CS396 Assignment 3: Baby’s First Inference Engine

Overview

In this assignment, you’ll build a highly simplified version of Step, also known as a “Horn clause theorem prover.” The bad news is that this uses the same explicit success continuation trick used in the parser assignment. The good news is also that it uses the same explicit success continuation trick used in the parser assignment. In fact, the algorithm here is almost identical to the parser algorithm even though it’s solving a completely different problem. The main thing that’s new is that you’ll have to implement unification, but we’ll walk you through that.

Example, in somewhat plain English

In class, we talked about an example of Step code where we have a predicate, IsA, that says what kind of monster different students in the Monster High School were. The lines defining IsA form a kind of a database of students and we can query that database by calling the IsA predicate: it tries to match (unify) its arguments against the values shown in each IsA rule until it finds one that matches. Calls can include variables that can be matched to any value, and when it finds a match, it reports the value of that variable back to you. That gives you a way of asking:

- “is Jayden a ghost?” – `[IsA jayden ghost]`
- “what kind of a monster is Jayden?” – `[IsA jaden ?what]`, to which the answer is `?what = ghost`
- “who is a ghost?” – `[IsA ?who ghost]`, to which one answer is `?who = Jayden`

We don’t want you to have to write a parser for the Step language for this assignment, so you’ll write the code in ersatz Step, written as a set of C# calls. The definition of IsA might look like this:

```
var IsA = new Predicate(“IsA”);     // Make a new Predicate called IsA
IsA[“aniyah”, “human”].Fact();      // Add rule that Aniyah is a human
IsA[“tiana”, “vampire”].Fact();     // Add rule that Tiana is a vampire
IsA[“cameron”, “werewolf”].Fact();  // Cameron is a werewolf
IsA[“david”, “witch”].Fact();       // David is a witch
IsA[“jayden", “ghost”].Fact();      // Jayden is a ghost
… etc …
```

Then we could query it by saying:

```
Prover.CanProve(IsA[“jayden", “ghost”])
```

Which will try to prove Jayden is a ghost and return true if it can. In AI terminology, `IsA[“jayden", “ghost”])` is a **goal** it’s trying to prove. `CanProve()` takes a goal to prove and tries to prove it’s true given the statements we’ve given it (the `Fact()` calls, above).

It tries to prove it by going through each rule for IsA and compares its arguments to the arguments of the goal. In this case, it matches one of the rules exactly, at which point the system is done and returns true.

If we want to ask "are there any werewolves?" we can say:

```
var who = new Variable("?who");
Prover.CanProve(IsA[who, "werewolf"])
```

`CanProve` calls a method, `Prover.Prove`, which we'll talk about shortly, which does the real work. `Prove` goes through each rule for IsA, trying to match it to the goal `IsA[who, "werewolf"]`. When it finds one, it reports success by calling a success continuation and passing it the value of the variable `who`, in this case `"cameron"`. It then returns to `CanProve`, which returns true. While `CanProve` ignores the value of the variable, there's a variant of it, `SolveFor`, that will return the value of the variable:

```
var who = new Variable("?who");
Prover.SolveFor(who, IsA[who, "werewolf"])
```

this will return the string `"cameron"`, since that was the value for `?who` in the first solution it found for the call.

Subgoaling

We can do more complicated rules, which have subgoals to run. For example, we can write the equivalent of the `Student` rule we wrote in Step in class:

```
var Student = new Predicate("Student");
// ?who is a student if ?who IsA ? for some monster ?
Student["?who"].If(IsA["?who", "?"]);
```

Then we make a new predicate, `Student`, and tell it that when trying to prove `?who` is a student, we should try to prove `IsA[?who, ?]`, i.e. that `?who` is some kind of monster (but we don't care what kind).

> **Note:** in the first example, we make a `Variable` object and used that as the argument to `IsA` to distinguish it from a regular string. However, we won't bother to do that in rules; we'll just type the rule with all strings. If we mean the rule to have a variable we type the string "?name" and the system will assume that strings that look like variable names are intended to be variables. This saves us a lot of typing "`var whatever = new Variable("whatever");`" and it isn't any less efficient because it's going to turn out that every time we try to run a rule we're going to have to make a fresh copy with new Variable objects anyway. So it's worth it, even if it's slightly more confusing. You might ask why we don't do that for queries (things we try to prove) too, but the answer is that there can end up being lots of different variables with the same name. It turns out we have to use the variable object to make sure we get the right one.

