

Seaman.h.zhang

博客园 :: 首页 :: 新随笔 :: 联系 :: 订阅  :: 管理 34 Posts :: 0 Stories :: 2 Comments :: 0 Trackbacks

公告

昵称: seaman.kingfall
园龄: 4年3个月
粉丝: 4
关注: 1
[+加关注](#)

搜索

常用链接

[我的随笔](#)
[我的评论](#)
[我的参与](#)
[最新评论](#)
[我的标签](#)

我的标签

[练习题\(6\)](#)
[合一\(3\)](#)
[递归\(3\)](#)
[中断\(2\)](#)
[类型变量\(2\)](#)
[数字\(2\)](#)
[列表\(2\)](#)
[Haskell\(2\)](#)
[recursive\(2\)](#)
[比较\(2\)](#)
[更多](#)

随笔分类

[Haskell\(2\)](#)
[Prolog\(32\)](#)

随笔档案

[2015年8月 \(7\)](#)
[2015年7月 \(22\)](#)
[2015年6月 \(5\)](#)

最新评论

1. Re:Learn Prolog Now
翻译 - 第一章 - 事实, 规则
和查询 - 第一节, 一些简单
的例子
学习!
--深蓝医生
2. Re:Learn Prolog Now
翻译 - 第一章 - 事实, 规则
和查询 - 第一节, 一些简单
的例子
翻译了这么多了, 而且每天一
篇, 不能望其项背啊。

Learn Prolog Now 翻译 - 第三章 - 递归 - 第二节, 规则顺序, 目标顺序, 终止

内容提要

[规则顺序](#)

[目标顺序](#)

[终止](#)

Prolog是第一门比较成功的逻辑编程语言。逻辑编程语言内在实现是简单和富有魅力的: 程序员的工作简单地说就是描述问题; 程序员应该写下(使用语言的逻辑)声明性的规格说明

(即, 一个知识库), 去描述有趣的状态、事实和关系; 程序员不应该告诉计算机如何去实现, 而他根据问一些问题去获取信息, 逻辑编程语言会给出答案。

然而, 以上是理想情况, Prolog本身也确实通过一些重要的特征, 往这个方向在努力。但是Prolog不是, 重复一次, 不是一门完整的逻辑编程语言。如果你只是从声明性方面去思考

Prolog程序, 那么实际使用上去就会十分困难。正如我们之前章节学习到的, Prolog通过特有的方式得出查询的结果: 它会自上而下地搜索知识库, 从左到右地匹配每个子句的目标, 并且

通过回溯从错误选择中进行恢复。这些程序性的方面对你的查询实际如何进行有很重要的影响。我们已经看过了一些例子在其声明性和程序性上不匹配(记得 `p :- p`吗?), 接下来, 我们

会继续看到, Prolog中很容易定义逻辑上相同的, 但是实现上却十分不同的程序。让我们思考如下的情况。

请回忆之前我们定义的“后辈”程序, 这里我们称为descend1.pl:



```
child(anne, bridget).  
child(bridget, caroline).  
child(caroline, donna).  
child(donna, emily).  
  
descend(X, Y) :- child(X, Y).  
descend(X, Y) :- child(X, Z), descend(Z, Y).
```



这里我们做一个调整, 并称新的程序为descend2.pl:



```
child(anne, bridget).  
child(bridget, caroline).
```

--Benjamin Yan

阅读排行榜

1. Learn Prolog Now 翻译 - 第三章 - 递归 - 第一节, 递归的定义(1168)
2. Learn Prolog Now 翻译 - 第一章 - 事实, 规则和查询 - 第一节, 一些简单的例子(1087)
3. Learn Prolog Now 翻译 - 第一章 - 事实, 规则和查询 - 第二节, Prolog语法介绍(781)
4. Haskell学习笔记二: 自定义类型(767)
5. Learn Prolog Now 翻译 - 第六章 - 列表补遗 - 第一节, 列表合并(753)

