CS644 Project Report: A4

Submitted in partial fulfillment of the requirements for the course

Submitted by

Name	ID
Siwei Yang	s8yang
Justin Vanderheide	jtvander
Jian Li	j493li

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada – N2L 3G1

Winter 2015

1 General Strategy

This section describes the meta strategy we applied in developing solutions to each of the deliverables. In principle, we are using the attribute grammar approach to finish all the tasks: name resolution, type linking and static analysis.

However, it takes more than merely coding the attribute rules to solve the problem. Auxiliary information might be needed at every node to make the right decision, and the information flows both ways which is hard to support in a pure functional language like Haskell. We would also like to achieve a nice separation between the supporting data structures and the grammar structures.

Therefore, we modeled our solution as three parts: a transformed data structure which captures the essence of the AST as well as supporting bi-directional information flow, a newly built data structure that answers queries from the attribute rules and the implementations of the attribute grammars.

1.1 Engineering a Pure Functional Double-linked AST

Our first major task was to re-structure the AST so that all the relevant information remains and is easily accessible while processing each node. This is trivial in languages that support mutation where adding a reverse pointer to each AST node solves the problem. However, we are working with a purely functional language. Adding reverse pointer is not viable as explicit pointers are not provided in the language. Therefore, we introduced a two layer structure of *SemanticUnit*s and *Environments*.

```
data SemanticUnit = Root {
    scope :: [String]
}                                | SU {
    scope :: [String],
    kind :: Kind,
    symbolTable :: [Symbol],
    inheritFrom :: SemanticUnit
} deriving (Eq)

data Environment = ENVE | ENV {
    semantic :: SemanticUnit,
    children :: [Environment]
} deriving (Eq)
```

We traverse the AST from top to bottom, meanwhile we build the environment using the information in AST nodes. The structure of the environment is basically a recursive tree. For each of the tree nodes, it has a *SemanticUnit* which contains the scope info, AST, symbol table and the parent pointer. The configuration of the symbol table is a linked list (or a list) corresponding to the path from the current tree node to the root, and every element in the symbol table is a structure *Symbol*. The *Symbol* can represent either Package, Class, Interface, Function or Variable, and includes all AST information we need.

Environment is a recursive data structure that goes top-down, so that down-stream information is naturally available. To get the up-stream information, we follow the *SemanticUnit* all the way to the desired ancestors.

1.2 Consolidating Type Hierarchy into a Database

To begin with, we put all the type information together into a tree of *TypeNodes*.

```
data TypeNode = TN {
    symbol :: Symbol,
    subNodes :: [TypeNode]
} deriving (Eq)
```

To support the searches executed on this type database, we also created a few helper methods to cope with the special rules from the JLS. The two most important ones were *traverseTypeEntry-WithImports* and *traverseInstanceEntry'*.

```
traverseTypeEntryWithImports :: TypeNode -> [[String]] -> [String] -> [[
   String]]
traverseTypeEntryWithImports tn imps query = nub [cname | (Just node, cname)
    <- results]
    where
        entries = map (traverseTypeEntry tn) imps
        entries' = map (\(mnode, imp) -> (fromJust mnode, imp)) (filter (
           isJust . fst) (zip entries imps))
        results = map (\(node, imp) -> (traverseTypeEntry node query, (init
           imp) ++ query)) ((tn, ["*"]):entries')
```

traverseTypeEntryWithImports supports type name lookups taking into account all the imports. *traverseInstanceEntry'* is omitted for brevity, but it takes care of chains of attribute accesses.

1.3 Coding Attribute Rules

The attribute rules are translated into structural recursion rules that work on the *Environment*. Here, we show a simplified segment of our code for illustration purpose.

```
typeLinkingCheck db imps (ENV su c) = if elem Nothing imps' then [] else tps
    where
        tps = case kd of
                Var expr ->
                Field varsym (Just expr) ->
                Exp expr ->
                Ret expr ->
                ForBlock ->
                WhileBlock expr ->
                IfBlock expr ->
                Class ->
                Method _ ->
                _ ->
```

The expressions are checked by similar code.

