

# 第 6 讲 表示不变式和抽象函数

## 6.1 介绍

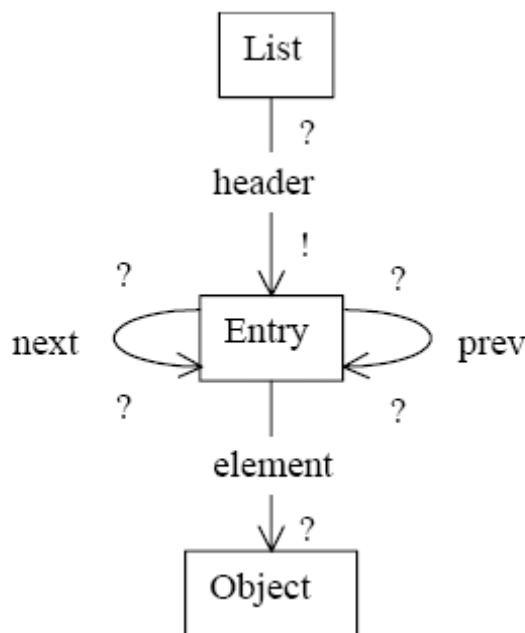
本讲中，我们描述了两个用于理解抽象数据类型的工具：表示不变式（representation invariant）和抽象函数（abstraction function）。表示不变式描述了一个类型的实例是否是合适，抽象函数告诉我们如何去解释它。表示不变式可以加强测试能力，在没有理解抽象函数的情况下编写一个抽象类或者修改它是不可能的。写下它是非常有用的，尤其是对于维护者，在非常（tricky）情况下也是很重要的。

## 6.2 什么是表示不变式

从表示的角度看，一个表示不变式，或者简称 rep invariant，是判断一个类型实例是否是合适的约束。从数学上看，它是一个实例表示的公式，你可以把它看成一个函数，它获得抽象类型并依赖于它们是否是合适而返回正确或错误：

$$RI: Object \rightarrow Boolean$$

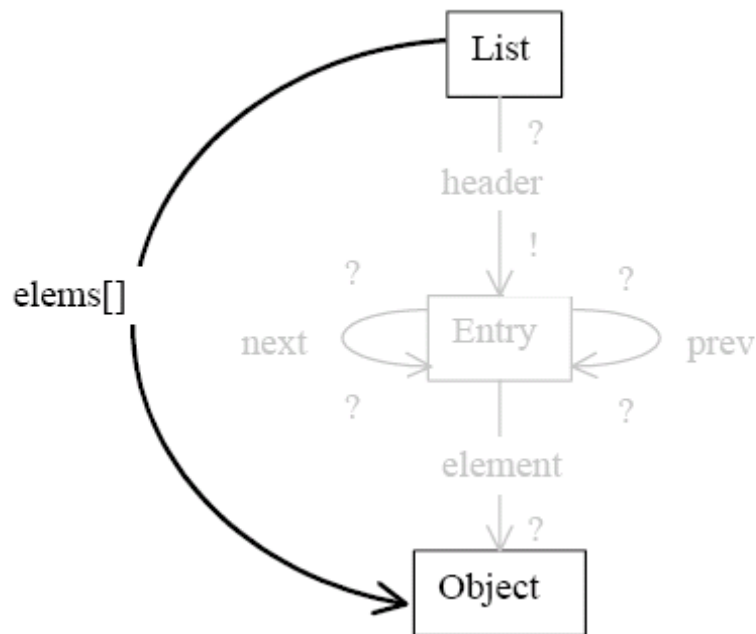
考虑上次课我们讨论的链式结构的列表实现。这是它的对象模型：



*LinkedList* 类有一个属性，*header*，它保持对 *Entry* 类对象的一个引用.这个对象有三个属性：*element*，保持对列表对象的引用；*prev*，指向列表的上一个元素；*next*，指向下一个元素。

这个对象模型显示了数据类型的表示，正如以前我们所提到的，对象模型可以在抽象的不同等级绘出。从列表用户的角度看，可能省略 *Entry* 方框，而仅仅显示 *List* 到 *Object* 的一个规格说明属性。这个图以黑色表示对象模型，以灰色显示了隐藏的表示（*Entry* 和它的进

入弧线和出去弧线):



表示不变式是判断一个类型实例是否是合适的约束。我们的对象模型已经给出了一些关于它的属性:

- 例如, 它显示的 *header* 属性保持了对 *Entry* 类对象的一个引用。这个属性非常重要, 但并不是那么很吸引人的注意, 因为属性具有那个类型; 这种属性对于多态性容器的内容更有兴趣, 如向量, 它的元素类型不可能表示成源代码。
- *header* 箭头的目标端的多重性标记 **!** 表明 *header* 属性不能为空。( **!** 符号表示恰好一个。)
- *prev* 和 *next* 箭头的目标端的多重性, **?** 表明他们至多可以指向一个条目。( **?** 符号表示 0 或者 1。)
- *prev* 和 *next* 箭头的源端的多重性, **?** 表明他们至多可以指向其它一个条目的 *next* 属性和其它一个条目的 *prev* 属性。
- 元素属性的目标端的多重性, **?** 表示每个 *Entry* 最多指向一个对象。