When we call:

```
Prover.SolveFor(who, Student[who])
```

It starts by trying to prove `Student[who]`. So it calls Prove. `Prove` goes through each rule for `Student`, but there' only one. It says that to prove `Student[who]`, you have to prove `IsA[who, v]`, where v is another variable. So `Prove` calls itself recursively on `IsA[who, v]`, repeating the process described above. When it succeeds, it then reports the values of both `who` and `v` to its continuation. At that point, it's proven `IsA[who, v]` and so by the rule, it's also proven `Student[who]`, so we're done.

Now let's consider a more complicated one and go into a little more detail. We make three predicates, p, q, and r. And we say:

```
p["a"].Fact();
p["b"].Fact();

q["c"].Fact();
q["b"].Fact();

r["?x"].If(p["?x"], q["?x"]);
```

That is:

- p["a"], p["b"], q["c"] and q["d"] are true
- r[?x] is true whenever both p[?x] and q[?x] are true

If we have the goal r[?y], that is, "find a ?y for which r[?y] is true", then the system does the following:

- It calls Prove to try to **prove r[?y]**
    - It tries all the rules for r, but there's **only one rule**: r[?x] is true if both p[?x] and q[?x] are true. So it tries this rule.
    - It compares its goal r[?y] to the rule's conclusion, r[?x], and remembers that ?y in the goal is "really" ?x in the rule.
    - It knows that in order to prove the conclusion of this rule (r[?y]) it should try proving both of the **subgoals p[?x] and q[?x].**
        - So it **calls Prove on p[?x]**
            - Prove iterates over the rules for p[?x]
                - It tries the rule **p["a"]**
                    - So it assumes that ?x="a"
                    - And goes on to try the next subgoal: q[?x], but since ?x is assumed to be "a", it knows to try to prove q["a"]
                        - It calls Prove on q["a"]
                            - It tries all the rules for q and neither lets it prove q["a"]
                            - So **Prove fails on q["a"]**
                            - **We backtrack**
                    - Since the solution **p["a"] was rejected**, it gives up on the assumption that ?x="a"
                    - We backtrack and try the next rule for p
                - It tries the rule **p["b"]**
                    - So it assumes that ?x="b"
                    - And goes on to try the next subgoal: q[?x], but since ?x is assumed to be "b", it knows to try to prove q["b"]
                        - It calls Prove on q["b"]
                        - We have a rule for that: q["b"] is a fact
                - So prove succeeds for q["b"]
            - And so prove succeeds for p["b"]
    - And so the rule that r[?x] whenever p[?x] and q[?x] succeeds with the stipulation that ?x="b"
- Our original goal was r[?y], and we'd determined that ?y=?x, so we end up concluding that ?y="b"

Variable substitutions

We did a lot of hand waving about “we remember this variable is really that”. Let’s explain that a little more.

In the process of matching rules we end up deciding that variables should be substituted with different values to make the match word. And in the course of trying different rules for a given goal, we might substitute one value in for a variable, then give up and use a different substitution. This can be repeated many times.

We’ll deal with this by having the code for Prove pass along a **linked list** of substitutions of variable values. Every time we need to know the value of a variable, we can over the substitution to see if that variable has been given a value. If it doesn’t appear in the substitution list, it hasn’t been given a value yet. If it does and that value is a string, then that string is the variable’s value.

The complication is that you’re allowed to substitute a variable with *another variable*. In that case, the first variable’s value is the value of the second variable, whatever it may be. That happened when we tried the rule “r[?x] if p[?x] and q[?x]” for the goal r[?y]. Then the system had to remember to substitute ?x for ?y. Later, when we’d decided that ?x = “a”, then magically ?y=”a” too. After that, we decided ?x=”a” was a bad idea and tried ?x=”b” and so then magically ?y=”b” too.

