Adding Friend Class and Friend Function to MiniJava

cs20b019 C Sai Prasanna cs20b070 Sasubilli Yuvan

November 4, 2022

CS3100 Paradigms Of Programming Project

1 Brief Motivation

1.1 Overview

This Project is about extending the Java language with a Friend class and a Friend function construct. The goal is to take a subset of Java as a case study (the subset is Minijava with some modifications here) and show that extending our language with this construct can be beneficial. With the currently available constructs in Java, private members of a class are not accessible from outside the class but there are instances where access to private members outside the class is helpful. With the help of the friend construct, classes and functions can access private members of another class in which they are declared as a friend.

2 Idea of Construct

A **Friend class** can access private members of the class in which it has been declared as a friend.
A **Friend function** is a special function which in spite of not being a member function of a class has the privilege to access the private data of a class.
For Reference The MiniJava Language MiniJava Type System

```
<class declaration>    ::= "class" <identifier> <optional inheritance> "{"
                           <declaration list> "}"
<optional inheritance> ::= "extends" <identifier> | epsilon
<declaration list>     ::= <variable declaration>* <method declaration>* <friend class>*
                           <friend function>*
<variable declaration> ::= <AccessType> <type> <identifier> ";"
<method declaration>   ::= <AccessType1> <type> <identifier> "(" <opt formals> ")"
                           "{" <statement list> "}"
<friend class>         ::= "friend" "class" <identifier>
<friend function>      ::= "friend" "function" <identifier> <type> <identifier>
<type list>            ::= <type> "," <type list> | epsilon
<type>                 ::= <simple type> <opt brackets>
<simple type>          ::= "int" | "boolean" | "void" | <type identifier>
<opt brackets>         ::= "[" "]" | epsilon
<type identifier>      ::= <identifier>
<AccessType>           ::= "public" | "private" | epsilon
<AccessType1>          ::= "public" | epsilon
```

3 Impact

The impact of adding the friend class and friend function gives access permission of private members of a class to classes and friends(relaxing the definition of encapsulation) which are friends of that particular class. A friend function can access private data in objects of different class types simultaneously.

4 Motivation

4.1 Code in Java

4.1.1 Example requiring access of private members of a class from another class

```
class Vehicle {
  private int brand = 20;        // Vehicle attributes
  public int number = 1234;
  public void honk() {                    // Vehicle method
    System.out.println(10);
  }
}

class Automobile {
  Vehicle v = new Vehicle();
  public void info() {
      System.out.println(v.brand);
  }
}

class Car extends Vehicle{
  private int modelNumber = 30;    // Car attribute
  public static void main(String[] args) {
    Car myCar = new Car();
    myCar.honk();
    System.out.println(myCar.brand + myCar.modelNumber );
  }
}
```

myCar.brand statement gives a compile time error

- For the above code to work, the class Car must be given access to the private member **brand** of the class Vehicle . We could have made brand a public member in the class Vehicle but we do not want the class Automobile to access it
- From the currently available constructs in MiniJava, we cannot grant access to private members of a class such that they can be accessible from only a certain set of other classes

4.1.2 Example requiring access of private members of a class from a function outside it

```
class Sales_data {
  private int revenue = 100;
  Sales_data(int number){ revenue = number;  }
};

class Book_data {
 private int bookNo;
 private int NoofPages;
}

class Bookinfo {
  public void read( Book_data book ,Sales_data item ) {
   System.out.print("Book : "+book.bookNo+" Revenue is "+ item.revenue);
   return ;
  }
};
```

System.out.print statement in read function gives a compile time error

- In the above example, the function **read** needs access to the private member **bookNo** of class **Book_data** and **revenue** of class **Sales_data**
- From the currently available constructs in MiniJava, we cannot access the private members of a class from a function to which an object of the class is passed as an argument.

4.2 Improvement after adding the new construct

4.2.1 Example requiring access of private members of a class from another class

```
class Vehicle {
  private int brand = 20;                 // Vehicle attributes
  public int number = 1234;
  public void honk() {                    // Vehicle method
    System.out.println(10);
  }
  friend class Car;                       // friend class
}

class Automobile {
  Vehicle v = new Vehicle();
  public void info() {
      System.out.println(v.brand);
  }
}

class Car extends Vehicle{
  private int modelNumber = 30;    // Car attribute
  public static void main(String[] args) {
    Car myCar = new Car();
    myCar.honk();
    System.out.println(myCar.brand + myCar.modelNumber );
  }
}
```

The object **myCar** can now access the brand by making the class Car friend of class Vehicle

- In the above improvement class Car is declared as a friend inside the class Vehicle. Therefore, Car is a friend of class Vehicle.
- Class Car can now access the private members of class Vehicle but Class Automobile cannot.