表示不变式的属性没有显示在图形对象模型中。

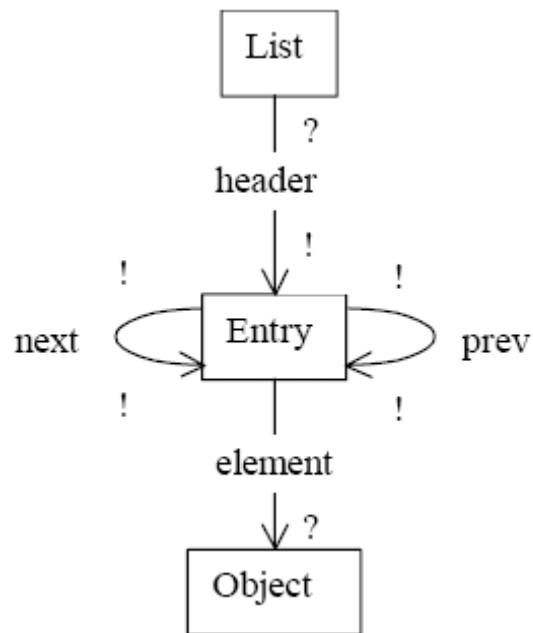
也有一些属性不显示出来:

- 当列表中有两个条目 *e1* 和 *e2* 时, 如果 *e1.next=e2*, 那么 *e2.prev=e1*。
- 列表前面的虚拟条目有一个空元素属性。

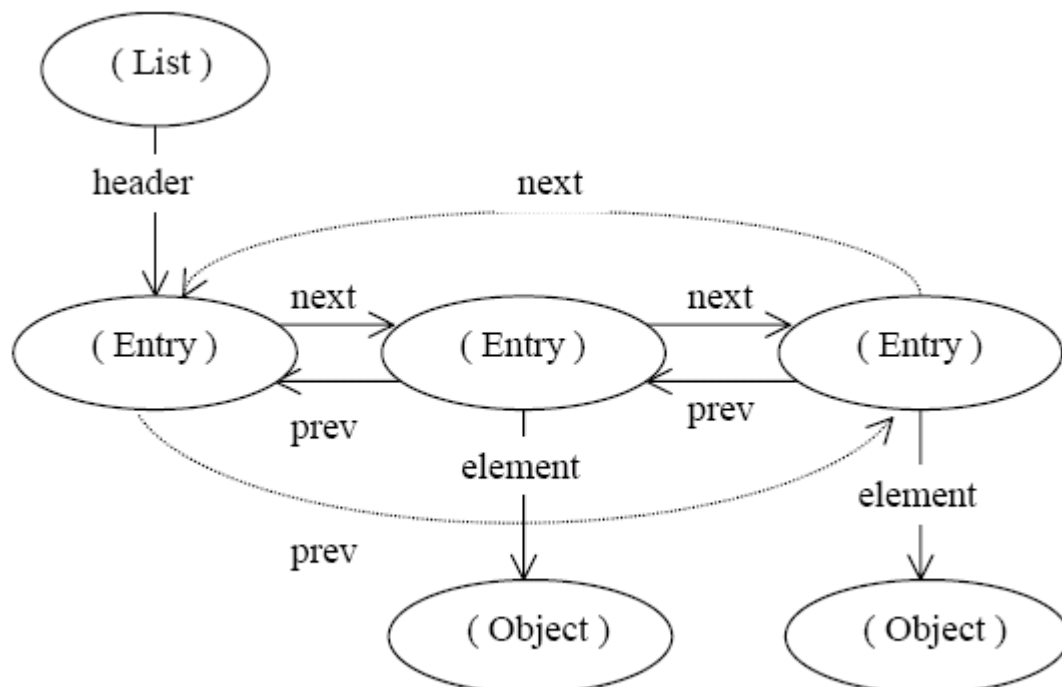
因为对象模型仅仅显示了对象, 而不是原始值。

*LinkedList* 的表示有一个属性 *size* 记录列表的大小。一个表示不变式的属性就是指 *size* 等于列表表示的条目的数目减一 (因为第一个条目是虚拟的)。

实际上，在 Java 的实现 `java.util.LinkedList` 中，对象模型有一个额外的约束反映在表示不变式中。每个条目有一个非空的 `next` 和 `prev`。



