4.10 ***Extended example:*** Unification and resolution

Relations are an important way to organize information. They play a central role in several kinds of databases—information can be queried from collections of relations, and new information can be deduced.

Suppose we have two sets—a set of people and a set of things—and two relations, *likes* from people to things and *knows* from people to people. We do not know exhaustively all the tuples in these relations, but suppose we do know at least that the tuples (Kathy, Cars), (Maisie, Cars), and (Maisie, Oatmeal) are in the relation *likes.*

Now suppose we also know some propositions about these particular relations. For example, anyone who likes cars or likes oatmeal knows Jim, and anyone who likes cars and likes oatmeal knows Fred. From these facts we would be able to deduce that Kathy knows Jim and that Maisie knows both Jim and Fred.

logic programming

In this section we develop a system to model information about a set of relations and to infer new information about those relations. Setting up the input information for such a system is called *logic programming*. The programming language Prolog is the most widely used language for this style of programming. The system we are writing is an interpreter of sorts for a very small programming language similar to Prolog.

To begin, suppose we have two kinds of facts about relations: facts that tell us a specific tuple in the relation ("Stephanie likes chocolate"), and facts that, using quantification, tell us about a large set of tuples ("Everyone likes chocolate"). In addition to facts we have queries, which can take the same two forms. If a query has the first form ("Does Stephanie like chocolate?"), then the result should be a yes or no answer; for the second form ("What does Stephanie like?") a query should result in a set of things we know fit the query. Suppose, then, this is our set of facts:

Likes(Kathy, Cars)
Likes(Maisie, Cars)
Likes(Maisie, Oatmeal)
Likes(Stephanie, Michigan)
Likes(x, Chocolate)
Likes(Harvey, x)

Sample queries:

Likes(Maisie, Oatmeal) [Answer: yes]
Likes(Kathy, Oatmeal) [Answer: no]
Likes(Stephanie, x) [Answer: { { x=Michigan }, { x=Chocolate } }]
Likes(x, Cars) [Answer: { { x=Harvey }, { x=Kathy }, { x=Maisie } }]

Notice that for facts variables are implicitly universally quantified—x means "everything." In queries, variables range over a truth set.

The main building block in the process by which queries are answered is called *unification*. The idea is to take a query and a fact and make them match, if possible, by substituting a constant in one of them for a variable in the other. Consider the following pairs of sentences. For readability, assume the first is a query and the second is a fact, but it really does not matter.

unification

Likes(Maisie, Oatmeal), Likes(Maisie, Oatmeal)	There are no variables, and this matches immediately. No substitution is necessary.
Likes(Stephanie, x), Likes(Stephanie, Michigan)	These can be unified by substituting Michigan for x.
Likes(Stephanie, y), Likes(Maisie, Oatmeal)	The constants Stephanie and Maisie are in conflict. No substitution for variables can fix this.
Likes(Stephanie, y), Likes(x, Chocolate)	These can be unified by substituting Stephanie for x and Chocolate for y.

In the last case, this works only because different variables were used. We cannot unify "Likes(Stephanie, x)" and "Likes(x, Chocolate)". However, since the name of a variable does not affect meaning, we can replace any variable with another name to avoid clashes like this.

Before deriving an algorithm for unification, consider how to represent the information. Variables and strings can appear in the same place, so they can be subsumed in the same datatype which we will call `atom`. A fact contains a `string` for the name of the relation and a list of `atom`s. We do not need a separate type for queries—they are essentially facts put to a different purpose.

```
datatype atom = Var of string | Const of string;
datatype fact = Fact of string * atom list;
```

Some examples:

```
Fact("Likes", [Const("Kathy"), Const("Cars")]);
Fact("Likes", [Const("Maisie"), Const("Cars")]);
Fact("Likes", [Const("Maisie"), Const("Oatmeal")]);
Fact("Likes", [Const("Stephanie"), Const("Michigan")]);
Fact("Likes", [Var("x"), Const("Chocolate")]);
Fact("Likes", [Const("Harvey"), Const("x")]);
```

