\texttt{->}\text{‘ } e_2 \text{ ‘}\texttt{)}\text{‘ } b}$$

References

[1] Zamua O Nasrawt. "Less-java, more learning: Language design for introductory programming". 2018.

[2] Zamua Nasrawt. "Less-java, more learning: Language design for introductory programming". In: *Senior Honors Projects, 2010-current* (May 5, 2018). URL: https://commons.lib.jmu.edu/honors201019/598.

[3] Robin Milner. "A theory of type polymorphism in programming". In: *Journal of computer and system sciences* 17.3 (1978), pp. 348–375.

[4] Monoceres, sepp2k, and Andreas Rossberg. *Why is type inference impractical for object oriented languages?* 2014. URL: https://stackoverflow.com/questions/22528534/why-is-type-inference-impractical-for-object-oriented-languages.

[5] Ralph E. Johnson. "Type-checking Smalltalk". In: *SIGPLAN Not.* 21.11 (June 1986), pp. 315–321. ISSN: 0362-1340. DOI: 10.1145/960112.28728. URL: http://doi.acm.org/10.1145/960112.28728.

[6] Justin Owen Graver. "Type Checking and Type Inference for Object-oriented Programming Languages". AAI9010868. PhD thesis. Champaign, IL, USA, 1989.

[7] Ole Agesen. "Concrete Type Inference: Delivering Object-oriented Applications". UMI Order No. GAX96-20452. PhD thesis. Stanford, CA, USA, 1996.

[8] Jens Palsberg and Michael I. Schwartzbach. "Object-oriented Type Inference". In: *SIGPLAN Not.* 26.11 (Nov. 1991), pp. 146–161. ISSN: 0362-1340. DOI: 10.1145/118014.117965. URL: http://doi.acm.org/10.1145/118014.117965.

[9] Benjamin C. Pierce. *Types and programming languages.* The MIT Press, 2002.

[10] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201633612.