

Milestone 7: Unification

| 10/18/2022

100 Possible Points

 Add Comment

▼ Details

The purpose of this milestone is to implement the core of the logical resolution process used to answer queries: the unification algorithm.

Unification in First-Order Logic

Over the next three milestones we will be implementing the ability make queries against a knowledge-base by implementing the `ask` member of the `KnowledgeBase` class. An important step in the process will be to find a set of substitutions for variables that will make two expressions equivalent. This process is called *unification*.

For example consider two vt-prolog expressions `f(a)` and `f(X)` as expression trees. Unification will find an assignment of the variable `X` that will make the two trees (and thus expressions) identical. This is obviously `X/a` in this simple example, where the slash is read as “substituted by”, i.e. “X substituted by a” or `f(a)` and `f(X)` unify under the substitution `X/a`.

Substitutions can replace whole subtrees as well. For example, trying to unify `f(g(a),b)` and `f(X,b)` succeeds under the substitution `X/g(a)`, i.e. the variable `X` must be replaced by the tree `g(a)` to make the two expressions equivalent.

In some cases, when there are no variables, unification acts like a test for equality. For example `f(a,g(b,c))` and `f(a,g(b,c))` unify under the empty substitution. Unification can also fail, for example by trying to unify `f(a,b)` and `f(X,c)`.

Finally, when we have multiple expressions, as in the clauses of a knowledge-base, we might have multiple valid substitutions. For example given the simple knowledge-base consisting of just facts:

```
likes(mary, pizza).  
likes(bob, pizza).  
likes(sue, soccer).
```

The query `likes(X, pizza)` should unify with two of the three expressions, giving the substitution `X/{mary,bob}`. This is more complicated in the case of rules, which we postpone to milestone 9.

Thus we see that we need to next implement two new pieces of code: a data structure to hold (possibly multiple) substitutions, and an algorithm to perform unification. That is the focus of this

milestone, which will introduce a new module: `Unification.hpp`, and `Unification.cpp` with associated unit tests `UnificationTests.cpp`.

Substitution Set

The data structure for holding substitutions needs to be able to map from expression trees, to multiple, arbitrary trees as fast as possible. We will need to be able to insert a substitution, lookup substitutions, and iterate through the set of substitutions. You should choose an appropriate container from the standard library and typedef it to the type `SubstitutionData`.

You should then implement, in the namespace `vtpl`, the class `Substitution` with the following interface:

```
class Substitution {
public:

    typedef typename SubstitutionData::iterator IteratorType;
    typedef typename SubstitutionData::const_iterator ConstIteratorType;

    // lookup an expression key in the substitution set and return a list of Expressions
    // the key maps to, or a list of size zero if no mapping exists
    std::list<ExpressionTreeNode> lookup(const ExpressionTreeNode& key) const;

    // insert a mapping from Expression key to Expression value, appending it if a mapping already exists.
    void insert(const ExpressionTreeNode & key, const ExpressionTreeNode & value);

    // return an iterator to the first element of the arbitrarily ordered set
    IteratorType begin();

    // return an iterator to one past the last element of the arbitrarily ordered set
    IteratorType end();

    // return a const iterator to the first element of the arbitrarily ordered set
    ConstIteratorType constBegin() const;

    // return a const iterator to one past the last element of the arbitrarily ordered set
    ConstIteratorType constEnd() const;

private:

    // TODO
};
```

Unification Algorithm

The unification algorithm returns a result consisting of a Boolean flag indicating if unification succeeded and if true the associated substitution. We can easily define a type to hold this as follows:

```
struct UnificationResult{

    bool failed;
    Substitution substitution;

};
```

The following function should take two expression trees and an initial substitution set, and attempt to unify them, modifying the passed `UnificationResult`:

```
void unify(const ExpressionTreeNode & x, const ExpressionTreeNode & y, UnificationResult & subst);
```

The algorithm does a recursive descent search of the two trees, comparing nodes and inserting substitutions as needed. It can be described in pseudo-code as follows:

```
function unify(x,y,s) return a substitution to make x and y the same
inputs:
  * x a variable, constant, list, or compound expression
  * y a variable, constant, list, or compound expression
  * s the substitution list build up so far

if s == failure then return failure
else if x = y then return s
else if isvariable(x) then return unify-var(x,y,s)
else if isvariable(y) then return unify-var(y,x,s)
else if iscompound(x) and ifcompound(y) then
  return unify(x.args, y.args, unify(x.op, y.op))
else if islist(x) and islist(y) then
  return unify(x.rest, y.rest, unify(x.first, y.first, s))
else return failure
```

where `op` refers to the predicate name of a compound, and `args` the argument list of a compound.