As alluded to earlier, the solution to a unification is a *substitution*, a set of pairs associating variables with atoms that can replace them. If no such substitution exists, then the unification results in a failure. Thus we have the following datatype to model results of unifications, as well as a function to lookup a variable in a substitution. Since the first item in each pair must be a variable, we represent it with a `string` rather than an `atom`.

substitution

```
datatype substitution = Failure | Sub of (string * atom) list;
exception failedLookup;

fun lookup(v, []) = raise failedLookup
  | lookup(v, (w, x)::rest) = if v = w then x else lookup(v, rest);
```

To unify two facts, we make sure that we are testing the same relation (which we will call the facts' *operator*s), and then unify the lists of parameters item by item, updating our solution substitution as we go. This breaks down to unifying individual atoms, given a "substitution so far." We do this with two mutually recursive functions, `unifyAtom` and `unifyVar`.

```
fun unifyAtom(Const(c), Const(d), subst) =
         if c = d then subst else Failure
  | unifyAtom(Var(v), y, subst) = unifyVar(v, y, subst)
  | unifyAtom(x, Var(w), subst) = unifyVar(w, x, subst)

and unifyVar(v, y, s as Sub(subs)) =
     (unifyAtom(lookup(v, subs), y, s)
      handle failedLookup =>
        (case y of
             Var(w) => (unifyAtom(Var(v), lookup(w, subs), s)
                        handle failedLookup =>
                           if v = w then Failure
                                    else Sub((v, y)::subs))
           | Const(s) => Sub((v, Const(s))::subs)))
  | unifyVar(_,_,_) = Failure;
```

as

This is a tricky piece of code. First, one new bit of ML: The third parameter to `unifyVar` is written `s as Sub(subs)`. The `as` construct allows us to define two names in one parameter: the variable `s` to stand for the entire substitution value and the variable `subs` to stand for the (string * atom) list component. It gives us the best of both worlds between parameter names and pattern-matching.

Now for what these functions do. If given two constants, `unifyAtom` makes sure they are equal. If given at least one variable, the action is passed off to `unifyVar`; notice that the parameter `v` has type string (the name of a variable) and parameter `y` has type atom—it could be either a variable or a constant. If the variable `v` already has a replacement in the substitution `s`, then try to unify that replacement with `y`.

Instead of first testing if `v` has a replacement and then retrieving the replacement if it exists, we optimistically retrieve the replacement and handle an exception if no replacement exists. Thus the first `handle failedLookup` clause is for the case that `v` has no replacement yet. If that happens, we look at `y`.

If `y` is a variable, then do what we did with `v`: optimistically lookup its replacement and unify. If that fails (another `handle failedLookup`), then we have two variables without replacements. If they are the same variable, that is bad, and we throw up our hands. If they are not equal, then we add `y` as a replacement for `v` to our substitution. We similarly add to our substitution if `y` is a constant.

Unifying a list of atoms is more straightforward. We apply `unifyAtom` on every pair of items in the two lists, but accumulate the substitution as we go—if the unification of two atoms results in a new replacement pair, that pair must be included in the substitution used for the later atoms in the list. If the lists are not of the same length, then we fail.

```
fun unifyList(_, _, Failure) = Failure
  | unifyList(xFst::xRest, yFst::yRest, subst) =
      unifyList(xRest, yRest, unifyAtom(xFst, yFst, subst))
  | unifyList([], [], subst) = subst
  | unifyList(_, _, _) = Failure;
```

The main unification function is `unifyFact`, which checks the operators and parameter lists.

```
fun unifyFact(Fact(xOp, xArgs), Fact(yOp, yArgs), subst) =
    if xOp = yOp then unifyList(xArgs, yArgs, subst)
                 else Failure;
```

Trying these out:

```
- unifyFact(Fact("Likes", [Const("Stephanie"), Var("x")]),
=           Fact("Likes", [Const("Stephanie"),
=                          Const("Michigan")]), Sub([]));

val it = Sub [("x",Const "Michigan")] : substitution

- unifyFact(Fact("Likes", [Const("Stephanie"), Var("x")]),
=           Fact("Likes", [Var("y"), Const("Chocolate")]),
=                Sub([]));

val it = Sub [("x",Const "Chocolate"),("y",Const "Stephanie")]
    : substitution

- unifyFact(Fact("Likes", [Const("Stephanie"), Var("x")]),
=           Fact("Likes", [Const("Maisie"),
=                          Const("Oatmeal")]), Sub([]));

val it = Failure : substitution
```

