Learn Prolog Now 翻译 - 第二章 - 合一和证明搜索 - 第一节， 合一

内容提要：

合一的定义；

一些合一的例子；

触发校验；

使用合一编程；

合一的定义

在上一章的知识库KB4中，我们简单地提及了合一的思想。比如，Prolog将woman(X)和woman(mia)合一，所以把变量X初始化为mia。现在是时候更加细致地研究合一，因为合一是

Prolog中最为基础的思想。

回顾一下Prolog中的三种语句类型：

1. 常量，可能是原子（比如vincent）或者是数字（比如24）。
2. 变量，比如X，Z3，List等。
3. 复杂语句，形式为：functor(term_1, ..., term_n)。

我们首先从两个语句如何合一的定义开始入手。这个定义是基于直觉的，并不是很严谨，两个语句能够合一，必须满足下面两个条件之一：

1. 两个语句是相同的语句；
2. 如果语句中有变量，能够通过将变量初始化后，两个语句是相同的。

举例说明上述定义的含义：

mia和mia是能够合一的，因为它们是相同的原子；

42和42是能够合一的，因为它们是相同的数字；

X和X是能够合一的，因为它们是相同的变量；

woman(mia)和woman(mia)是能够合一的，因为它们是相同的复杂语句；

woman(mia)和woman(vincent)就不能够合一了，因为它们就不相同，也没有包含变量，无法通过变量的初始化来达成合一；

那么mia和X呢？mia和X表面上不是相同的，但是可以通过将变量X初始化为mia，而使得两者相同。所以根据定义的第二种情况，mia和X能够合一；

类似地，woman(X)和woman(mia)能够合一，通过将变量X初始化为mia，使得两个复杂语句相等；

--Benjamin Yan

那么loves(vincent, X)和loves(X, mia)呢？它们不能够合一，因为无法通过初始化变量X而使得两者相等；如果将X初始化为mia，那么两个复杂语句就变成了loves(vincent, mia)和loves(mia, mia)，明显不相等；如果将X初始化为vincent，那么两个复杂语句就变成了loves(vincent, vincent)和loves(vincent, mia)，

也无法相等。

通常我们不仅仅关注两个语句是否能够合一，而是希望进一步知道能够将其中的变量初始化什么样的值使得合一能够实现。Prolog能够给出这样的信息。Prolog在合一的过程中，

会尝试所有可能的初始化值，使得两个语句能够合一。这种能力，结合允许我们构建复杂语句（即，递归结构的语句），使得Prolog的合一成为了强有力的编程机制。

以上是合一基于直觉的基本定义，下面给出更加精确的定义。这个定义不仅仅给出了Prolog的合一，而且定义了如何通过变量初始化来达到合一：

1. 如果term1和term2都是常量，那么term1和term2能够合一，当且仅当它们是相同的原子，或者相同的数字。

2. 如果term1是变量并且term2是任意类型的语句，那么term1和term2能够合一，并且term1被初始化为term2；同理，如果term2是变量并且term1是任意类型的语句，那么term1

