ECE 479/579

Principles of Artificial Intelligence

Dr. Marefat

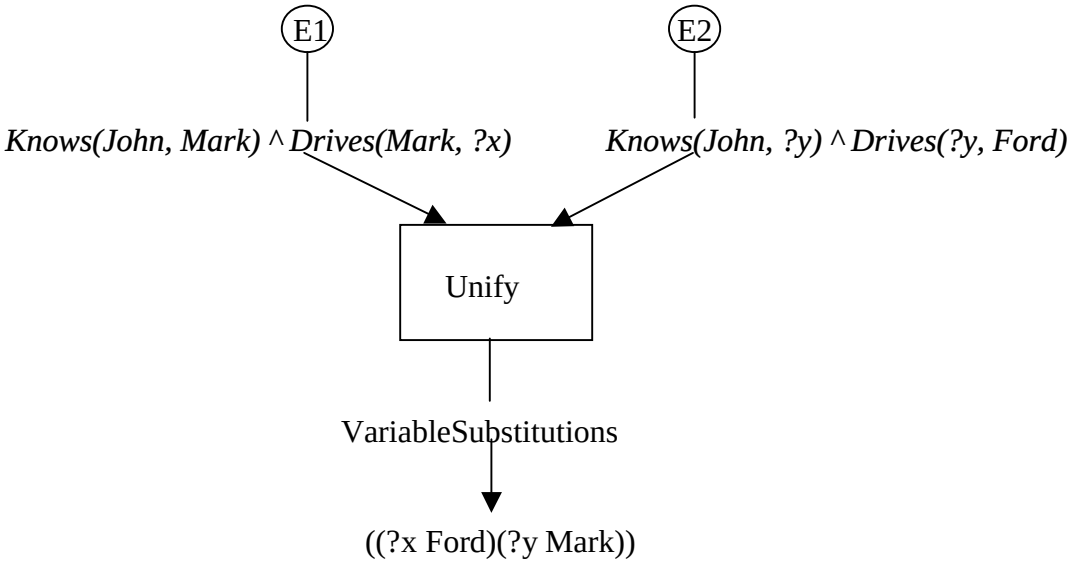
PROJECT # 1

“AI Programming”

PART I (compulsory for all students) This project requires you to implement a basic version of the unify procedure discussed in class in JAVA (a set of classes have been provided to you to help standardize coding and all students are expected to make use of them). You have to follow the mentioned specifications for writing your solution:

a. Both E1 and E2 are each an array of objects. Each object of the array is an instance of the `AtomicExpression` class. E1/E2 are assumed to be in CNF (Conjunctive Normal Form), where each object of the array is a conjunct.
b. E1 and E2 don't have functions inside them. Each object/conjunct consists of an object of type `Predicate` and a `List` of elements (in proper order), each item of which needs to be either an object of type `Variable` or of type `Constant` only.
c. You are supposed to write a class `Unify`, which contains a `'public static'` function `unify(final AtomicExpression[] e1, final AtomicExpression[] e2)`, which returns an object of type `VariableSubstitutions`, which contains the most general unifier (MGU), for the two expressions.

All the source files (*.java) of the standard classes has been provided, along with their java documentation (in HTML). A `aiProgramming.jar` (JAVA archive) file has also been provided.



Example: to see if *Knows(John, Mark) ^ Drives(Mark, ?x)* unifies with *Knows(John, ?y) ^ Drives(?y, Ford).* Then:

E1 = *Knows(John, Mark) ^ Drives(Mark, ?x)*
E2 = *Knows(John, ?y) ^ Drives(?y, Ford)*

In this example, if your program runs correctly, it will produce the following as the necessary substitution.

((?x Ford)(?y Mark))

E1, E2, substitutions etc. are really all JAVA objects and the above has been shown just as an illustration of the functionality.

PART II - Graduate requirement (Extra credit for undergraduate students)

A knowledge base (KB) is a set of facts (in our model an `AtomicExpression` object) and rules (in our model an `ImplicationExpression` object). The KB can be represented by an object of type `KnowledgeBase.` Hence you need to turn in three files:

- **ImplicationExpression.java à** The constructor for this class should take in 2 arrays. Both the arrays contain `AtomicExpression` objects, assumed to be in CNF. The first array is the premise/antecedent and the second one is the conclusion/consequent. The internal working of this class is up to you.
- **KnowledgeBase.java à** This class needs to implement 2 functions:
  - **`public void addFact(final AtomicExpression e)`** – This adds a fact to the KB.
  - **`public void addRule(final ImplicationExpression ie)`** – This adds a rule to the KB.
  - **`public AtomicExpression[] infer()`** – This will perform forward chaining and return back an array of atomic expressions. Each of the `AtomicExpression` objects in the array represents a new fact which has been inferred. ANY new fact which was not explicitly added by the user to the KB needs to be returned to get full credit.
- **FCTester.java à** This is a test class needs to be made to test whether your classes have been correctly coded. This class should not need any command line argument and should have a `public static void main(String[] args)` function. You need to implement the example from section 9.3 (Page 280) from the course book. Add the facts and rules as mentioned on page 281. Your `infer()` function call on the KB should be able to infer the following facts (as can be seen in Figure 9.4 on Page 282):
  - *Weapon(M_1)*
  - *Sells(West, M_1 , Nono)*
  - *Hostile(Nono)*
  - *Criminal(West)*