2 Name Resolution and Type Linking

2.1 Type Linking

Before we reach the type linking stage, we have already constructed the symbol table and type database which can be used for looking up local variables, classes, and interfaces. However, the Joos language supports both single-type-import and import-on-demand. This allows programmers to use the names of classes which are not in the same package. This feature significantly complicates name resolution. Our solution is to first look up the name in the symbol table, if we can find a binding for this name we return the type declared for this variable. Otherwise, it is not in the symbol table, then we look in our type database to check whether it is the name of a class or interface. If we cannot find it in the type database either, then we report that we cannot resolve this name.

2.2 Type Checking

Assignment 2 only required us to do type linking which is a part of name resolution. However, when we were doing that part, we already had the entire environment and type database. This meant we could evaluate the type of each expression and return a list of possible types for that expression. This is a recursive process since the type attribute of expression are synthesized. Each time we pass the current node's environment to it's children, evaluate the types of subexpressions(children), and then combine all types of subexpressions with the format of the current expression to get the type of this expression. If the return of an expression is an empty list, then we know there exists an error when we evaluated a subexpression or this expression, then we pass this empty list to the upper caller, i.e. propagating the error info to the root. Or if the return has multiple types, there must exist some ambiguous name, in this case, we also report the error.

2.3 Type Conversion

Another issue we dealt with was type conversion. We had to relax some strict type equality checking to a more complicated casting type system defined by conversion rules, since Joos supports explicit type casting and implicit type conversion in some kinds of expressions. This part is mainly implemented in the file *TypeChecking.hs*. According to the JLS, there are many kinds of conversions for primitive types and references types. The conversions used in assignment(operator {=}) and casting are also slightly different. For the binary operators {==, !=}, there are also other rules for this equality checking. We tried to summarize the common parts of these rules, and only change as few values as possible when dealing with different cases. This is why we split the conversion into two parts: *primitiveConversion* and *objectConversion*, and reuse them in *assignConversion*, *castConversion* and *equalityCheck*.

3 Static Analysis

Reachability Checking is implemented in the file *Reachability.hs*. In order to make the checking easier, two separate tests are done. First a reachability test is performed, then a second test is performed to see if a function can terminate without a return. The test for unreachability simply implements the rules from section 14.20 of the JLS. The standard case is that a statement can complete normally iff it is reachable. There are few exceptions to this rule, which were handled by pattern matching. Return statements, whiles, fors, and ifs are pattern matched along with their condition expressions and the rules specified in the JLS. The function that tests for unreachability returns a list of unreachable statements so that they can be shown to the programmer.

The test for function completion without a return also implements the rules from section 14.20 of the JLS, this time with an emphesis on whether or not a code path to the end of the function without a return exists.

This separation of concerns made the reachability testing much easier to implement.

4 Challenges

Last but not least, we would like to cover some technicalities involving the following challenges. Each further demonstrates how we model the solution, and how we engineer solution taking into account the characteritics of pure functional language.

4.1 Type Queries

We have briefly talked about the structure and primitive operations supported by the type database. However, it takes more than a tree traversal to find the correct type of each "name", simple or qualified. The first enhancement to the data structure is to support parallel search with respect to each of the imports. So that all possible matches can be found. The second enhancement is to support access control of static and instance field and methods.

4.2 Instance Queries

Instance query is somewhat similar to the type query, except the traversal of type database can go back and forth multiple times.

```
traverseInstanceEntry' root (TN (SYM mds ls ln lt) _) (nm:cname) =
    traverseInstanceEntry' root root ((typeToName lt) ++ (nm:cname))
```

Note this special rule guarantees attribute access on a field can be redirected to the type node corresponds to the field. We also wraps this method with another one that take care of access control on each attribute access.