和term2能够合一，并且term2被初始化为term1。（如果两个都是变量，他们都能够被互相初始化，即它们共享相同的值）

3. 如果term1和term2都是复杂语句，那么它们能够合一当且仅当：

   3.1 它们有相同的函子和元数，并且

   3.2 所有对应的参数能够合一，并且

   3.3 变量的初始化能够匹配。（比如，如果两个复杂语句在进行合一，不可能在一个复杂语句中将X初始化为mia，在另外一个复杂语句中将X初始化为vincent）

4. 两个语句能够合一当且仅当它们遵循上面三个定义之一。

让我们分析一下上述关于合一精确定义的形式。第一个定义子句给出了常量合一的定义。第二个定义子句给出了如果两个语句中，有一个语句是变量，两者合一的定义（这种情况

两个语句通过将变量初始化为另一个语句，都可以进行合一，类似其他编程语言中的赋值加上模式匹配效果），尤其重要的是，这个子句给出了通过初始化而达成合一的方式。最后，

第三个定义子句给出了两个复杂语句合一的定义。注意这种定义的方式，和我们定义（递归方式）语句的形式是类似的。

第四个子句也很重要：它给出了如果两个语句能够合一，必须是遵循前三个子句定义的原则。反过来说，如果两个语句不能通过前三个子句定义的原则进行合一，那么它们就不能

合一。比如，batman不能和daughter(ink)进行合一，为什么？第一个语句是一个原子，但是第二个语句是复杂语句，合一定义的前三个子句都没有告诉我们如何

将两者进行合一，

所以（根据第四个子句）他们不能合一。

一些合一的例子

为了比较透彻地理解合一，下面是一些例子。在这些例子中，我们会使用一个重要的内置谓词：=/2（回忆一下，/2是指谓词有几个参数）。=/2谓词会测试两个参数是否合一。

比如，如果我们查询：

```
?- =(mia, mia).
```

Prolog会回答true，如果我们查询：

```
?- =(mia, vincent)
```

Prolog会回答false。

但是我们通常不会这样查询，因为前缀语法=(mia, mia)不是很自然。我们更习惯使用中缀语法进行查询：

```
?- mia = mia.
```

Prolog允许我们使用对=/2使用中缀语法。回到上面的这个查询，Prolog会回答true。为什么？这个问题好像太容易了，它们当然是合一的！但是我们如何通过合一定义得出这

个结论？系统化地思考合一是十分重要的，系统化思考意味着我们需要找到上面例子合一的定义子句。分析一下，明显定义子句一是相关的，这个子句定义告诉我们，如果两个原子

要合一，那么它们必须是相同的。正是因为mia和mia是相同的原子，所以合一成功了。

类似地可以解释下面Prolog的回答：

```
?- 2 = 2.
```

Prolog会回答true。

```
?- mia = vincent.
```

Prolog会回答false。

有一个小小的惊喜，比如查询：

```
?- 'mia' = mia.
```

Prolog会回答true。为什么？因为在Prolog中，'mia'和mia是相同的原子。事实上，任何'symbols'都认为和symbols是相同的原子，在一些类型的程序中，这是一个有用的特性，

所以请别忘记这点。

另一方面，如果我们查询：

```
?- '2' = 2.
```

Prolog会回答false，应为2是一个数字，而'2'是一个原子，它们明显不相同。

再试试有变量的查询：

```
?- mia = X.
```

Prolog会回答：X = mia.，这也是一个简单的例子：显而易见X能够和mia合一，Prolog也的确这么做了。但是，我们如何根据合一的严格定义得出这个结论呢？

可以运用的是定义子句2，它告诉我们如果至少有一个变量的合一是如何进行的：即通过将变量初始化为需要合一的另外一个语句，比如上面的mia，Prolog就完成了合一。

下面是一个重要的例子：如果进行下面的查询：

```
?- X = Y.
```

Prolog会如何回答？这依赖不同的Prolog实现，在我的机器上，SWI-Prolog会回答：

```
X = Y
```

即简单地认为X和Y可以合一，因为变量可以和任意类型的语句合并，如果变量和变量合一，表示它们分享同样的值。另外，可能其他Prolog实现会这么回答：

```
X = _5071

Y = _5071

true
```

这里发生了什么？本质上，和之前的第一个回答是一致的。注意_5071是一个变量（以下划线开头），合一的定义子句2告诉我们，如果两个变量合一，它们会共享一个值，

所以Prolog就创建了一个新的值（名字是_5071），X和Y都共享这个值。不必在意_5071的实际值，它仅仅是表示X和Y有相同的值，这个值可能是另外的变量，比如_5075，

或者_6189。

下面的例子仅仅包含了原子和变量，Prolog会如何回答：

```
?- X = mia, X = vincent.
```

Prolog会回答false。这个查询包含了两个目标，X = mia和 X = vincent，分开来看，Prolog会认为两个目标都成功，首先将X初始化为mia，然后是vincent。但是这里存在

