4.5 合一

::: details INFO
译者：silver

来源：4.5 Unification

对应：无
:::

在本节中，我们将介绍用 logic 语言进行推理的查询解释器的实现。解释器是一种通用的问题求解器，但在求解问题的规模和类型上有很大的局限性。目前已有更复杂的逻辑编程语言，但构建高效的推理程序仍是计算机科学领域一个活跃的研究课题。

查询解释器执行的基本操作称为合一。合一是将查询与事实（每个事实都可能包含变量）进行匹配的一般方法。查询解释器反复执行这一操作，首先将原始查询与事实的结论相匹配，然后将事实的假设与数据库中的其他结论相匹配。在此过程中，查询解释器会搜索与查询相关的所有事实。如果它找到了支持查询的变量赋值方法，就会将赋值作为成功结果返回。

4.5.1 模式匹配

为了返回与查询匹配的简单事实，解释器必须将包含变量的查询与不包含变量的事实相匹配。例如，查询 `(query (parent abraham ?child))` 与事实 `(fact (parent abraham barack))` 在变量 `?child` 值为 `barack` 时相匹配。

一般而言，如果有变量名绑定到值，使得将这些值替换到模式中产生表达式，那么模式就与某个表达式（可能是嵌套的 Scheme 列表）匹配。

例如，表达式 `((a b) c (a b))` 与模式 `(?x c ?x)` 匹配，其中变量 `?x` 绑定到值 `(a b)`。同样的表达式也与模式 `((a ?y) ?z (a b))` 匹配，其中变量 `?y` 绑定到 `b`，`?z` 绑定到 `c`。

4.5.2 表示事实和查询

通过导入所提供的逻辑示例程序，可以复制以下示例。

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

在逻辑语言中，查询和事实都以 Scheme 列表的形式表示，使用前一章中的相同 `Pair` 类和 `nil` 对象。例如，查询表达式 `(?x c ?x)` 表示为嵌套的 `Pair` 实例。

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

与 Scheme 项目一样，将符号绑定到值的环境由 `Frame` 类的一个实例表示，该实例具有一个名为 `bindings` 的属性。

在 logic 语言中执行模式匹配的函数称为 `unify`。它接受两个输入，`e` 和 `f`，以及一个记录变量与值绑定的环境 `env`。

```
>>> 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)
```

上面，`unify` 的返回值为 `True`，表示模式 `f` 能够匹配表达式 `e`。合一的结果记录在 `env` 中，将 `?x` 绑定到 `(a b)`。

4.5.3 合一算法

合一是模式匹配的一种推广，它试图在两个可能都包含变量的表达式之间找到映射关系。`unify` 函数通过递归过程实现合一，它对两个表达式的相应部分执行合一，直到出现矛盾或者可以建立与所有变量的可行绑定。

让我们从一个例子开始。模式 `(?x ?x)` 可以匹配模式 `((a ?y c) (a b ?z))`，因为有一个不含变量的表达式可以同时匹配这两个模式：`((a b c) (a b c))`。合一通过以下步骤确定这一解决方案：

1. 匹配每个模式的第一个元素，变量 `?x` 与表达式 `(a ?y c)` 绑定。
2. 匹配每个模式的第二个元素，首先将变量 `?x` 替换为其值。然后，通过将 `?y` 绑定到 `b`，将 `?z` 绑定到 `c`，`(a ?y c)` 与 `(a b ?z)` 匹配。