Every time we decide to try a new value for a variable, we add to the beginning of he linked list (just like in CS-111). It’s important that this doesn’t change the original list: when we backtrack, we need to go back to the original list, so we can’t have scribbled over it. Using linked lists this way lets us “undo” variable substitutions. So the sequence of substitutions we went through in the above expand was:

- Start with the empty list (null)
- Then we decided ?y=?x, so our substitution was: ?y->?x
- Then we decided ?x=”a”, so our substitution was: ?x->”a”, ?y->?x
- Then we said “oops, ?x=”a” was a bad idea. Back to just: ?y->?x
- Then we said ?x=”b”, which gave us the substitution: ?x->”b”, ?y->?x
- And that one worked

We’ve already implemented the linked lists for you. They’re in the class `Substitution`.

Unification

The process of finding a substitution that makes two things the same is called **unification**. It takes two arguments and whatever substitutions are already in effect, and it returns the substitution that makes the two arguments be the same, assuming that’s possible. If they’re already the same under the current substitution, it just returns that original substitution. If one of them is a variable, it can just add to the substitution list an extra entry making variable have the other item as its value. And if the two values are just different (and not variables), then unification (matching) fails.

The Prove methods

Conceptually, the Prove method takes:

- A goal to prove (that is, a predicate and some arguments), and
- A substitution linked list (or null) for any substitutions we’ve already committed to

And it returns a new substitution (possibly the same as the previous one) that makes the goal be true. Unfortunately, it needs to be able to backtrack, so we have to do the thing we did before with having a success continuation we call when we succeed, and then we return a Boolean for whether we succeeded or not. In this case, the success continuation is going to take the new Substitution as its argument and return a

Boolean indicating whether it liked that substitution, or whether we need to backtrack and try something else.

The prover is implemented by the methods:

```
/// <summary>
/// Try to prove goal using a substitution
/// </summary>
/// <param name="g">Goal to prove</param>
/// <param name="s">Variable substitutions in effect</param>
/// <param name="k">Success continuation to call with final substitutions</param>
/// <returns>True if successful and continuation returned true</returns>
public static bool Prove(Goal g, Substitution s, SuccessContinuation k)
{
   ...
}

/// <summary>
/// Try to prove goal using the specified rule
/// This will work if the goal can be unified with the head of the rule and all
/// the subgoals can also be proven.
/// </summary>
/// <param name="g">Goal to prove</param>
/// <param name="r">Rule to try to use to prove it.</param>
/// <param name="s">Substitutions in effect</param>
/// <param name="k">Success continuation to call with final substitutions</param>
/// <returns>Successful and continuation returned true</returns>
public static bool ProveUsingRule(Goal g, Rule r, Substitution s,
                                  SuccessContinuation k)
{
    ...
}
```

`Prove` should take a `Goal` g and try each of the rules for the predicate. For each of them, it should try calling `ProveWithRule`. `ProveWithRule` then checks whether it can match the rule to the goal. If so, it recursively call `Prove` on each of the rule's subgoals.

So what the system *really* does in the last example is the following. Here I won't write out the success continuations in the arguments for the calls, I'll just write k for "put a kontinuation here":

- **Call Prove(r[?y], null, k)**
  - I.e. try to prove r[?y] with no substitutions (null is the empty substitution)
  - It tries "all the rules" for r, but there's **only one rule**: r[?x] is true if both p[?x] and q[?x] are true
  - It makes a copy of that rule; it doesn't matter in this example that we make a copy, but it matters if you have recursive predicates.
  - It calls **ProveUsingRule(r[?y], the rule, q[?x]), null, k)**
    - i.e. try to prove it using that rule, and still no substitutions
    - ProveUsingRule tries to **unify r[?y] with the conclusion ("head") of the rule, r[?x]**
    - It can do that using the substitution ?y->?x
    - ProveUsingRule now has to run the subgoals ("body") of the rule
      - It calls the first subgoal, p[?x]
        - So it calls **Prove(p[?x], ?y->?x, k)**
          - I.e. "prove p[?x], understanding that ?y is really ?x"