Unification is just a part of the process we are interested in. We have not yet addressed how to infer new facts using information like "Everyone who likes cars or oatmeal knows Jim" and "Everyone who likes cars and oatmeal knows Fred." Notice that at their core these are conditionals:

$$\begin{array}{lcl} \forall x, \text{Likes}(x, \text{cars}) \vee \text{Likes}(x, \text{oatmeal}) & \rightarrow & \text{Knows}(x, \text{Jim}) \\ \forall x, \text{Likes}(x, \text{cars}) \wedge \text{Likes}(x, \text{oatmeal}) & \rightarrow & \text{Knows}(x, \text{Fred}) \end{array}$$

We will call sentences like these *rules*, to distinguish them from facts. To standardize the format, we will specify that the hypothesis of any rule is a conjunction of facts (the premises) and the hypothesis is a single fact. As before, all variables are universally quantified. We can handle disjunctions—when we would like to "or" the premises—by splitting them up into several rules. Here is our datatype modeling rules and some examples.

```
datatype rule = Rule of fact list * fact;

Rule([Fact("Likes", [Var("x"), Const("Cars")])],
     Fact("Knows", [Var("x"), Const("Jim")]));
Rule([Fact("Likes", [Var("x"), Const("Oatmeal")])],
     Fact("Knows", [Var("x"), Const("Jim")]));
Rule([Fact("Likes", [Var("x"), Const("Cars")]),
      Fact("Likes", [Var("x"), Const("Oatmeal")])],
     Fact("Knows", [Var("x"), Const("Fred")]));
```

resolution

The process of generating new facts from a set of facts and rules is called *resolution*. If a fact matches the premise of a rule under a certain substitution, then the conclusion of the rule is a known fact under that substitution. As we work through the process of resolution, we will assume a list of facts and rules; a current *goal*, the query we are currently trying to prove or find a substitution for; and a substitution or set of substitutions. As before, the result of a query is a set of substitutions that make the query true.

We will maintain a list of facts and a list of rules as reference variables in the system. This way it is easy to add to them, and they do not have to be passed to the resolution functions.

```
val factList =
  ref [Fact("Likes", [Const("Kathy"), Const("Cars")]),
   Fact("Likes", [Const("Maisie"), Const("Cars")]),
   Fact("Likes", [Const("Maisie"), Const("Oatmeal")]),
   Fact("Likes", [Const("Stephanie"), Const("Michigan")]),
   Fact("Likes", [Var("x"), Const("Chocolate")]),
   Fact("Likes", [Const("Harvey"), Var("x")])];
val ruleList =
  ref [Rule([Fact("Likes", [Var("x"), Const("Cars")])],
          Fact("Knows", [Var("x"), Const("Jim")])),
     Rule([Fact("Likes", [Var("x"), Const("Oatmeal")])],
          Fact("Knows", [Var("x"), Const("Jim")])),
     Rule([Fact("Likes", [Var("x"), Const("Cars")]),
           Fact("Likes", [Var("x"), Const("Oatmeal")])],
          Fact("Knows", [Var("x"), Const("Fred")]))];
```

Here is how resolution works.

Resolving a goal. Suppose we have a goal to resolve. This may be a sub-goal produced by the resolve functions themselves, or it may be a query that the user makes. We also have a substitution generated so far, but it may be easier to assume for now that the substitution is empty. We find all substitutions (if any) by resolving using the fact list and concatenate that list of substitutions to those found by resolving using the rules. These resolution functions are mutually recursive, but here is what the main `resolve` function looks like.

```
and resolve(goal, subst) =
    resolveFact(goal, !factList, subst)@
    resolveRule(goal, !ruleList, subst);
```

Resolving a goal using facts. The function `resolveFact` takes a goal, a list of facts, and a substitution (the replacements found so far). We attempt to unify the goal with the first fact in the list. If it fails, try the next one. If it succeeds with a refined substitution, then try the next one anyway, concatenating the refined substitution with the substitutions found by trying the rest of the list.