4.3 Hierarchy Queries

The most challenging part of hierarchy checking was implementing the third rule from JLS 9.2, that any interface with no superinterface must implicitly declare public abstract versions of all methods in Object. This posed a problem because it is essentially an interface that is extending a class. We implemented this by creating an interface called "ObjectInterface" that has all of the signatures from Object in it, and having all interfaces without a super use that interface. There were problems with clone() and getClass() however, since clone is protected not private, and getClass is final. Neither the protected or final keyword are allowed in interface method definitions, so the ObjectInterface was being rejected by the weeder, but these modifiers were required to ensure interfaces didn't incorrectly overwrite these methods. To get around this we simple skip ObjectInterface in the weeder, everything works out fine once we get to hierarchy checking!

Another aspect of hierarchy checking that required careful consideration was getting the hierarchy chain for interfaces. Since cycles can appear in the hierarchy chain, it was important to have cycle detection in our function that finds the chain. "getInterfaceSupers", the function that returns a list of all interfaces in the hierarchy chain, does depth first search and aborts upon reaching an interface twice. Rather than having a global visited list, each split has it's own visited list since common ancestors are valid. After the split the results are unioned together to get the set of all super interfaces.

4.4 Forward-references Detection

Catching forward-reference should be trivial where detecting usage of unavailable variable names suffices. However, to maximize code reuse, our first attempt is to try hacking the symbol table where unavailable variables are moved out of the symbol so that processing of the initialization expression can go on without modification. Unfortunately, this solution fails to work on local variable declararions where hacking the local symbol table is not enough since the same name can appear at higher level. So we went with the detection solution in the end.

```
Field varsym (Just expr) ->
  let syms = dropWhile (varsym /=) [sym | sym@(SYM mds _ _ _) <- sti , not $
     elem "static" mds]
    forward = or (map (\sym -> identifierInExpr (localName sym) expr) syms)
  in if forward
      then typeLinkingFailure $ "forward␣use␣of␣syms␣" ++ (show varsym) ++ (
         show expr)
      else if typeLinkingExpr db imps su (Binary "=" (ID (Name ([localName
         varsym])) 0) expr 0) == [] then typeLinkingFailure $ "field␣type␣
         mis␣match␣expression␣" ++ (show varsym) ++ (show expr) else cts'
```

4.5 Autoboxing and unboxing

Joos supports autoboxing from a primitive type to a corresponding reference type, for example, primitive type *int* can automatically convert to *Object Integer* which is predefined in Java standard library. Also we can do unboxing with some specific predefine reference types, then get a primitive type. So we implemented functions *boxingType* and *unboxingType* so that we can support autoboxing and unboxing features. However, *String* is a special case here, because Java does not define a primitive type for literal string. It means every literal string can be thought with type *Object String*. So we remove our predefine primitive type *TypeString* for literal string from AST, and replace every *TypeString* with type *Object (Name ["java", "lang", "String"])*(need a canonical name here).

4.6 Static Analysis

The biggest challenge in implementing static analysis was implementing constant folding which we did not have previously. We wrote a function called conditionConstant which attempts to evaluate an expression, and returns an Either value. If we can determine the value of an expression then the return value is Right Int, where the Int represents the computed value. For booleans 0/1 is used rather than True/False since our function can only have 1 return type. If we cannot determine the value of the expression then Left () is returned, indicating we cannot figure it out. These values are used in reachability testing to see if the conditions for various blocks are a constant true or false.

5 Debugging and Testing Strategies

In assignment 1 we wrote our test runners in Haskell, however assignments 2-4 were significantly more involved than assignment 1 so we needed a more sophisticated approach. We created a bash script that automatically runs all of the valid and invalid tests. It handles passing all of the necessary files from the standard library as well as all java files from a directory for tests with multiple files. Internally the script exports various helpful functions that make invoking the compiler easier. For example the function "joosc_run" takes in the path to either a single file test or a test directory and runs joosc with the appropriate standard library files and test directory files.

Because some assignments have over 300 test cases it was taking quite a while to run the test suite. It is important that our test suite runs quickly so that we can run it often and identify regressions sooner rather than later. GNU Parallel was used to distribute the testing across all available cores, causing a 4x speedup.

Our test suite results always matched the results given by marmoset, so we always knew that our submissions would suceed and didn't have to frequently submit to marmoset just to test if our compiler was working.