

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

找找看

谷歌搜索

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 - 第十章 - 中断和否定 - 第三节, 使用否定作为失败判定

Prolog一个很有用的特征就是可以让使用者概括地描述事物, 对其进行抽象。比如我们如果想描述Vincent喜欢汉堡, 可以这么写:

```
enjoys(vincent, X) :- burger(X).
```

但是在现实中总会存在例外。也许Vincent不喜欢Big Kahuna汉堡。即, 正确的规则是: Vincent喜欢汉堡, 除了Big Kahuna汉堡。好了, 我们如何在Prolog中描述呢?

作为第一步, 先介绍另一个Prolog内置的谓词: fail/0。正如它的名字所示, fail/0是一个当Prolog运行到这个目标时会立即失败的特殊标识。这可能听上去没有什么用处, 但是请记住, 如果Prolog失败了, 它会尝试回溯。所以fail/0可以看作一个强制回溯的指令。而且在和中断一起使用时, 由于中断会阻止回溯, 这会让我们写出一些有趣的程序, 特别是, 它可以定义通用规则中的一些异常和特殊的情况。

思考下面的代码:

```
enjoys(vincent, X) :- big_kahuna_burger(X), !, fail.
enjoys(vincent, X) :- burger(X).

burger(X) :- big_mac(X).
burger(X) :- big_kahuna_burger(X).
burger(X) :- whopper(X).

big_mac(a).
big_kahuna_burger(b).
big_mac(c).
whopper(d).
```

前两行代码描述了Vincent的喜好。后六行代码描述了汉堡的类型和具体四个汉堡: a, b, c, d。假设前两行真实地描述了Vincent的喜好(即, 他喜好所有的汉堡除了Big Kahuna汉堡), 那么他应该喜好汉堡a, c和d, 但没有b。事实上, 这是正确的:

```
?- enjoys(vincent, a).
true

?- enjoys(vincent, b).
false

?- enjoys(vincent, c).
true

?- enjoys(vincent, d).
true
```

这是如何起作用的? 关键在于第一行代码中!和fail/0的组合使用(这个甚至有一个名字: 称为“中断-失败”组合)。当我们进行查询enjoys(vincent, b)时, 会首先使用第一个规则, 然后到达中断。这会提交我们已经做出的选择, 而且特别

--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)

需要说明的是, 会阻止使用(回溯)第二个规则。但是随后就会到达fail/0,。这会强制尝试回溯, 但是中断阻止了回溯, 所以查询会失败。

这很有趣, 但是不够理想。首先, 注意规则的顺序是关键: 如果我们调换了前两行的顺序, 就得不到想要的结果了。类似地, 中断也是关键: 如果我们移除它, 程序也不会按照相同的方式运行(所以这是一个红色中断)。简而言之, 我们得到的是两个互相依赖的子句。从这个例子出发, 如果我们能够从中提取一些有用的部分, 并且包装为更健壮的通用方式会更好。

确实可以这么做。关键点在于第一个子句是从本质上描述了Vincent不喜欢X如果X是一个Big Kahuna汉堡。即, “中断-失败”组合看上去就是某种形式的__否定__。事实上, 这就是关键的抽象: “中断-失败”组合让我们定义了某种形式的否定称为: 使用否定作为失败判定。下面是实现:

```
neg(Goal) :- Goal, !, fail.
neg(Goal) .
```

对于任意Prolog目标, neg(Goal)将会为真当Goal为假时。

使用新定义的谓词neg/1, 我们可以更清晰地描述Vincent的喜好:

```
enjoys(vincent, X) :- burger(X), neg(big_kahuna_burger(X)) .
```

即, Vincent喜欢X如果X是一个汉堡而且X不是Big Kahuna汉堡。这很接近我们原始的描述: Vicent喜欢汉堡, 除了Big Kahuna汉堡。

使用否定作为失败判定是一个重要的工具。不仅仅在于它提供了有用的表述性(描述异常情况的能力), 更在于它提供了相对安全的方式。通过使用否定进行失败判定(而不是低层次的中断-失败组合形式), 我们可以更好地进行失败判定, 避免使用红色中断而导致的一些错误。事实上, 否定作为失败判定十分的有用, 以至于成为了标准Prolog的内置实现, 所以我们不用再定义它了。在标准的Prolog实现中, 操作符+就是否定作为失败判定, 所以我们可以重新定义Vincent的喜好:

```
enjoys(vincent, X) :- burger(X), \+ big_kahuan_burger(X) .
```