4.2.2 Example requiring access of private members of a class from a function outside it

```
class Sales_data {
  private int revenue = 100;
  Sales_data(int number){ revenue = number;  }
  friend function Bookinfo void read( Book_data , Sales_data );      //friend function
};

class Book_data {
 private int bookNo;
 private int NoofPages;
 friend function Bookinfo void read( Book_data , Sales_data );      //friend function
}

class Bookinfo {
  public void read( Book_data book ,Sales_data item ) {
   System.out.print("Book : "+book.bookNo+" Revenue is "+ item.revenue);
   return ;
  }
};
```

read function can now access the private members of Sales_data class and Book_data class

- In the above improvement i.e after declaring read function as a friend in class Sales_data, the read function gets access to all the members of the class Sales_data
- The keyword **friend** is placed only when the function is being declared as a friend inside a class and not in the function definition.

4.3 Advantages of the Proposed scheme

Friend Class : It is sometimes useful to allow a particular class to access private members of other class and using friend class we can achieve this construct in MiniJava. It is useful when access to private members of a class is required by only a subset of other classes rather than the whole program.
Friend Function : Friends functions come into play when a particular function can be used to perform a task on data from different classes simultaneously. By making a function friend to multiple classes the function gets access to private members of different classes.

4.4 Possible Drawbacks

The Friends construct should be used for a limited purpose. If too many friend functions/classes are declared in a class with private and protected data, the true purpose of encapsulation achieved with different classes in object-oriented programming will be lost.

5 Semantics

5.1 Rules of TypeChecking

5.1.1 Helper functions

6.1 The *classname* Helper Function

The function *classname* returns the name of a class. The definition of *classname* is:

$$classname(\texttt{class}\ id\ \{\ \texttt{public static void main (String [] }id^S\texttt{) }\{\ \ldots\ \}\}) = id \tag{5}$$

$$classname(\texttt{class}\ id\ \{\ t_1\ id_1;\ \ldots;\ t_f\ id_f;\ m_1\ \ldots\ m_k\ \}) = id \tag{6}$$

$$classname(\texttt{class}\ id\ \texttt{extends}\ id^P\ \{\ t_1\ id_1;\ \ldots;\ t_f\ id_f;\ m_1\ \ldots\ m_k\ \}) = id \tag{7}$$

6.2 The *methodname* Helper Function

The function *methodname* returns the name of a method definition. The definition of *methodname* is:

$$methodname(\texttt{public}\ t\ id^M\ \texttt{(}\ldots\texttt{)}\ \ldots) = id^M \tag{8}$$

6.3 The *distinct* Helper Function

The *distinct* function checks that the identifiers in a list are pairwise distinct.

$$\frac{\forall i \in 1..n : \forall j \in 1..n : id_i = id_j \Rightarrow i = j}{distinct(id_1, \ldots, id_n)} \tag{9}$$

6.4 The *fields* Helper Function

We use the notation $fields(C)$ to denote a type environment constructed from the fields of C and the fields of the superclasses of C. The fields in C take precedence over the fields in the superclasses of C. The definition of *fields* is:

$$\frac{\texttt{class}\ id\ \{\ t_1\ id_1;\ \ldots;\ t_f\ id_f;\ m_1\ \ldots\ m_k\ \}\ \text{ is in the program}}{fields(id) = [id_1 : t_1, \ldots, id_f : t_f]} \tag{10}$$

$$\frac{\texttt{class}\ id\ \texttt{extends}\ id^P\ \{\ t_1\ id_1;\ \ldots;\ t_f\ id_f;\ m_1\ \ldots\ m_k\ \}\ \text{ is in the program}}{fields(id) = fields(id^P) \cdot [id_1 : t_1, \ldots, id_f : t_f]} \tag{11}$$

6.5 The *methodtype* Helper Function

We use the notation $methodtype(id, id^M)$ to denote the list of argument types of the method with name id^M in class id (or a superclass of id) together with the return type (or $\perp$ if no such method exists). The result of *methodtype* is of the form $(t_1, \ldots, t_n) \to t$, or $\perp$. The definition of *methodtype* is:

$$\frac{\begin{array}{l}\texttt{class}\ id\ \{\ \ldots\ m_1\ \ldots\ m_k\ \}\ \text{ is in the program} \\ \text{for some } j \in 1..k : methodname(m_j) = id^M \\ m_j \text{ is of the form} \\ \quad \texttt{public}\ t\ id^M\ (t_1^F\ id_1^F, \ \ldots, \ t_n^F\ id_n^F)\ \{t_1\ id_1; \ \ldots; \ t_r\ id_r; \ s_1\ \ldots\ s_q\ \texttt{return}\ e; \ \}\end{array}}{methodtype(id, id^M) = (id_1^F : t_1^F, \ldots, id_n^F : t_n^F) \to t} \quad (12)$$