Resolving a goal using rules. The function `resolveRule` takes a goal, a list of rules, and a substitution. For each rule in the list, we try to unify the goal with the conclusion. If it matches (with a refined substitution), then the substitution only works if we can satisfy the premises of the rule. All the premises, then, become sub-goals which we try to resolve, starting with the substitution that unifies the goal with the conclusion. Any substitution we find to work from this rule we add to a list of other substitutions that work, using other rules.

Suppose our goal is "Who does Harvey know?"—that is, Knows(Harvey, x). Since none of the facts in our list are about the Knows relation, we start looking at the rules. The first rule has Knows(x, Jim) as a conclusion, and that would match if it were not for the reuse of variable x. We will rename the variable to get Knows(x_1, Jim), and now the two are unified with substitution { x_1 = Harvey, x = Jim }.

This does not mean we have yet found someone Harvey knows. We still need to prove the premises. Our subgoal is now Likes(Harvey, Cars). More accurately, our subgoal is Likes(x_1, Cars) with substitution { x_1 = Harvey, x = Jim }. Testing this against the facts, we find that it unifies with Likes(Harvey, x_2) (the variable x

renamed to be unambiguous), with refined substitution { x_1 = Harvey, x = Jim, x_2 = x_1 } Notice that variables can be associated with other variables in a substitution. It does imply, of course, that ultimately x_2 = Harvey.

However, that substitution is not the only answer, nor is that the only route to that answer. The next rule in the list also has Knows(x_1, Jim) as a conclusion, with Likes(x_1, Oatmeal) as the premise. This also can be resolved with the fact Likes(Harvey, x_2).

Finally, the query unifies with the conclusion of the third rule, Knows(x_1, Fred). The substitution is { x_1 = Harvey, x = Fred }. Now we have a list of two sub-goals: Likes(x_1, Oatmeal) and Likes(x_1, Cars). It is important to note that both variables are x_1—that is, we already have a value for them in the substitution. We need to satisfy both of them.

As we have seen, we can resolve Likes(x_1, Oatmeal) with this substitution by unifying with Likes(Harvey, x_2). Our substitution is now { x_1 = Harvey, x = Jim, x_2 = Oatmeal }, and we need to use this substitution when resolving for the other subgoal. It can be resolved by unifying with Likes(Harvey, x_3)—same fact as before, but with a fresh variable. The final substitution in this case is { x_1 = Harvey, x = Jim, x_2 = Oatmeal, x_3 = Cars }.

Two observations in particular with all this. First, to do any of this, we need to be able to take a fact or rule and give the variables fresh, unique names. As an easy naming scheme, we will append each variable name with a unique number. The mechanism for doing this is a reference variable `idGen` to keep track of the last number used and a function `makeUnique`:

```
val idGen = ref 0;

fun makeUnique(v) = (idGen := !idGen + 1;
                     v ^ Int.toString(!idGen));
```

Making all the variables unique in parts of our data representation is handled by functions `standardizeFact`, `standardizeRule`, and others. The algorithm for doing this is simple but tedious. The code can be found on the accompanying website.

The second observation involves how we resolve lists of goals. Given a list of goals and a single substitution, we try to resolve the first goal. Remember that `resolve` returns a list of substitutions that work. If that list is empty, everything fails (for the original given substitution). If it is not empty, then we need to check every other goal in the list *with every substitution* and concatenate the results together. This function is part of the process:

```
and resolveSubstList(goals, []) = []
  | resolveSubstList(goals, subst::rest) =
    resolveGoalList(goals, subst)@resolveSubstList(goals, rest)
```

As a final step, we provide a few functions to make the system easier to use. We want functions that allow the user to tell the system a new fact or rule and to ask the system a query. The tell functions are easy:

```
fun tellFact(fact) = factList := fact::!factList;
fun tellRule(rule) = ruleList := rule::!ruleList;
```