- Prove tries all the rules for p, in order:
    - It calls **ProveUsingRule(p[?x],**
      the rule p[“a”].Fact(),
      ?y->?x,
      k)
        - I.e. “try to prove p[?x] using p[“a”] as a rule, and understanding ?y is really ?x”
        - ProveUsingRule **unifies p[?x] with p[“a”]**
        - That succeeds with the substitution ?x->”a”, ?y->?x
        - There are no subgoals, of this fact, so it calls its success continuation, passing it the substitution ?x->”a”, ?y->?x
        - That puts us back into the previous call to ProveUsingRule, which moves on to recursively prove its second subgoal, q[?x]
            - So it calls **Prove(q[?x],**
              ?x->”a”, ?y->?x,
              k)
            - i.e. prove q[?x], understanding that both ?x and ?y are really “a”
            - **This fails**
            - So we **backtrack**
    - **We go back to Prove(p[?x], ?y->?x, k)**
    - And **try the next rule**
    - Which is p[“b”]
        - It calls **ProveUsingRule(p[?x],**
          the rule:
          p[“b”].Fact(),
          ?y->?x,
          k)
        - That unifies p[?x] with p[“b”] to get the substitution: ?x->”b”, ?y-?x
        - It calls its continuation
        - Which tries to prove the second subgoal again:
            - So it calls **Prove(q[?x],**
              ?x->”b”, ?y->?x,
              k)
            - i.e. prove q[?x], understanding that both ?x and ?y are really “b”
            - **This succeeds**

        - So many happy success continuations are called with the substitution **?x->"b", ?y->?x**
        - **And we are done**

It's hairy to see that all spelled out in detail, but we'll walk you through how to implement it.

Tour of the code

Start by reading the code in the HornClause folder. You can ignore the Tests folder for the moment. As with any C# program, it's structured as a set of classes"

- **Substitution**
  A linked list of variable/value pairs. So in CS214 parlance, it's basically a dictionary.
- **Variable**
  Represents a variable within the language. Remember this doesn't hold the value of the variable. It's really just a key into the Substitution dictionary, which holds the values. That lets us give values to variables by adding to the substitution and undo that assignment by going back to the old substitution.
- **Predicate**
  A thing you can prove that takes arguments. It includes a set of Rules that include calls to other Predicates. It's analogous to PhrasalCategory in the parser assignment: you're going to iterate through the rules trying them, just like in PhrasalCategory.
- **Goal**
  Represents something the system is trying to prove. It contains the Predicate being called and an array of its arguments (strings and/or Variables)
- **Rule**
  Represents an If/Then rule. The "if" part is an array of Goals called its Body and the "then" part is a Goal called (for historical reasons) its Head. It's analogous to Sequence in the parser assignment: you're going to recurse through the subgoals in the Body, just like in Sequence.

These are all written for you. You should read them over, but you won't need to modify them.

The code you'll write will be in two "static" classes. Static classes aren't real classes (you can't make an instance of one); they're just places where you can group related methods:

- **Unifier.cs**
  This contains the methods for unifying values
- **Prover.cs**
  This is where `Prove` and `ProveUsingRule` go

It's these two files that you'll fill in and that you'll turn in when you're done.

Your job

Okay. That took a lot of setup. Sorry. But in the end, you're just writing a few methods; in the neighborhood of 50-100 lines of code.

Writing the unifier

Start by writing the code to match values to one another (unification).

Dereferencing

When you compare values, you'll compare the values you get *after* substitution. So start by implementing the method `Dereference` in Unifier.cs. It should do all the substitutions it can on a value and return the result:

```
/// <summary>
/// Return the final value of a variable within a substitution.  It's defined as
/// follows (constant here just means anything other than a Variable):
/// - Dereference(constant, subst) = constant
/// - Dereference(variable, subst) = Dereference(variable's value in subst, subst)
///
/// Note that this means Dereference will only ever return a constant or a
/// variable that doesn't have a value in the substitution, aka an unbound
/// variable.
/// </summary>
public static object Dereference(object term, Substitution subst)
```

There are three cases:

- If `term` isn't a `Variable`, just return it
- If `term` a `Variable`, and `subst` has a substituted value for the variable, then dereference that substituted value (because it may be another variable) and return the result.
- If `term` is a `Variable`, but `subst` doesn't have a value for it, just return the original variable.

You can test whether `term` is a variable by saying:

```
if (term is Variable v)
{
    … code to run if it's a variable …
}
else
{
    … code to run if it isn't
}
```

This will run the first chunk of code if `term` points at a `Variable` object. And if it does, it will cast `term` to the type `Variable` and put the result in `v`. So the compiler will think of term as an object (which could be a string or `Variable`), but it will understand that v is a `Variable` and so you can call `Variable` methods on it. If `term` isn't a `Variable`, it runs the else part of the if.