问题：一旦Prolog尝试满足第一个目标，X被初始化为mia，它就不能再和vincent合一了（变量只能初始化一次）。所以第二个目标会失败，一个已经初始化的变量不再是变量了，

它已经变成了它初始化的值。

现在，考虑如下的复杂查询：

```
?- k(s(g), Y) = k(X, t(k))
```

Prolog会回答：

```
X = s(g)
```

Y = t(k)

很明显两个复杂语句通过变量的初始化已经合一了。但是这个过程是如何匹配合一定义的了？首先，因为我们试图合一两个复杂语句，所以定义子句3必须用到，所以我们首先

检查两个复杂语句是否有相同的函子和元数，它们确实相同；定义子句3告诉我们必须将复杂语句的每个对应的参数都合一，所以对于第一个参数，s(g)和X？通过将X初始为s(g)

能够合一；对于第二个参数，Y和t(k)，通过将Y初始化为t(k)也能够合一。

下面是另外一个复杂语句的例子：

?- k(s(g), t(k)) = k(X, t(Y)).

Prolog会回答：

X = s(g)

Y = k

通过变量的初始化，两个复杂语句可以合一。可以自己尝试根据合一定义，一步一步解释这个过程。

下面是最后一个例子：

?- loves(X, X) = loves(marchellus, mia).

Prolog会回答no。虽然两个都是复杂语句，并且有相同的函子和元数，但是根据定义子句3，复杂语句的每个对应参数都需要合一。在这个例子中，第一个参数，可以通过将X

初始化为marcellus而合一，但是X不能再初始为mia，这就是问题存在，所以Prolog认为两个复杂语句不能合一。

触发校验

合一是一个著名的概念，运用到了很多计算科学的分支中。它已经被深入地研究过，并且提出了许多有名的合一算法。但是Prolog中使用的合一算法不是标准的，而是其子集。

我们必须了解这点。思考如下的查询：

?- father(X) = X.

这两个语句能够合一吗？标准的合一算法会回答：不能。为什么？将X初始化为你选择的任意一个语句，比如，将X初始化为father(father(butch))，左边的部分变成

father(father(father(butch)))，右边的部分变成father(father(butch))，左右明显是不能合一的。而且更进一步，无论你将X初始化为什么样的值，以上的结果都是一致的。

无论如何选择，两个语句都不可能会相等，因为左边的部分始终都会比右边的部分多一层father。标准的合一算法会识别到这种情况（即我们后面提及的触发校验），停止下来，

并且告诉我们不能合一的结果。

Prolog中的递归定义不会这样做。因为右边语句是变量X，根据Prolog合一定义子句2的合一方式，X将会被初始化为左边的语句，即father(X)。但是由于有X在这个语句中，所以

X被初始化为father(X)，所以Prolog意识到father(X)其实是father(father(X))。但是里面还是有X，所以X又被初始化为father(X)，所以这个结果语句由变成了：

father(father(father(X)))，等等。通过将X初始为father(X)，Prolog进入一个无法停止的序列扩展中。

至少，这是一种理论。那么实际中呢？在早期的Prolog实现中，只是会客观地表述这种情况，可能会得到如下的结果：

```
Not enough memory to complete query!
```

或者是一个很长的字符串：

```
X = father(father(father(father
    (father(father(father(father
    (father(father(father
    (father(father(father
```

Prolog会拼命地返回正确的初始化值，但是这个运行无法停止，因为初始化的过程是没有边界的。从抽象的数学方面来说，Prolog的尝试是有意义的。直观上看，是两个语句能够

合一的唯一方式，就是将X初始化为无限嵌套的father函子。无限语句是有趣的数学抽象，但是我们不能实际使用。所以无论Prolog如何尝试，它都无法构建这个初始化结果。

现在，如果Prolog像上面那样耗尽内存是很烦人的，所以更高级的Prolog实现已经找到了一种更优雅的方式。请在SWI Prolog或者SICStus Prolog中尝试查询father(X) = X，