  Using the `toString()` function of the `AtomicExpression` class print out all the newly found facts on the console.

Students are referred to relevant portions of Chapter 8 and 9 of the course book, for this part of the assignment.

Specifications **(for graduate requirement/extra credit, specifications have been mentioned above)**

Your program is required to adhere to the specifications listed here or it cannot be graded.

PROGRAM NAME: The program names for Part I of the assignment should be "Unify.java". Please comment your code neatly and if possible, provide `javadoc` comments for all class variables and functions of your classes.

COMPILATION PROCEDURE: Your programs will be tested with multiple test cases on shell.ece.arizona.edu. Please make sure that your programs compile correctly, making use of the provided jar file (in this case the command will be "`javac –classpath .:./aiProgramming.jar` <YourFile.java>") or by making use of the source code itself (javac *.java). The latter case assumes that all the java files are in the same directory. Package names in Java translate to a similar/analogous directory structure. Hence you would need to make a directory structure similar to ~/ece479/projects/aiProgramming/<your classes>.

The package name for the classes provided is "`ece479.projects.aiProgramming`", and hence the import statements (if any needed) in your program will be something like "`import ece479.projects.aiProgramming.Constant`".

INPUT: A test class would be needed to check whether your unification is working properly or not. A sample class `UnifyTester` with a `main()` function has been provided to you. The relevant contents of the same are mentioned below. It tells the basic procedure of making the expressions and calling the `unify()` function:

```
// constants
Constant fred = new Constant("Fred");
Constant engineer = new Constant("Engineer");
Constant tucson = new Constant("Tucson");

// variables
Variable x = new Variable("x");
Variable y = new Variable("y");
Variable z = new Variable("z");

// list of elements
ArrayList l1 = new ArrayList();
l1.add(fred);
```

```
l1.add(engineer);
l1.add(tucson);

ArrayList l2 = new ArrayList();
l2.add(x);
l2.add(y);
l2.add(z);

// Predicate
Predicate personPredicate = new Predicate("Person", 3);

// expressions
AtomicExpression[] e1 = new AtomicExpression[1];
AtomicExpression[] e2 = new AtomicExpression[1];
e1[0] = new AtomicExpression(personPredicate, l1);
e2[0] = new AtomicExpression(personPredicate, l2);

// Unify expressions!!
VariableSubstitutions vs = Unify.unify(e1, e2);
```

Before turning in your code please test your program with additional test cases which have been provided on the class website, by clicking on "Testing your code" link (Also see Appendix A).

OUTPUT: Once the `unify()` function has been called, the user can go over all the possible variable substitutions and using the `toString()` function print them on the console. Below is the code from "UnifyTester.java":

```
// Unify expressions!!
VariableSubstitutions vs = Unify.unify(e1, e2);

// Print out the subsitutions (if any have been found!)
if (vs == null) {
     System.out.println("\nNo subsitutions were
found\n");
} else {
     System.out.println("\nThe substitutions are: \n" +
vs.toString()
                        + "\n\n");
}
```

TURNIN

All program files need to be turned in as follows on shell.ece.arizona.edu:

```
turnin ece479 proj1 Unify.java <otherExtraCreditFiles.java>
```

Appendix A
Testing your code

Find below some java files which you can use for testing your program. They are similar in format and usage to the UnifyTester.java which has also been provided. The specific test cases that each file checks are also mentioned below:

Test 1 - `UnifyTester1.java`

Tests for the unification of the following:
Brother(?x, Adam, ?y)
Brother(?y, Adam, ?x)

Test2 - `UnifyTester2.java`

Tests for the unification of the following:
Brother(?x, Adam, ?y) ^ Brother(Bruce, Adam, ?y)
Brother(Bruce, Adam, ?z) ^ Brother(Jim, Adam, ?z)

Test3 - `UnifyTester3.java`

Tests for the unification of the following:
Person(?x, ?y) ^ Brother(?x, Adam) ^ Person(Jim, ?z)
Brother(Bruce, Adam) ^ Person(Jim, Sam) ^ Person(Bruce, Jim)

Grad / Extra Credit tests

Test1 - `FCTester1.java`

The following is tested for:
// facts
Grad(Adam)
Grad(Jim)