The helper function `unify-var`, to unify variables and make the substitution, is given by:

```
function unify-var(var, x, s) returns substitution
inputs:
  * var is a variable
  * x a variable, constant, list, or compound expression
  * s is a substitution

if var/val in s then return unify(val, x, s)
else if x/val in s then return unify(var, val, s)
else return add var/x to s
```

where `x/y` means `y` is assigned to `x` in the substitution.

You should write out a few expression trees and manually go through the algorithm to ensure you understand how it works.

Instructions

Your task in this milestone is to finish the `Unification` module for VT-Prolog as described above.

- `Unification.hpp` should define the structure `UnificationResult` and the function `unify`, described above, in the namespace `vtpl`.
- The `unify` function should be implemented in the file `Unification.cpp`.
- The file `UnificationTests.cpp` should contain Catch based unit tests for your Unification module.

As in previous milestones, you can define additional units of code (classes, helper functions) in any implementation (.cpp) file, but do not change the public interface of any headers (.hpp) beyond that specified.

Steps to build and run the tests in the reference environment (after vagrant ssh).

Note: you should be working primarily on your host system for development. These steps are just to check the code in the reference environment, as well as test code coverage and memory safety.

- To configure the build

```
cmake /vagrant
```

- To run the build

```
make
```

or

```
cmake --build .
```

- To run the unit tests

```
make test
```

or

```
cmake --build . --target test
```

The test output is placed in ``Testing/Temporary/LastTest.log. You can also just run the tests directly:

```
./unit_tests
```

- To run the memory checks on your unit tests.

```
cmake -DMEMTEST=TRUE /vagrant
```

followed by

```
make memtest
```

or

```
cmake --build . --target memtest
```

This should report no “definitely lost” leaks and no access errors. Look for

```
LEAK SUMMARY:  
  definitely lost: 0 bytes in 0 blocks
```

or

```
All heap blocks were freed -- no leaks are possible
```

and

```
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

- To run your tests and check code coverage.

```
cmake -DCOVERAGE=TRUE /vagrant
```

followed by

```
make coverage
```

or

```
cmake --build . --target coverage
```

This will create a directory called `Coverage_Report` in the build directory. You can copy it back to your host by doing `cp -r Coverage_Report /vagrant`. Open the `index.html` file inside it to examine the detailed report.

- To configure and run the build in strict mode (increased warnings, warnings become errors)

```
cmake -DSTRICT=True /vagrant  
cmake --build . --target clean  
cmake --build .
```

Note: you can also combine all the options into one build configuration, i.e.

```
cmake -DSTRICT=True -DCOVERAGE=TRUE -DMEMTEST=TRUE /vagrant  
cmake --build .  
cmake --build . --target test  
cmake --build . --target memtest  
cmake --build . --target coverage
```

Submission

You must add changes to your repository and push them to GitHub regularly to demonstrate progress. When you are ready to submit your assignment you must:

1. Tag the git commit that you wish to be considered for grading as “m7”.

```
git tag m7
```

or

```
git tag m7 COMMIT
```

where COMMIT is the commit id you wish to tag.

2. Push this change to GitHub

```
git push origin m7
```

If you need to tag a different version of your code simply create and push a new tag appending a full stop followed by a monotonically increasing number, e.g. m7.1, m7.2, etc.

Be sure you have committed all the changes you intend to. Before the milestone due date it is a good idea to re-clone your repository into a separate directory and double check it is what you intend to submit. **Failure to complete these steps by the due date will result in a failed submission.**

Milestone Assessment

Milestone 7 will be assessed according to the following criteria.

Your code compiles in the grader/reference environment	1 point
Your tests pass	1 points
Your tests show no memory errors	1 points
Your code coverage	5 points
Your code compiles in strict mode	1 points
Your code compiles with instructor tests in strict mode	1 points
Your code passes instructor tests	20 points
Instructor tests show no memory errors	5 points

Note, if your code does not build in the reference environment you will receive **no points**. Correctness is determined by the proportion of instructor unit tests that pass. Code quality will be assessed in this assignment by ensuring your code compiles cleanly, with no warnings, using the strict mode specified above, there are no memory errors (leaks, use before initialize, etc), and proportion of your tests that cover your code (code coverage), which should exceed 98% of lines.

Your milestone can be **checked in the auto-grader (<https://grader.ece.vt.edu>)**. The auto-grader uses the exact same environment as the reference so if your code compiles there, it should in the auto-grader as well (but check anyway!). You are rate-limited to only four submissions every hour to the auto-grader

so as to prevent you from using it as your development environment and encourage proper debugging skills.