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

内容提要：

append/3存在性能问题

列表反转原始版本

列表反转高效版本

append/3性能问题

谓词append/3十分有用，而且了解如果使用它搭建应用也很重要。但是同样重要的是，我们应该知道它可能是低效的源头，并且不是任何使用都想要使用它。

为什么append/3可能是低效的源头？如果回想它的工作方式，你就会注意到其缺点：append/3不是通过简单方式将列表连接在一起的，而是，需要将第一个列表遍历完，再级联地

找到最终的结果。

这很多情况下不会造成什么问题。比如，如果我们有两个列表希望合并，可能情况并不太糟。当然，Prolog将会遍历完第一个列表的所有元素，但是如果第一个列表不是太长，使用

append/3就不会有太高的代价。

但是如果前两个参数是变量的话，情况就会很不同。正如我们之前看到的，将append/3的前两个参数传入变量也是十分有用的，因为这样可以让Prolog搜索列表拆分的方式。但是这

是有代价的：会进行很多的搜索，会导致非常严重的性能问题。

为了展示这种情况，我们将会验证反转列表的性能问题。即，定义一个谓词，接受一个输入的列表（比如，[a, b, c, d]），返回一个包含相同元素但是顺序不同的列表（比如，

[d, c, b, a]）。一个反转列表的谓词是很有用的。正如我们已经学习过的，在Prolog中，从前端访问列表是十分容易的。比如，为了获取列表L的头元素，我们只需要进行合一操作，

[H|_] = L即可，结果就是将L的头元素作为H的初始化值。但是要获取一个列表的最后一个元素就十分困难：我们不能够简单地通过合一来完成了。从另外一个角度来说，如果我们有一个

谓词可以反转列表，我们可以首先将输入列表进行反转，然后再获取反转后的列表的头元素，这就会得到原始列表的最后一个元素。所以反转列表谓词是一个有用的工具。然而，如果我们

必须要反转一个很大的列表，我们就希望反转的谓词十分高效。所以我们需要仔细地思考这个问题。

下面就是我们马上要进行的工作，我们将会定义两个反转谓词：一个使用append/3的原始版本；一个使用累加器高效版本（实际上也是更自然的版本）。

列表反转的原始版本

如下是反转列表的递归定义：

1. 如果我们反转一个空列表，就会得到一个空列表。

2. 如果我们反转列表[H|T]，我们通过反转T，并且将其和[H]进行合并来完成。

为了验证递归子句是正确的，请思考列表[a, b, c, d]，如果我们将其尾部进行反转，可以得到[d, c, b]，将其和[a]合并，得到[d, c, b, a]，就是[a, b, c, d]的反转结果。

借助append/3可以轻松地实现这个递归定义：

```
naiverev([], []).

naiverev([H|T], R) :- naiverev(T, RevT), append(RevT, [H], R).
```

现在，这个定义是正确无误的，但是它完成工作的方式很低效。可以通过追踪程序运行来观察。可以看到这个程序会花费大量的时间在append上，这个不必太惊讶：毕竟，我们是

在递归地调用append/3。得出结果是十分低效的（如果运行了追踪，你将会发现一个8个元素的列表，大概有90步左右）并且难以理解的（因为谓词中花费了大量时间在递归调用append/3

，使得难以理解具体进行到了哪里）。

这个原始版本确实不太好。不过我们将会看到，有更好的解决方案。

列表反转的高效版本

更好的解决方案是使用累加器。其核心思想很简单和自然。累加器是一个列表，当开始时候为空。假设我们想要反转[a, b, c, d]，开始的时候，累加器是[]。所以我们简单地将我们

希望反转列表的头部取出，加入到累加器中去。然后我们继续处理列表尾部，即我们面临反转列表，[b, c, d]，这个时候累加器为[a]。类似地，我们取出列表的头部，并将其插入到累加器

列表的头部（这样累加器就变为，[b, a]），同时继续反转列表[c, d]。然后继续使用相同的方式，所以又可以得到新的累加器[c, b, a]和反转列表[d]；然后下一步会得到累加器[d, c, b, a]和

反转列表[]。这时到了处理结束点：累加器保存的列表就是反转过的列表。总结一下：思路是从头遍历我们想要反转的列表，将每一次取出的头元素插入累加器的头部，像下面这样：

```
List: [a, b, c, d]          Accumulator: []

List: [b, c, d]             Accumulator: [a]
```

List: [c, d] Accumulator: [b, a]

List: [d] Accumulator: [c, b, a]

List: [] Accumulator: [d, c, b, a]

因为我们只是简单地遍历了一次列表，所以这个方法是高效的，我们没有浪费时间在计算级联中间变量的结果和其他无关的工作上。

在Prolog上实现也很容易，下面就是使用累加器的代码：

```
accRev([H|T], A, R) :- accRev(T, [H|A], R).

accRev([], A, A).
```

这是一段经典的累加器代码：它遵循了之前我们介绍数字运算时使用累加器的模式，递归子句负责将输入列表的头部取出，然后压入累加器列表中；基础子句停止递归，将结果从

累加器拷贝到最后一个参数中。

正如累加器代码的常规使用方式一样，写出一个谓词调用累加器谓词，并且给出初始化值是一个很好的实践：

```
rev(L, R) :- accRev(L, [], R).
```

同样可以开启追踪程序进行观察，以便可以和naiverev/2进行对比。累加器版本明显性能更好。比如，8个元素的列表反转，它只使用了20步就能完成，对比之前原始版本的90步。

而且，追踪的结果也更容易理解。累加器版本的内在原理比原始版本更简单，也更自然。

总结一下，append/3是一个有用的谓词，我们不应该惧怕使用它。而且，应该意识到当我们使用它时，有可能是低效的源头，所以我们应该寻找更好的方式。通常的情况是，使用

累加器是一个更好的方式，并且累加器是处理列表操作很自然的选择。

分类: Prolog

标签: 列表反转, reverse

seaman.kingfall
关注 - 1
粉丝 - 4
+加关注

« 上一篇: Learn Prolog Now 翻译 - 第六章 - 列表补遗 - 第一节，列表合并
» 下一篇: Learn Prolog Now 翻译 - 第六章 - 列表补遗 - 第三节， 练习题和答案

posted on 2015-07-21 16:35 seaman.kingfall 阅读(444) 评论(0) 编辑 收藏