You can use the method `Substitution.Lookup` to check whether a `Variable` appears in a `Substitution`:

```
/// <summary>
/// If the Substitution has a binding for the variable, then return true and
/// output its value.  Otherwise return false.
/// </summary>
/// <param name="substitution">Substitution to check</param>
/// <param name="variable">Variable to find a value for</param>
/// <param name="value">Output argument to write the value to</param>
/// <returns>True if the variable has a value in this substitution</returns>
public static bool Lookup(Substitution substitution, Variable variable,
                          out object value)
```

This uses a programming technique called “output parameters.” Output parameters are parameters that you include in the call to the method, but that the method will write a value back to when it’s done. It’s basically like returning a second value from the method, only it’s often more convenient to write it this way.

It’s common to use output parameters when looking things up in dictionaries because you don’t know if the thing you’re looking up will appear there or not. So you have it return a Boolean to indicate whether it found the variable, and return the actual value of the variable in the output parameter. You can then write a call to it this way:

```
if (Substitution.Lookup(subst, v, out var vValue)
{
   … code to run if v has a value in subst …
}
else
{
   … code to run otherwise …
}
```

If `subst` gives v a value, then this will put that value in `vValue` and run the first part of the if. Otherwise it will run the else part of the if.

We’ve provided a set of unit tests for `Dereference()`, so you can test your implementation.

Unifying two values

Now write Unify (notice this time your writing a method that takes an output parameter):

```
/// <summary>
/// Test if it's possible to make the dereferenced versions of the two values the
/// same, possibly by extending the substitution.  Output the Substitution that
/// makes them the same.  If the dereferenced versions are already the same, then
/// the unifiedSubst will just be the original one.  If they're not the same, but
/// one of them is a variable, then the unifiedSubst will be the original subst
/// extended with the variable set to the other dereferenced value.
///
/// If the dereferenced values are different constants, they can't be made the
/// same, and unify should return false.
/// </summary>
/// <param name="a">The first value to compare</param>
/// <param name="b">The second value to compare</param>
/// <param name="subst">The substitution currently in use</param>
/// <param name="unifyingSubst">The extension of subst that makes a and b the same
(this may just be the original subst)</param>
/// <returns></returns>
public static bool Unify(object a, object b, Substitution subst,
                         out Substitution unifyingSubst)
```

This checks to see if a and b can be made the same through substitution. If so, it sets `unifyingSubst` to the substitution that does that (which may just be `subst`), and returns true. Otherwise, it returns false.

Unify is also called with `subst`, which has any substitutions that are already in effect. So start by calling `Dereference` on both a and b, asking what their current values are given the substitution `subst`. Those dereferenced values are the ones we're going to compare:

- If the dereference values are the same, then it's easy. The existing substitution `subst` already unifies them. So set `unfyingSubst = subst` and return true.

  IMPORTANT: when comparing two `object` variables, `x` and `y` to see if they're the same, use `x.Equals(y)`. If you use ==, it asks if they're the literal same object in memory, which will return false for strings that are different memory objects but have the same data in them. `Equals()` will realize when they're strings and compare the data in them.

- Otherwise, if either of them is a variable, then you can unify them by substituting the one that's a variable with the value of the other. So set:
  ```
      unifyingSubst = new Substitution(theOneThatsAVariable, theOtherOne,
                                       subst);
  ```
  and return true. Again, you can use the `if (x is Variable v)` … construction to test if something is a variable.

- Otherwise, they're different and they're not variables, so you can't unify them. Set `unifyingSubst` to null and return false.

Once again, we've give you unit tests to test your implementation.

Unifying two argument lists

The last thing to implement in the unifier is something that takes two arrays that each contain `Variables` and/or strings, and finds the `Substitution` that makes all their respective values the same:

```
/// <summary>
/// Check if a[0] can be unified with b[0], a[1] with b[1], etc.  Output the
/// substitution needed to make each pair unify.
/// </summary>
public static bool UnifyArrays(object[] a, object[] b, Substitution subst,
                               out Substitution unifyingSubst)
```

This should return false if the arrays are different lengths, or if there's no way to make the arrays be the same. All you need to do is to loop, calling Unify on a[i] and b[i], updating your substitution as you go along. If Unify ever returns false, you return false. Otherwise, you output the substitution you end up with at the end and return true.