因此，传递给 `unify` 的环境绑定将包含 `?x`、 `?y` 和 `?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)))}
```

合一的结果可能会将变量绑定到一个同样包含变量的表达式中，就像我们上面看到的将 `?x` 绑定到 `(a ?y c)`。`bind` 函数会递归、重复地将表达式中的所有变量与其值绑定，直到没有绑定变量为止。

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

一般来说，合一是通过检查几个条件来进行的。合一的实现直接遵循下面的描述。

1. 如果输入 `e` 和 `f` 是变量，则用它们的值替换。
2. 如果 `e` 和 `f` 相等，则合一成功。
3. 如果 `e` 是变量，则合一成功，并将 `e` 绑定到 `f`。
4. 如果 `f` 是变量，则合一成功，并将 `f` 绑定到 `e`。
5. 如果两者都不是变量，也不是列表，并且不相等，那么 `e` 和 `f` 就不能合一，因此合一失败。
6. 如果这些情况都不成立，那么 `e` 和 `f` 都是列表，因此要对它们的第一个和第二个对应元素分别进行合一。

```
>>> def unify(e, f, env):
        """Destructively extend ENV so as to unify (make equal) e and f,
returning
        True if this succeeds and False otherwise.  ENV may be modified in either
        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,
env)
```

4.5.4 证明

对于 logic 语言，一种思考方式是将其视为形式系统中断言的证明器。每个陈述的事实在形式系统中建立了一个公理，而每个查询都必须由查询解释器根据这些公理建立。换句话说，每个查询断言存在某种变量赋值，使其所有子表达式都同时符合系统的事实。查询解释器的作用是验证这一点是否成立。

例如，根据有关狗的事实集合，我们可以断言存在克林顿狗和棕褐色狗的共同祖先。查询解释器只有在能够确定这一断言为真的情况下，才会输出 `Success!`。作为副产品，它还会告知我们该共同祖先和棕褐色狗的名称：

```
(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
```

结果中显示的三个赋值中的每一个都是查询在给定事实情况下为真的更大证明的一部分。完整的证明将包括使用的所有事实，例如包括 `(parent abraham clinton)` 和 `(parent fillmore abraham)`。

4.5.5 搜索

为了从系统中已经建立的事实中建立查询，查询解释器在所有可能的事实中进行搜索。合一是对两个表达式进行模式匹配的基本操作。查询解释器中的搜索程序会选择要合一的表达式，以便找到一组事实，将它们串联起来，从而建立查询。

递归的 `search` 函数实现了 logic 语言的搜索过程。它的输入包括表示查询子句的 Scheme 列表 `clauses`、包含当前符号与值绑定（初始为空）的环境 `env`，以及已经串联起来的规则链的深度 `depth`。

```
>>> 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+1):
                                yield result
```

同时满足所有子句的搜索从第一个子句开始。在第一个子句是否定的特殊情况下，我们不再尝试用事实来合一查询的第一个子句，而是通过递归调用 `search` 来检查是否存在这种合一的可能性。如果递归调用一无所获，我们就继续搜索其余子句。如果可以合一，我们就立即失败。

如果我们的第一个子句不是否定，那么对于数据库中的每个事实，`search` 都会尝试将事实的第一个子句与查询的第一个子句合一起来。合一在一个新的环境 `env_head` 中进行。作为合一的副作用，变量会绑定到 `env_head` 中的值。

如果合一成功，则子句与当前规则的结论相匹配。下面的 for 语句试图建立规则的假设，以便建立结论。在这里，递归规则的假设将递归传递给 `search`，从而建立结论。

最后，每次对 `fact.second` 的成功搜索，生成的环境都会绑定到 `env_rule`。在给定这些值对变量的绑定之后，最终的 for 语句进行搜索，以建立初始查询中的其余子句。任何成功的结果都会通过内部的 yield 语句返回。

唯一名称：合一假设 `e` 和 `f` 之间不共享任何变量。然而，我们经常在 logic 语言的事实和查询中重复使用变量名。我们不希望混淆一个事实中的 `?x` 和另一个事实中的 `?x`；这些变量是不相关的。为了确保变量名不会混淆，在将一个事实传入 `unify` 之前，我们会使用 `rename_variables` 将其变量名替换为唯一的变量名，并为该事实添加一个唯一的整数。

```
>>> 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
```

其余的细节，包括 logic 语言的用户界面和各种辅助函数的定义，都展示在 logic 示例中。