但是, 有一些使用否定作为失败判定的建议: 不要认为否定作为失败判定就是逻辑否。它并不是, 思考下面的“汉堡”世界:

```
burger(X) :- big_mac(X) .
burger(X) :- big_kahuna_burger(X) .
burger(X) :- whopper(X) .

big_mac(a) .
big_kahuna_burger(b) .
big_mac(c) .
whopper(d) .
```

如果我们进行查询enjoys(vincent, X), 得到正确的回答:

```
?- enjoys(vincent, X) .
X = a;
X = c;
X = d;
false
```

但是假设我们重写了第一行代码实现:

```
enjoys(vincent, X) :- \+ big_kahuna_burger(X), burger(X).
```

注意从声明性来看, 这里没有什么不同: 毕竟, `burger(X)`和不是`big kahuna burger(X)`在逻辑上等同于: 不是`big kahuna burger(X)`和`burger(X)`。然而, 下面是我们进行相同的查询得到的结果:

```
?- enjoys(vincent, X).  
false
```

发生了什么? 在更新后的知识库中, Prolog首先会判断`+ big_kahuna_burger(X)`是否成立, 这意味着必须证明`big_kahuna_burger(X)`失败。但是这是能够成功的。因为, 知识库中有包含`big_kahuna_burger(b)`这个事实。所以查询`+ big_kahuna_burger(X)`会失败, 同时导致原始查询也会失败。在内核中, 两个程序关键的不同在于原来的版本(能够正常工作的)中, 我们在将变量`X`初始化后再使用的`+`, 在新的版本中, 我们在变量初始化前就使用了`+`, 这就是关键的不同。

总结一下, 使用否定作为失败判定并不等于逻辑否定, 我们必须理解其程序维度上的含义。然而, 这是一个重要的编程思路: 通常情况下, 使用否定作为失败判定会优于直接使用红色中断的程序。但是, “通常”并不意味着“总是”。有些特殊的时候, 使用红色中断会更好一些。

比如, 假设我们需要写出代码如下逻辑: 如果`a`和`b`都成立, 或者`a`不成立但是`c`成立, 那么`p`成立。在否定作为失败判定的帮助下, 我们能够写出的代码如下:

```
p :- a, b.  
p :- \+ a, c.
```

但是设想如果`a`是一个很复杂的目标, 需要很长时间的计算。上面这样的程序意味着我们需要计算`a`两次, 这通常会导致不能接受的性能问题。如果是那样, 可能使用如下的程序会更好:

```
p :- a, !, b.  
p :- c.
```

注意这里是一个红色中断: 移除它会改变程序的行为。

关于否定作为失败判定的介绍到此为止, 这里没有普遍适用的原则可以覆盖到所有的情况。编程更像是科学的艺术: 这使得它更加有趣。你需要尽可能熟悉你学习的语言的一切(无论是Prolog, Java, Perl还是其他的任何语言), 理解需要解决的问题, 找到合适的解决方案。然后: 尽你所能地尝试和完善!

分类: Prolog

标签: 中断, 否定, 失败判定

好文要顶

关注我

收藏该文



seaman.kingfall

关注 - 1

粉丝 - 4

+加关注

0

0

« 上一篇: Learn Prolog Now 翻译 - 第十章 - 中断和否定 - 第二节, 中断的运用

» 下一篇: Learn Prolog Now 翻译 - 第十一章 - 知识库相关操作和解决方案的收集 - 第一节, 知识库相关操作

posted on 2015-08-05 13:29 seaman.kingfall 阅读(359) 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:

- [编程和否定](#)
- [Learn Prolog Now 翻译 - 第十章 - 中断和否定 - 第二节, 中断的运用](#)
- [否定之否定](#)
- [Learn Prolog Now 翻译 - 第十章 - 中断和否定 - 第一节, 中断](#)
- [长尾理论否定之否定](#)

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