结果如下：

```
X = father(father(father(father(...))))
```

即，这种实现认为合一是可能的，但是不会陷入试图通过无限初始化X的陷阱中。替代方式是，Prolog会检查到这是一个可能的问题，然后停止，声明合一是可能的，同时给出

一个有限的结果表示无限的语句，比如：
father(father(father(father(...))))。（在我自己的机器上，查询的结果是：X = father(X)）

简而言之，father(X)和X是否合一的问题有三种不同的答案。标准合一算法认为不可能，较早的Prolog实现会拼命尝试从而耗尽内存，更加高级的Prolog实现会认为可以合一，

并且使用有限结果表示无限语句。这里没有“标准答案“，重要的是能够理解标准合一算法和Prolog合一的差异，理解当面临这类问题时，Prolog的实现方式。

这里更多地介绍一些标准合一算法和Prolog合一的差异。由于处理上述例子使用了很不同的方式，我们可能会认为标准合一算法和Prolog合一有很本质的差异。其实不然，它们

之间只有一个简单的差别，即标准合一算法，在进行两个语句合一的时候，第一步会进行触发校验：如果要求一个变量和一个语句进行合一，那么首先检查这个

变量是否在语句中

存在；如果存在，标准合一算法会认为这个合一是不能实现的，比如之前X和father(X)的例子；只有变量在语句中不存在，标准合一算法才会尝试进行合一操作。

换句话说，标准合一算法是悲观类型的。它会首先进行触发校验，只有肯定情况是安全的，才会尝试合一。所以标准合一算法不会因为无限初始化尝试而死锁。

Prolog合一，是另一种方式，是乐观类型的。它假设任何操作都不会存在危险。所以它取了标准合一算法的子集：移除了触发校验。当有两个语句需要合一，它就直接进行尝试。

因为Prolog是一种编程语言，所以这是一种明智的策略。合一是Prolog能够工作的基础，所以它需要快速地进行。每次合一前进行触发校验会明显地降低执行速度。悲观是安全的，

但是乐观会快很多！Prolog只有在面临类似X和father(X)合一时才会出现问题，但是现实中这种情况很少出现。

最后需要注意的是，Prolog有一个内置的谓词：unify_with_occur_check/2能够支持标准的合一算法，所以如果我们查询：

```
?- unify_with_occur_check(father(X), X).
```

Prolog会回答false。

使用合一编程

我们已经说过，合一是Prolog的一个基础操作。合一是Prolog证明查询树的核心，也是最为重要的。然而，随着学习Prolog的深入，合一本身也是有趣而且重要的，这点会越发

清晰。事实上，有时能够简单地通过使用复杂语句定义有趣的概念，从而构建有用的程序，合一经常能够将你需要的有用信息拉出来。

下面是一个这方面的简单例子，来自于Ivan Bratko。下面两行知识库定义了两个谓词，名字是vertical/1和horizontal/1，分别代表垂直的线条和水平的线条：

```
vertical(line(point(X, Y), point(X, Z))).

horizontal(line(point(X, Y), point(Z, Y))).
```

如果是第一眼扫过去，可能会觉得太简单了，没什么意思：它只包含了两个事实，没有规则。但是等等，两个事实都是使用复杂语句构建的，并且都有复杂语句作为参数。事实上，

在语句内部存在3层嵌套。而且，在最里层的参数都是变量，所以这个概念是通过通用方式定义的。也许它并不像想象中的那么简单，我们更深入的分析一下：

进入最底层，是一个复杂语句，函子是point，有两个参数。这两个参数应该都会使用数字初始化：point(X, Y)代表一个笛卡尔坐标的点。即，X代表从固定点到表示的点的水平距