$$\frac{\begin{array}{l}\texttt{class}\ id\ \{\ \ldots\ m_1\ \ldots\ m_k\ \}\ \text{ is in the program} \\ \text{for all } j \in 1..k : methodname(m_j) \neq id^M\end{array}}{methodtype(id, id^M) = \perp} \quad (13)$$

$$\frac{\begin{array}{l}\texttt{class}\ id\ \texttt{extends}\ id^P\ \{\ \ldots\ m_1\ \ldots\ m_k\ \}\ \text{ is in the program} \\ \text{for some } j \in 1..k : methodname(m_j) = id^M \\ m_j \text{ is of the form} \\ \quad \texttt{public}\ t\ id^M\ (t_1^F\ id_1^F, \ \ldots, \ t_n^F\ id_n^F)\ \{t_1\ id_1; \ \ldots; \ t_r\ id_r; \ s_1\ \ldots\ s_q\ \texttt{return}\ e; \ \}\end{array}}{methodtype(id, id^M) = (id_1^F : t_1^F, \ldots, id_n^F : t_n^F) \to t} \quad (14)$$

$$\frac{\begin{array}{l}\texttt{class}\ id\ \texttt{extends}\ id^P\ \{\ \ldots\ m_1\ \ldots\ m_k\ \}\ \text{ is in the program} \\ \text{for all } j \in 1..k : methodname(m_j) \neq id^M\end{array}}{methodtype(id, id^M) = methodtype(id^P, id^M)} \quad (15)$$

6.6 The *noOverloading* Helper function

$$\frac{methodtype(id^P, id^M) \neq \perp \ \Rightarrow \ methodtype(id^P, id^M) = methodtype(id, id^M)}{noOverloading(id, id^P, id^M)} \quad (16)$$

The *checkDistinctIfSame* **Helper function**
takes two lists L1 and L2 of same length as arguments and checks that for any two indices i, j(i != j) in the range [0, length of L1-1] if L1[i] = L1[j] then L2[i] != L2[j]

The *methodAccess* **Helper function**
methodAccess(D, id) returns the access modifier of the method id in class D or one of its super-classes

The *methodType* **Helper function**
methodType(D, id) returns the type of the method id in class D or one of its super-classes

The *variableAccess* **Helper function**
variableAccess(D, id) returns the access modifier of the variable id in class D or one of its super-classes

The *currentFunction* **Helper function**
currentFunction() returns the name of the function in which the current execution of the program is taking place

The *isFriendClass* **Helper function**
isFriendClass(D, C, id) returns true if C is declared as a friend class in D or any of its super classes say E and if id is declared in E or any of its super classes

The *isFriendFunction* **Helper function**
isFriendFunction(D, F, id) returns true if F is declared as a friend function in class D or any of its super classes say E and if id is declared in E or any of its super classes

5.1.2 Goal

$$\frac{\text{distinct}\ (\text{classname}(mc), \text{classname}(d_1), \ldots, \text{classname}(d_n)) \qquad \vdash mc \qquad \vdash d_i \quad i \in 1..n}{\vdash mc \quad d_1 \ldots d_n}$$

5.1.3 Main Class

$$\frac{distinct\,(id_1,\ldots,id_r) \quad [id_1:t_1,\ldots,id_r:t_r],\bot\vdash s_i \quad i\in 1..q \quad \bot\vdash a_i \;\; i\in 1..n}{\vdash \text{ class } id\{ \text{ public static void main (String [ ]}id^S)\,\{a_1t_1id_1;\ldots;a_rt_rid_r;s_1\ldots s_q\}\}}$$

5.1.4 Type Declarations

$$\frac{\begin{array}{c}\text{distinct}\,(id_1,\ldots,id_f) \qquad distinct(id1_1,\ldots,id1_p)\\ distinct(methodname\,(m_1),\ldots,methodname\,(m_k))\\ checkDistinctIfSame \;\; ([id2_1,\ldots,id2_p],[id3_1,\ldots,id3_q])\\ id\vdash m_i \quad i\in 1..k \qquad id\vdash a_i \quad i\in 1..f\end{array}}{\begin{array}{c}class \;\; id\{a_1t_1id_1;\ldots;a_ft_fid_f;m_1\ldots m_kfriend \;\; class \;\; id_1;\ldots;friend \;\; class \;\; id_p\\ friend \;\; function \;\; id_1;\ldots friend \;\; function \;\; id_q;\}\end{array}}$$

$$\frac{\begin{array}{c}\text{distinct}\,(id_1,\ldots,id_f) \qquad distinct(id1_1,\ldots,id1_p)\\ distinct(methodname\,(m_1),\ldots,methodname\,(m_k))\\ checkDistinctIfSame \;\; ([id2_1,\ldots,id2_p],[id3_1,\ldots,id3_q])\\ noOverloading\left(id,id^P methodname\,(m_i)\right) \qquad id\vdash m_i \quad i\in 1..k \qquad id\vdash a_i \quad \in 1..f\end{array}}{\begin{array}{c}class \;\; id\{a_1t_1id_1;\ldots;a_ft_fid_f;m_1\ldots m_kfriend \;\; class \;\; id_1;\ldots;friend \;\; class \;\; id_p\\ friend \;\; function \;\; id_1;\ldots friend \;\; function \;\; id_q;\}\end{array}}$$