评论排行榜

1. Learn Prolog Now 翻译 - 第一章 - 事实, 规则和查询 - 第一节, 一些简单的例子(2)

推荐排行榜

1. Haskell学习笔记二: 自定义类型(1)
2. Learn Prolog Now 翻译 - 第三章 - 递归 - 第四节, 更多的实践和练习(1)

```
child(caroline, donna).  
child(donna, emily).
```

```
descend(X, Y) :- child(X, Z), descend(Z, Y).  
descend(X, Y) :- child(X, Y).
```



这里的修改只是换了一些两个规则的顺序。所以如果只是从纯粹的逻辑定义上去理解, 是什么都没有改变的。但是这种改变带给程序性上有什么不同吗? 是的, 但不明显。

比如, 如果你查询所有的情况, 将会看到descend1.pl的第一个回答是:

```
X = anne
```

```
Y = bridget
```

然而descend2.pl的第一个回答是:

```
X = anne
```

```
Y = emily
```

但是两个程序生成的答案是相同的, 只是顺序不一致。这是具有共性的。简要说, 改变Prolog程序中规则的顺序, 不会改变程序的行为。

我们继续, 在descend2.pl的基础上, 再进行一点小的修改, 变成descend3.pl:



```
child(anne, bridget).  
child(bridget, caroline).  
child(caroline, donna).  
child(donna, emily).
```

```
descend(X, Y) :- descend(Z, Y), child(X, Z).  
descend(X, Y) :- child(X, Y).
```



请注意不同之处。这里我们对一个规则中的目标顺序进行了调整, 而并非调整规则的顺序。现在, 如果我们纯粹是从逻辑定义方面去理解, 并没有任何不同, 这和之前的两个定义

是一样的含义。但是这个程序的行为已经彻底改变。比如, 如果我们进行查询:

```
?- descend(anne, emily).
```

Prolog会报错(类似“Out of local stack”)。Prolog进入了死循环, 为什么? 为了满足查询descend(anne, emily), Prolog会使用第一个规则。这意味着下一个目的是满足查询:

```
descend(W1, emily).
```

这里引入了新的变量W1。但是为了满足这个新目标, Prolog又会使用第一个规则, 这意味着下一个目标会是: descend(W2, emily), 这里引入了新的变量W2。当然, 就会循环引入下

一个新的目标`descend(W3, emily)`，接下去又是`descend(W4, emily)`，等等。即，目标顺序的改变导致了程序的崩溃。使用标准的术语，这里我们有一个经典的关于左递归规则的例子，

即一个规则的主干部分最左端的目标是和规则的头部一样的。正如我们的例子所示，这种规则会导致非终止的计算。目标顺序，特别是左递归，当其不能终止时，就会变成一切罪恶之源。

还有，这里有一个针对规则顺序的提醒。我们之前提及规则顺序的改变，只会影响其查询的结果的顺序。但是这个结论在非终止程序中是不适用的。为了说明这点，请参考关于“后辈”

代码的第四次修改，称为`descend4.pl`：



```
child(anne, bridget).  
child(bridget, caroline).  
child(caroline, donna).  
child(donna, emily).  
  
descend(X, Y) :- child(X, Y).  
descend(X, Y) :- descend(Z, Y), child(X, Z).
```



这个程序只是在`descend3.pl`的基础上，调整了规则的顺序。现在这个程序和其他之前的程序具有一样的声明性含义，但是程序性上有所差别。首先，很明显的是，和`descend1.pl`和

`descend2.pl`有明显的差别，因为`descend4.pl`包含了左递归的规则，它会在进行一些查询时无法终止计算。比如，我们如果进行下面的查询，将无法终止计算：

```
?- descend(anne, emily).
```

但是`descend4.pl`在程序性上和`descend3.pl`也有所不同。规则顺序的不同导致了这种差异性。比如，`descend3.pl`在进行查询：

```
?- descend(anne, bridget).
```

