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 deliveribles. 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 is a pure functional language like Haskell. Besides, we would like to achieve a nice separation between the supporting data structures and the grammar structures.

Therefore, we modelled our solution as three parts: a transformed data structure captures the essense of the AST as well as supporting bi-directional information flow, a newly build 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 work is to re-structure the AST so that all the relevant information remains, yet easily accessible while processing each node. This is trivial in language supports mutation where adding a reverse pointer to each AST node solves the problem. However, we are working with a pure functional language. Adding reverse pointer is not viable as explicit pointer is not provided in the langauge. Therefore, we introduced a two layer structure of *SemanticUnit* and *Environment*.

```
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 environment using the information in AST nodes. The structure of the environment is basically a recursive tree. For each of tree node, it has a *SemanticUnit* which capsules scope info, AST, symbol table and the parent pointer. The configuration of 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 include 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 *TypeNode*.

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

To support the searches operated on this type database, we also created a few helper methods to cope with the special rules from JLS. Two most important of them are the *traverseTypeEntryWithImports* 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')
```

```
traverseInstanceEntry' :: TypeNode -> TypeNode -> [String] -> [TypeNode]
traverseInstanceEntry' root tn@(TN (SYM _ _ _ _) _) [] = [tn]
traverseInstanceEntry' root tn@(TN (FUNC _ _ _ _ _) _) [] = [tn]
traverseInstanceEntry' root tn [] = []
traverseInstanceEntry' root (TN (SYM mds ls ln lt) _) (nm:cname) =
   traverseInstanceEntry' root root ((typeToName lt) ++ (nm:cname))
traverseInstanceEntry' root cur (nm:cname) = case [node | node <- subNodes
   cur, (localName . symbol) node == nm, (not $ isSYMFUNCNode node) || (not
   $ elem "static" ((symbolModifiers . symbol) node))] of
                                        []          -> []
                                        targets     -> concat $ map (\
                                           target -> traverseInstanceEntry'
                                           root target cname) targets
```

traverseTypeEntryWithImports supports type name lookups taking into account all the imports. And *traverseInstanceEntry'* takes care of chains of attribute accesses.

1.3 Coding Attribute Rules

The attribute rules are translated into structural recursion rules that works 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 code with similar nature. Due to the limitation of space, we are not going to show it here though.

2 Name Resolution and Type Linking

2.1 Type Linking

Before we move on to this stage, we already got the symbol table and type database which can be used for looking up local variables and classes(interfaces as well) respectively. However, the Joos language supports both single-type-import and import-on-demand. So we can directly use the names of classes which are not in the same package. This feature would cause lots of troubles during name resolution. Our solution is first looking up the name from 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 up our type database to check whether it is a name of class or interface. If we cannot find in the type database either, then we claim that we cannot resolve this name.

2.2 Type Checking

The assignment 2 only needs to do type linking which is a part of name resolution. However, when we were doing that part, we already got entire environment and type database. So 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 environment to children, evaluate the types of subexpressions(children), and then combine all types of subexpressions with the format of 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 just 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 need to report such error.

2.3 Type Conversion

Another issue need to deal with is type conversion. We have to relax some strict type equality checking to a more complicated casting type system defined by conversion rules, since Joos language 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 JLS, there are many kinds of conversions for primitive types and references types. And the conversions used in assignment(operator {=}) and casting are also slightly different. Besides, for the binary operators {==, !=}, they are also another rules for this equality checking. We were trying to summarize the common parts of these rules, and only change as few as possible when we deal 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

3.1 Attribute Recursion Rules

3.2 Handling Base Cases

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

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).

5 Debugging and Testing Strategies

Considering the complexity of the tasks at hand, we applied coding tricks to facilitate debugging in development stage. Note we indicate type linking error by providing an empty type candidate list. This leads to a clean propagation of error. However, at development times, this also means we do not have specific error information available to us. So we created an wrapper on error reporting so that we can have the error message exposed to us in development stage. Yet, the message can easily be turned off for submission.

```
typeLinkingFailure :: String -> [Type]
--typeLinkingFailure msg = error msg
typeLinkingFailure msg = []
```

Note by swapping the definition of *typeLinkingFailure*, we can either let type check fail silently or reporting the error but terminate the program abnormal.