Implementing the prover

Okay. Now you're ready to do some inference. You need to implement the two methods in Prover.cs.

Implementing ProveUsingRule for facts

Now let's write a simplified version of ProveUsingRule that doesn't worry about subgoals (the Body of the rule). It takes a goal to prove, a rule to try to use to prove it, the substitutions currently in effect, and a success continuation. In this assignment the success continuation takes a `Substitution` and returns a Boolean:

```
        /// <summary>
        /// Try to prove goal using the specified rule
        /// This will work if the goal can be unified with the head of the rule and all
        /// the subgoals can also be proven.
        /// </summary>
        /// <param name="g">Goal to prove</param>
        /// <param name="r">Rule to try to use to prove it.</param>
        /// <param name="s">Substitutions in effect</param>
        /// <param name="k">Success continuation to call with final substitutions</param>
        /// <returns>Successful and continuation returned true</returns>
        public static bool ProveUsingRule(Goal g, Rule r, Substitution s,
                                          SuccessContinuation k)
```

If we don't worry about subgoals, then all `ProveUsingRule` has to do is ask if the Head of the rule (`r.`Head) matches the goal we're trying to prove, g. The Head and g are both `Goal` objects, but their `Predicate` fields will always be the same, or we wouldn't be trying this rule. So all you have to do is ask if you can unify g's `Arguments` field with r's head's `Arguments` field.

So call `Unifier.UnifyArrays` on the two argument arrays. If `UnifyArrays` succeeds, then call the success continuation and pass it the resulting substitution, and return whatever the continuation returns. Otherwise, just return false.

We've given you a unit test, `RuleTest()` in ProofTests.cs to test out this simplified implementation.

Implementing Prove

Now, let's implement Prove. This is the thing that tries all the rules for a predicate:

```
        /// <summary>
        /// Try to prove goal using a substitution
        /// </summary>
        /// <param name="g">Goal to prove</param>
        /// <param name="s">Variable substitutions in effect</param>
        /// <param name="k">Success continuation to call with final substitutions</param>
        /// <returns>True if successful and continuation returned true</returns>
        public static bool Prove(Goal g, Substitution s, SuccessContinuation k)
```

Once again, this is like the PhrasalCategory in the parser: it loops through all the rules, trying them. In particular, it should loop through all the rules of g's Predicate. For each one, it should try calling `ProveUsingRule`.

> **VERY IMPORTANT:** Our tests include a predicate that is itself recursive (see DirectedAcyclicGraphTest() in the unit tests). That means that every time we use the rules for the predicate, we need to use **new copies of the Rules with new Variables**. Otherwise, all the recursive calls are sharing one set of "global" variables, and it won't work. We've given you a method that makes the new copy. You can just **call Copy()** and you get a brand new rule that's identical except it has new Variables in it. So when Prove calls `ProveUsingRule`, it should call Copy`()` on that rule first and pass the copied version to `ProveUsingRule`.

Hopefully the unit test `FactTest()` will also work now.

Implementing subgoaling

Okay. This is the final thing you need to do. Once you do this, you have a simple, but full inference system.

Go back to your implementation of `ProveUsingRule` and modify it to recursively call `Prove` for each subgoal of the rule. You can find the subgoals in the Body field of the rule. This is the way that it ends up being like the Sequence category in the parser.

As before, you will match the Head to the goal. If the match fails, then you return false. But if it succeeds, you need to run the subgoals. You'll do this with a recursive helper function. It should take the substitution you got from unifying the head and the goal, and then:

- Call `Prove` on the first subgoal (`r.Body[0]`) using the substitution we got when we unified the head with the goal.
- If that works, it should recursively call the second subgoal (`r.Body[1]`) using the substitution we got from the first subgoal. In other words, the continuation for the Prove call for the first subgoal, should call back into the helper function to run the second subgoal.
- When we run out of subgoals, the helper function should call `ProveUsingRule`'s continuation k on the final substitution
- If any step, `Prove` fails on a subgoal, then backtrack to the previous subgoal by returning false

If a rule is a fact, the Body will be empty, so make sure your helper function can handle that. (It can just call `ProveUsingRule`'s continuation immediately).

At this point, hopefully all the unit tests run. If so, you're done! Congratulations on writing your first inference engine!

Turning it in

Make a zip file containing only Prover.cs and Unifier.cs and upload it to Canvas.