4.5 Unification

This section describes an implementation of the query interpreter that performs inference in the `logic` language. The interpreter is a general problem solver, but has substantial limitations on the scale and type of problems it can solve. More sophisticated logical programming languages exist, but the construction of efficient inference procedures remains an active research topic in computer science.

The fundamental operation performed by the query interpreter is called *unification*. Unification is a general method of matching a query to a fact, each of which may contain variables. The query interpreter applies this operation repeatedly, first to match the original query to conclusions of facts, and then to match the hypotheses of facts to other conclusions in the database. In doing so, the query interpreter performs a search through the space of all facts related to a query. If it finds a way to support that query with an assignment of values to variables, it returns that assignment as a successful result.

4.5.1 Pattern Matching

In order to return simple facts that match a query, the interpreter must match a query that contains variables with a fact that does not. For example, the query `(query (parent abraham ?child))` and the fact `(fact (parent abraham barack))` match, if the variable `?child` takes the value `barack`.

In general, a pattern matches some expression (a possibly nested Scheme list) if there is a binding of variable names to values such that substituting those values into the pattern yields the expression.

For example, the expression `((a b) c (a b))` matches the pattern `(?x c ?x)` with variable `?x` bound to value `(a b)`. The same expression matches the pattern `((a ?y) ?z (a b))` with variable `?y` bound to `b` and `?z` bound to `c`.

4.5.2 Representing Facts and Queries

The following examples can be replicated by importing the provided logic example program.

```
>>> from logic import *
```

Both queries and facts are represented as Scheme lists in the logic language, using the same `Pair` class and `nil` object in the previous chapter. For example, the query expression `(?x c ?x)` is represented as nested `Pair` instances.

```
>>> read_line("(?x c ?x)")
Pair('?x', Pair('c', Pair('?x', nil)))
```

As in the Scheme project, an environment that binds symbols to values is represented with an instance of the `Frame` class, which has an attribute called `bindings`.

The function that performs pattern matching in the `logic` language is called `unify`. It takes two inputs, `e` and `f`, as well as an environment `env` that records the bindings of variables to values.

```
>>> e = read_line("((a b) c (a b))")
>>> f = read_line("(?x c ?x)")
>>> env = Frame(None)
>>> unify(e, f, env)
True
>>> env.bindings
{'?x': Pair('a', Pair('b', nil))}
```

```
>>> print(env.lookup('?x'))
(a b)
```

Above, the return value of `True` from `unify` indicates that the pattern `f` was able to match the expression `e`. The result of unification is recorded in the binding in `env` of `?x` to `(a b)`.

4.5.3 The Unification Algorithm

Unification is a generalization of pattern matching that attempts to find a mapping between two expressions that may both contain variables. The `unify` function implements unification via a recursive process, which performs unification on corresponding parts of two expressions until a contradiction is reached or a viable binding to all variables can be established.

Let us begin with an example. The pattern `(?x ?x)` can match the pattern `((a ? y c) (a b ?z))` because there is an expression with no variables that matches both: `((a b c) (a b c))`. Unification identifies this solution via the following steps:

1. To match the first element of each pattern, the variable `?x` is bound to the expression `(a ?y c)`.
2. To match the second element of each pattern, first the variable `?x` is replaced by its value. Then, `(a ?y c)` is matched to `(a b ?z)` by binding `?y` to `b` and `?z` to `c`.

As a result, the bindings placed in the environment passed to `unify` contain entries for `?x`, `?y`, and `?z`:

```
>>> e = read_line("(?x ?x)")
>>> f = read_line(" ((a ?y c) (a b ?z))")
>>> env = Frame(None)
>>> unify(e, f, env)
True
>>> env.bindings
{'?z': 'c', '?y': 'b', '?x': Pair('a', Pair('?y', Pair('c', nil)))}
```

The result of unification may bind a variable to an expression that also contains variables, as we see above with `?x` bound to `(a ?y c)`. The `bind` function recursively and repeatedly binds all variables to their values in an expression until no bound variables remain.

```
>>> print(bind(e, env))
((a b c) (a b c))
```

In general, unification proceeds by checking several conditions. The implementation of `unify` directly follows the description below.