离，Y代表从固定点到表示的点的垂直距离。

一旦我们确定了两个点，我们就确定了一条线，即两点之间的线。所以两个代表点的复杂语句组合起来，作为另外一个代表线条的复杂语句的参数。或者说，我们使用复杂语句去表

示一条线，其中包括了两个参数，这两个参数都是复杂语句，表示组成线的两个点。我们正是使用了Prolog的复杂语句能力去表示了具有层次的事物的概念。

垂直或者水平，都是线条的特征。谓词vertical和 horizontal都有一个“线条”的参数。vertical/1的定义可以简单地理解：两点之间的线条，如果两点具有相同的X坐标，那么线

条就是垂直的。注意我们是如何使“具有相同的X坐标”在Prolog中起作用的，我们使用了同一个变量X作为代表坐标的复杂语句的第一个参数。

同理，horizontal/1的定义可以简单地理解：两点之间的线条，如果两点具有相同的Y坐标，那么线条就是水平的，我们使用了同一个变量Y作为代表坐标的复杂语句的第二个参数。

那么我们能够使用这个知识库做什么？看下面的例子：

```
?- vertical(line(point(1,1), point(1,3))).
```

Prolog会回答true，因为这个查询和vertical/1的定义合一。同理，如果我们查询：

```
?- vertical(line(point(1,1), point(3,2))).
```

Prolog会回答false，因为这个查询和vertical/1的定义无法合一。但是如果我们更通用地问：

```
horizontal(line(point(1,1), point(2, Y))).
```

Prolog会回答：Y = 1; false。这个查询是说，如果要找到一条水平的线，坐标1在（1,1），坐标2的横坐标是2，那么坐标2的纵坐标应该是多少？Prolog正确地告诉我们结果：

纵坐标是1，没有其他可能值（分号后面是false，表示没有其他值了）。

现在考虑下面的查询：

```
?- horizontal(line(point(2,3), P)).
```

在我的机器上，SWI-Prolog的回答是：

```
P = point(_G452, 3).
```

这个查询是问：如果是一条水平的线，它的坐标1是（2,3），那么它的坐标2是什么样的？答案是：任意纵坐标为3的坐标都可以。注意答案中的point的第一个参数_G452，这是

Prolog表达任意值的方式。

备注：我们最后一个查询例子的答案，point(_G452, 3)是一个被结构化的答案。即，这个答案是一个复杂语句，代表了复杂概念（任意纵坐标为3的点）。这个结构化的答案只是

使用了合一，没有逻辑相关（没有使用假言推理）的应用去构建它。通过合一构建结构化的答案在Prolog编程中是一个强大的思想，远比上面的简单例子的要强大。而且，如果程序是

使用大量合一写出的，会十分地高效。我们将会在第7章讨论不同lists的时候看到更优雅的例子。

这种编程的形式在构建有层次的结构时特别有用，我们可以使用复杂语句去表示这些结构，通过合一访问它们。这种方式在计算机语言学中特别有用，因为语言本身就是有层次的结

构：想象一下句子可以分解为名词词组和动词词组，动词词组有可以分解为代词和名词，等等。

分类: Prolog

标签: 合一

« 上一篇: Learn Prolog Now 翻译 - 第一章 - 事实，规则和查询 - 第三节， 练习题和答案

» 下一篇: Learn Prolog Now 翻译 - 第二章 - 合一和证明搜索 - 第二节， 证明搜索

posted on 2015-07-01 10:00 seaman.kingfall 阅读(660) 评论(0) 编辑 收藏

相关博文:

· Learn Prolog Now 翻译 - 第十章 - 中断和否定 - 第一节， 中断
· Learn Prolog Now 翻译 - 第六章 - 列表补遗 - 第一节，列表合并
· Learn Prolog Now 翻译 - 第三章 - 递归 - 第一节，递归的定义
· Learn Prolog Now 翻译 - 第二章 - 合一和证明搜索 - 第二节， 证明搜索
· Learn Prolog Now 翻译 - 第四章 - 列表 - 第一节，列表定义和使用

最新新闻:

· 一线｜“美团配送”品牌发布：对外开放配送平台 共享配送能力