时不会终止；但是`descend4.pl`在这个查询中会有结果。因为它会首先使用非递归的规则，并且找到答案，终止计算。所以在非终止的程序中，规则顺序的改变会导致找到一些额外的

解决方案。但无论如何，目标顺序的改变，而非规则顺序的改变，会使得程序性完全不同。为了确保计算能够终止，我们必须注意规则主干部分的目标顺序。因为调整规则的顺序，不会

改变非终止程序的本质——最多可以找到一些额外的解决方案而已。

总结一下，以上四个关于“后辈”程序的变种，描述了同样的问题，但是具体实现上有所不同。`descend1.pl`和`descend2.pl`在实现上的不同相对来说比较小：它们会生成相同的解决方案，

但是顺序不同。然而`descend3.pl`和`descend4.pl`在程序性上的差异和之前两个更大，因为它们的规则中目标的顺序不同。具体而言，这两个版本都包含了左递归规则，都会导致非终止的计算

行为。descend3.pl和descend4.pl在规则顺序上有所不同, 意味着在某些情况下, descend4.pl可以终止计算, 但是descend3.pl不能。

那么我们如何构建有用的Prolog程序呢? 通常你首先需要通过声明性思考去确定整体的想法(蓝图), 即思考如何精确地描述问题。这是解决问题的优先方式, 同时也是逻辑编程的灵魂。

但是一旦你完成了这部分工作, 就必须结合Prolog的具体实现检查你的方案。特别是需要检查规则中目标的顺序, 从而确保计算能够终止。规则绝不要写出规则的主干最左边目标和规则头部

相同的情况, 而是应该将触发递归的目标写到主干的最右边, 即让递归目标出现在所有非递归目标的后面。这样做会使得Prolog有最多的机会不通过递归就找到答案。

分类: Prolog

标签: 递归, recursive, 规则顺序, 目标顺序, 终止



seaman.kingfall

关注 - 1

粉丝 - 4

+加关注

0

0

« 上一篇: Learn Prolog Now 翻译 - 第三章 - 递归 - 第一节, 递归的定义

» 下一篇: Learn Prolog Now 翻译 - 第三章 - 递归 - 第三节, 练习题和答案

posted on 2015-07-07 15:40 seaman.kingfall 阅读(536) 评论(0) 编辑 收藏
刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论, 请 [登录](#) 或 [注册](#), [访问网站首页](#)。

【推荐】超50万C++/C#源码: 大型实时仿真组态图形源码

【活动】看雪2019安全开发者峰会, 共话安全领域焦点

【培训】Java程序员年薪40W, 他1年走了别人5年的路

相关博文:

· Learn Prolog Now 翻译 - 第六章 - 列表补遗 - 第二节, 列表反转

· Learn Prolog Now 翻译 - 第一章 - 事实, 规则和查询 - 第二节, Prolog语法介绍

- [Learn Prolog Now 翻译 - 第三章 - 递归 - 第一节, 递归的定义](#)
- [Learn Prolog Now 翻译 - 第三章 - 递归 - 第四节, 更多的实践和练习](#)
- [Learn Prolog Now 翻译 - 第三章 - 递归 - 第三节, 练习题和答案](#)

最新新闻:

- [知否 | 太空垃圾如何清理? 卫星测试用鱼叉击中太空垃圾碎片](#)
 - [一线 | “美团配送”品牌发布: 对外开放配送平台 共享配送能力](#)
 - [苍蝇落在食物上会发生什么? 让我们说的仔细一点](#)
 - [科学家研究板块构造变化对海洋含氧量影响](#)
 - [日本程序员节假日全员加班? 都是“令和”惹的祸](#)
- » [更多新闻...](#)

Copyright © seaman.kingfall
Powered by: .Text and ASP.NET
Theme by: .NET Monster