1. Both inputs `e` and `f` are replaced by their values if they are variables.
2. If `e` and `f` are equal, unification succeeds.
3. If `e` is a variable, unification succeeds and `e` is bound to `f`.
4. If `f` is a variable, unification succeeds and `f` is bound to `e`.
5. If neither is a variable, both are not lists, and they are not equal, then `e` and `f` cannot be unified, and so unification fails.
6. If none of these cases holds, then `e` and `f` are both pairs, and so unification is performed on both their first and second corresponding elements.

```
>>> def unify(e, f, env):
        """Destructively extend ENV so as to unify (make equal) e and f, returni
        True if this succeeds and False otherwise.  ENV may be modified in eithe
        case (its existing bindings are never changed)."""
        e = lookup(e, env)
        f = lookup(f, env)
        if e == f:
            return True
        elif isvar(e):
            env.define(e, f)
            return True
        elif isvar(f):
            env.define(f, e)
```

```
            return True
        elif scheme_atomp(e) or scheme_atomp(f):
            return False
        else:
            return unify(e.first, f.first, env) and unify(e.second, f.second, en
```

4.5.4 Proofs

One way to think about the `logic` language is as a prover of assertions in a formal system. Each stated fact establishes an axiom in a formal system, and each query must be established by the query interpreter from these axioms. That is, each query asserts that there is some assignment to its variables such that all of its sub-expressions simultaneously follow from the facts of the system. The role of the query interpreter is to verify that this is so.

For instance, given the set of facts about dogs, we may assert that there is some common ancestor of Clinton and a tan dog. The query interpreter only outputs `Success!` if it is able to establish that this assertion is true. As a byproduct, it informs us of the name of that common ancestor and the tan dog:

```
(query (ancestor ?a clinton)
       (ancestor ?a ?brown-dog)
       (dog (name ?brown-dog) (color brown)))
```

```
Success!
a: fillmore brown-dog: herbert
a: eisenhower brown-dog: fillmore
a: eisenhower brown-dog: herbert
```

Each of the three assignments shown in the result is a trace of a larger proof that the query is true given the facts. A full proof would include all of the facts that were used, for instance including `(parent abraham clinton)` and `(parent fillmore abraham)`.

4.5.5 Search

In order to establish a query from the facts already established in the system, the query interpreter performs a search in the space of all possible facts. Unification is the primitive operation that pattern matches two expressions. The *search procedure* in a query interpreter chooses what expressions to unify in order to find a set of facts that chain together to establishes the query.

The recursive `search` function implements the search procedure for the `logic` language. It takes as input the Scheme list of `clauses` in the query, an environment `env` containing current bindings of symbols to values (initially empty), and the `depth` of the chain of rules that have been chained together already.

```
>>> def search(clauses, env, depth):
        """Search for an application of rules to establish all the CLAUSES,
        non-destructively extending the unifier ENV.  Limit the search to
        the nested application of DEPTH rules."""
        if clauses is nil:
            yield env
        elif DEPTH_LIMIT is None or depth <= DEPTH_LIMIT:
            if clauses.first.first in ('not', '~'):
                clause = ground(clauses.first.second, env)
                try:
                    next(search(clause, glob, 0))
                except StopIteration:
                    env_head = Frame(env)
                    for result in search(clauses.second, env_head, depth+1):
                        yield result
            else:
                for fact in facts:
                    fact = rename_variables(fact, get_unique_id())
                    env_head = Frame(env)
                    if unify(fact.first, clauses.first, env_head):
```

```
            for env_rule in search(fact.second, env_head, depth+1):
                for result in search(clauses.second, env_rule, depth
                    yield result
```

The search to satisfy all clauses simultaneously begins with the first clause. In the special case where our first clause is negated, rather than trying to unify the first clause of the query with a fact, we check that there is no such unification possible through a recursive call to `search`. If this recursive call yields nothing, we continue the search process with the rest of our clauses. If unification is possible, we fail immediately.

If our first clause is not negated, then for each fact in the database, `search` attempts to unify the first clause of the fact with the first clause of the query. Unification is performed in a new environment `env_head`. As a side effect of unification, variables are bound to values in `env_head`.

If unification is successful, then the clause matches the conclusion of the current rule. The following `for` statement attempts to establish the hypotheses of the rule, so that the conclusion can be established. It is here that the hypotheses of a recursive rule would be passed recursively to `search` in order to be established.

Finally, for every successful search of `fact.second`, the resulting environment is bound to `env_rule`. Given these bindings of values to variables, the final `for` statement searches to establish the rest of the clauses in the initial query. Any successful result is returned via the inner `yield` statement.

Unique names. Unification assumes that no variable is shared among both `e` and `f`. However, we often reuse variable names in the facts and queries of the `logic` language. We would not like to confuse an `?x` in one fact with an `?x` in another; these variables are unrelated. To ensure that names are not confused, before a fact is passed into unify, its variable names are replaced by unique names using `rename_variables` by appending a unique integer for the fact.

```
>>> def rename_variables(expr, n):
        """Rename all variables in EXPR with an identifier N."""
        if isvar(expr):
            return expr + '_' + str(n)
        elif scheme_pairp(expr):
            return Pair(rename_variables(expr.first, n),
                        rename_variables(expr.second, n))
        else:
            return expr
```

The remaining details, including the user interface to the `logic` language and the definition of various helper functions, appears in the logic example.

Continue: 4.6 Distributed Computing