The result should be printed in a readable format. The result is a a list of substitutions, and these substitutions include variables that the user does not need to know about. We want to display the ultimate replacement for each of the variables in the original query.

```
fun deepLookup(v, subs) =
    case lookup(v, subs) of
        Const(c) => c
      | Var(vv) => deepLookup(vv, subs);

fun getVars([]) = []
  | getVars(Const(c)::rest) = getVars(rest)
  | getVars(Var(v)::rest) = v::getVars(rest);

fun printResult([], _) = print(";\n")
  | printResult(_, Failure) = print("fail")  (* shouldn’t happen *)
  | printResult(x::rest, Sub(subs)) =
    (print(x ^ "=" ^ deepLookup(x, subs) ^ " ");
     printResult(rest, Sub(subs)));

fun printResults(vars, []) = print(".\n")
  | printResults(vars, sub::rest) =
    (printResult(vars, sub); printResults(vars, rest));
```

The function `ask` puts all this together.

```
fun ask(Fact(oper, args)) =
    let val results = resolve(Fact(oper, args), Sub([]));
        val vars = getVars(args);
    in if results = [] then print("no\n")
                           else printResults(vars, results)
    end;
```

Trying it out:

```
  - ask(Fact("Likes", [Const("Maisie"), Var("x")]));
```

```
x=Cars ;
x=Oatmeal ;
x=Chocolate ;
.
val it = () : unit

- ask(Fact("Knows", [Const("Harvey"), Var("x")]));

x=Jim ;
x=Jim ;
x=Fred ;
.
val it = () : unit
```

More information about the unification and resolutions algorithms can be found in Harrison [12] and Russell and Norvig [24].

Project

4.A Finish the following function to resolve a goal using a list of facts.

```
fun resolveFact(goal, fact::rest, subst) =
    (case unifyFact(goal,
                #1(standardizeFact(fact, [])),
                    subst) of
         Failure => ??
       | Sub(s) => ??
  | resolveFact(goal, [], subst) = []
```

4.B Finish the following function to resolve a goal using a list of rules.

```
and resolveRule(goal, r::rest, subst)  =
    let val Rule(premises, conclus) =
                standardizeRule(r)
    in
        case unifyFact(goal, conclus, subst) of
            Failure => ??
          | subst2 =>
            (case resolveGoalList(premises,
                                  subst2) of
                 [] => ??
               | substs => ??
    end
  | resolveRule(goal, [], subst) = []
```

4.C Finish the following function to resolve a list of goals. This will include a call to function `resolveSubstList`.

```
and resolveGoalList(goal::rest, subst) =
    (case ?? of
        [] => ??
      | substs=> ??
  | resolveGoalList([], subst) = [subst]
```

4.D Write improved versions of the tell and ask functions that will parse string input and create appropriate datatype values, so the user can enter something like `tell("Likes(Maisie, ML)")` instead of `tellFact(Fact("Likes", [Const("Maisie"), Const("ML")]))`. Use as a model the parser for the language system in Section 1.17. Make reasonable assumptions like all constants begin with a capital letter and all variables begin with a lowercase letter.

4.E Make a collection of facts and rules so that you can use the system to solve the following problem.

Angie, Brad, Casey, Dora, Evert, and Fuchsia are at a conference. It is lunchtime, and they need to figure out who can eat with whom at what restaurant.

The nearby restaurants are the East Grille, Bertie's, the City Tavern, and Fish King. The East Grille serves hamburgers, tofu, and halibut. Bertie's serves hamburgers. The City Tavern serves hamburgers and tofu. Fish King serves halibut. Any place that serves hamburgers also serves French fries. Bertie's and Fish King have patios. The City Tavern and Fish King require patrons to wear shoes, and the East Grille requires (male) patrons to wear ties. Any place that requires ties also requires shoes.

Angie wants to eat halibut, and Brad wants to eat French fries. Casey will only eat a place that requires shoes, and he wants to eat tofu. Dora is more formal and wants only to eat at a place that requires ties. Evert wants to discuss his research with Angie and so will eat any place where she will eat. Fuchsia wants to eat on a patio.

Who can eat with whom, and where? (You do not need to have a solution to Project 4.D to do this one, but it would make the input more convenient.)

4.11 *Special topic:* Representing relations

Our initial problem with the relation `inSameTimeZone` was that it was missing self-loops. Hence the reflexive closure would have solved our problem just as well as the transitive closure. Unfortunately, there is no way to compute the reflexive closure