// rules
Grad(?x) => Brainy(?x)

Test2 - `FCTester2.java`

The following is tested for:
// facts

Grad(Adam)
Friends(Adam, Jim)

// rules
Grad(?x) => Brainy(?x)
Brainy(?x) ^ Friends(?x, ?y) => Brainy(?y)

Appendix B
Frequently Asked Questions (FAQ)

Q. Could you give some examples for the results expected out of the unification of two AtomicExpression arrays ?

A. Example 1. Does this unify?
E1 = Brother(?x, Dan) ^ Brother(Joe, Dave)
E2 = Brother(Joe, Dave) ^ Brother(Stan, Dan)

Yes it does. ?x/Stan is the substitution. A common question which could arise:

Brother(?x, Dan) does not unify with Brother(Joe, Dave), which is the first check I would make. I would in this case fail the unification, even though Brother(?x, Dan) unifies with Brother(Stan, Dan) and Brother(Joe, Dave) unifies with Brother(Joe, Dave). So the real question is, if ANY part of the unification of E1 and E2 fails, does the entire unification of E1 with E2 fail?

No. You surely have to consider all options/possible combinations, since E1 and E2 may have multiple instances of a predicate in them. Hence EVERY unique predicate (of which there is only one, Brother, in this example) has to fail before the entire unification is considered to fail.

Example 2. Does this unify?
E1 = Brother(?x, Dan) ^ Brother(Joe, Dave) ^ Sister(Jane, Martha)
E2 = Brother(Joe, Dave) ^ Brother(Stan, Dan) ^ Sister(Kylie, Jen)

No.

Example 3. Does this unify?
E1 = Brother(?x, Dan) ^ Brother(Trevor, Sumit)
E2 = Brother(Stan, Dan) ^ Brother(Mike, Ted)

No.

Q. I Can't seem to get the source code provided for this project to compile ?

A. Putting all the files in a directory structure like <some directory>/ece479/projects/aiProgramming/ will do the trick. Then simply do "javac ./ece479/projecs/aiProgramming/*.java" or even "javac *.java" if you are in the aiProgrammming directory. As expected Unify.java won't compile since thats the file you need to make. Simply put a return statement (for the time being to get everything to compile)

like "return new VariableSubstitutions();". Now from your "<some directory>" call "java ece479.projects.aiProgramming.UnifyTester" to run the program.

Q. Can I change the source code for the JAVA files provided for this project ?

A. No! Please do not make any changes to the provided JAVA files. This is because we will be using a standard set of files for all students while grading. If you make changes (add functions, variables etc.) which suit the logic of your program then it will complicate grading. On the other hand if you feel that some functionality is missing in the code we have given you (and it prevents you/takes extra time to code to solution), then please send an email to ece479@ece.arizona.edu about it.

Q. How do I use the provided aiProgramming.jar file ?

A. This archive does not contain the Unify and UnifyTester classes for obvious reasons. Hence these 2 files need to be downloaded and compiled separately. To make things simple you may remove the package statements from these files effectively making them package less (otherwise you will need the directory structure similar to that mentioned above). To get things to compile simply call "javac -classpath .:./aiProgramming.jar *.java".

Q. What do I need to turnin ?

A. Undergraduate students need to just turnin the completed "Unify.java" file. For graduate students there are some other java files. Please see the project specifications [PDF]. If for whatever reason you plan to turnin extra files, then please send an email explaining the same to ece479@ece.arizona.edu. There is no need to turnin the code which has been provided to you (since we already have it!!).

Q. Can I turnin Unify.java with an arbitrary package name ?

A. No! Either use the package name ece479.projects.aiProgramming or simply remove that line (making it packageless). Using arbitrary package names will complicate grading.

Q. How do I get the project configured in Eclipse ?

A. We cannot help students get the project configured in Eclipse. That said it is extremely easy and you may want to refer to some helpful links put up on the Class Software section of the class website.

Q. Can you provide some more coding hints/tips on getting this project done !?

A. Please take a special note of the overloading of "equals()" in all the classes. This should save you some time when making comparisons (not that the "toString()" functions are just for printing stuff to the console). The "instanceof" comparator will be useful too to find out whether something (an object) is say a variable or constant or predicate etc. Also the "getPredicate()" and "getElements()" functions in the AtomicExpression class should let you tear an atomic expression apart and implement the unify procedure. Also as you can see in the code it is perfectly legal to put a variable-variable mapping in the VariableSubstitutions class (using the "put()" function). A variable-constant mapping is also legal. For obvious reasons a constant-constant mapping is not allowed.

Q. I see a bug in the code provided. What should I do ?

A. Shoot an email to ece479@ece.arizona.edu so that the bug can be resolved.