5.1.5 Typechecking Method Calls

$$\frac{\begin{array}{c}A,C\vdash p:D \quad methodType(D,id)=(t'_1,\ldots,t'_n)\longrightarrow t\\ (methodAccess(D,id)=public\vee(methodAccess(D,id)=protected\wedge C\leq D)\\ \vee(methodAccess(D,id)=private\wedge C=D) \qquad A,C\vdash e_i:t_i \quad t_i\leq t'_i \quad i\in 1,\ldots,n\end{array}}{A,C\vdash p.id(e_1,e_2,\ldots,e_n):t}$$

[1]

5.1.6 Method Declarations

$$\frac{\begin{array}{c}a=\text{"}public\text{"}|\text{"}private\text{"}|epsilon\\ distinct(id_1^F,\ldots,id_n^F)\\ distinct(id_1,\ldots,id_r)\\ A=fields(C)\cdot[id_1^F:t_1^F,\ldots,id_n^F:t_n^F]\cdot[id_1:t_1,\ldots,id_r:t_r]\\ A,C\vdash s_i \quad i\in 1..q \quad A,C\vdash e:t\end{array}}{C\vdash a \;\; t \;\; id^M(t_1^Fid_1^F,\ldots,t_n^Fid_n^F)\;\{t_1id_1;\ldots;t_rid_r;s1,\ldots,sq \;\; return\; e;\}}$$

5.1.7 Statements

$$\frac{A,C\vdash s_i \quad i\in 1..q}{A,C\vdash s_1\ldots s_q}$$

$$\frac{A(id)=t_1 \quad A,C\vdash e:t_2 \quad t_2\leq t_1}{A,C\vdash id=e;}$$

$$\frac{A(id)=int[] \quad A,C\vdash e_1:int \quad A,C\vdash e_2:int}{A,C\vdash id[e1]=e2}$$

$$\frac{A(id1.id)=t_1 \quad A,C\vdash e:t_2 \quad t_2\leq t_1}{A,C\vdash id1.id=e;}$$

$$\frac{A(id1.id)=int[] \quad A,C\vdash e_1:int \quad A,C\vdash e_2:int}{A,C\vdash id1.id[e1]=e2}$$

$$\frac{A,C\vdash e:boolean \quad A,C\vdash s_1 \quad A,C\vdash s_2}{A,C\vdash if(e)s_1elses_2}$$

$$\frac{A,C\vdash e:boolean \quad A,C\vdash s}{A,C\vdash while(e) \;\; s}$$

$$\frac{A,C\vdash e:int}{A,C\vdash System.out.println(e);}$$

[1] t1 $\leq$ t2 if t1 is a subtype of t2

5.1.8 Expressions and Primary Expressions

Along with the type rules defined for MiniJava, the following are the additions to be made to typecheck expressions and primary expressions

$$\frac{\begin{array}{c}A, C \vdash id1 : D \quad variableAccess(D, id2) == "public" \vee isFriendClass(D, C, id2) \\ \vee \ (A, C \vdash currentFunction() = F \wedge isFriendFunction(D, F, id2)) \quad A, D \vdash id2 : t\end{array}}{A, C \vdash id1.id2 : t}$$

$$A, C \vdash a : public$$

$$A, C \vdash a : private$$

5.2 Operational Semantics

Same as defined for MiniJava, by adding the new construct the semantics are not going to change

6 Prior Work

C++ supports this feature friend class and friend function which is used to break the encapsulation as per the user's wish. This feature is present exclusively in **C++**. No other object oriented language supports this feature.Similar to Java there is no option of friend function and friend class in python.

7 Implementation

7.1 Plan

Modifications are to be done in the syntax tree of the MiniJava Language. Nodes like friend declaration ,friend class ,friend function ,Accesstype and some more should be added to the syntax tree directory. The definition of the classes should be written following the above type rules and MiniJava operational semantics

7.2 Challenges

This implementation involves creation of new classes and changing the syntax tree directory and some classes in the visitor directory which are to be written carefully. The creation of symbol table takes place differently when a class or a method is declared to be a friend of another class granting them access to the class's private members