注意那个较强的 `next` 和 `prev` 箭头的多重性。这是一个两个元素的列表的例子（当然有三个条目，包括一个虚拟的）：



当检查一个表示不变式时，不仅要注意显示出来的是什么约束，而且要注意哪些丢失了。在这种情况下，没有要求元素的属性为非空或元素不能共享。这是我们所期望的：它允许一

个列表包括空引用，或在多个位置包括相同的对象。

让我们非正式地总结一下我们的表示不变式：

对于类 *LinkedList* 的每个实例

*header* 属性非空

*header* 属性有一个空元素属性

有 (*size*+1) 个条目

条目从 *header* 条目开始到 *header* 条目结束形成了一个环

对于每个条目，*prev* 和 *next* 将返回一个条目

我们也可以稍微正式一点表示：

*all p:LinkedList |*

*p.header!=null*

*&& p.header.element=null*

*&& p.size+1=|p.header.\*next|*

*&& p.header=p.header.next<sup>p.size+1</sup>*

*&& all e in p.header.\*next|e.prev.next=e*

为了理解这个公式，你需要了解如下：

对于任意表达式 *e* 表示对象的某个集合，任意属性 *f*，*e.f* 表示你获得的对象集，如果 *f* 跟在 *e* 的每个对象的后面；*e.\*f* 表示对象集，如果 *f* 跟在 *e* 的每个对象的后面若干次的话；*|e|* 指对象的数目。

因此 *p.header.\*next* 表示列表中所有条目的集合，因为你通过列表 *p* 的 *header* 属性以及 *next* 属性的任意次获得了它。

使这个公式变得非常清楚的就是表示不变式是关于一个单个的链式列表 *p*。另一个写出不变式的好方法是：

*R(p)=*

*p.header!=null*

*&& p.header.element=null*

*&& p.size+1=|p.header.\*next|*

*&& p.header=p.header.next<sup>p.size+1</sup>*

*&& all e in p.header.\*next | e.prev.next=e*

其中，我们把不变式看成一个布尔函数。这是当我们把不变式转换成运行时断言时所持有的观点。

不变式的选择可以对编码实现抽象类型以及如何更好地执行产生重要的影响。假定我们通过要求所有条目的 *element* 属性而不是 *header* 为非空来加强我们的不变式。这就允许我们通过比较元素是否为空来检测 *header* 条目；采用当前的不变式，需要浏览列表的操作必须对条目计数而不是比较 *header* 属性。假如我们在 *next* 和 *prev* 指针上弱化了不变式，允许头端的 *prev* 和尾端的 *next* 有任何值。这将导致需要在头端和尾端对条目进行特殊处理，导致代码的不统一。要求头端的 *prev* 和尾端的 *next* 都为空不会起到任何帮助。

## 6.3 归纳推理

表示不变式使得模块推理变成可能。为了检查是否一个操作实现正确，我们不需要查看其它方法，而只需看看*归纳*的原则。我们确保每个构造函数创建了一个对象来满足不变式，每个 *mutator* 和 *producer* 保存了不变式；也就是说，如果给定的对象满足它，那么它产生一个对象也会满足它。现在我们可以认为类型的每个对象满足表示不变式，因为它必被一个构造函数和一系列 *mutator* 或 *producer* 应用产生。

为了看看这个如何工作，让我们看看 *LinkedList* 类的一些操作。表示以 Java 声明如下：

```
public class LinkedList {
    Entry header;
    int size;
    class Entry {
        Object element;
        Entry prev;
        Entry next;
        Entry (Object e, Entry p, Entry n) {element=e; prev=p; next=n; }
    }
    ...
}
```

这是我们的构造函数：

```
public LinkedList () {
    size=0;
    header=new Entry (null, null, null);
    header.prev=header.next=header;
}
```

注意它建立了不变式：它创建了虚拟元素，形成了环，设置了大小。

*Mutator add* 取出了一个元素，并把它加入到了列表的尾端：

```
public void add (Object o) {
    Entry e=new Entry (o, header.prev, header);
    e.prev.next=e;
    e.next.prev=e;
    size++;
}
```

为了检查这个方法，我们假设在程序入口遵守不变式。我们的任务是显示它在出口也同样满足这个不变式。代码的影响是在 *header* 条目之前浏览一个新的条目，也就是说，这个新的条目变成了下一个链的最后条目，因此我们可以看到来自同一个环的约束被保持了。注意在条目上假定不变式的一个序列就是指我们不需要做空引用的检查：例如，我们可以假定 *e.prev* 和 *e.next* 是非空的，因为他们是列表中推出方法的条目，表示不变式告诉我们所有的条目都有非空的 *prev* 和 *next* 属性。

最后，让我们看一下观察者 (*observer*)，*getLast* 的操作返回列表中的最后一个元素，

或者当列表为空时抛出一个异常。

```
public Object getLast () {
    if (size==0) throw new NoSuchElementException ();
    return header.prev.element;
}
```

我们再次假定条目上的不变式。这允许我们废弃 *header.prev*，表示不变式告诉我们它不能为空。在这种情况下，检查不变式被保持是微不足道的。因为没有改变。

## 6.4 解释表示

再次考虑改变者（mutator）的 *add*，它取出一个元素并加入到列表的尾端：

```
public void add (Object o){
    Entry e=new Entry (o, header.prev, header);
    e.prev.next=e;
    e.next.prev=e;
    size++;
}
```

我们检查到这个操作通过正确的把一个新的条目加入到列表里面而保持了表示不变式。不过我们没有检查的是它是否插入到了正确的位置。新的元素插入到了头端或者列表的尾端？看起来似乎插在尾端，但是这就决定了条目的顺序和元素的次序关系。对于一个拥有元素 *o1, o2, o3* 的列表，就有可能这样：

```
p.header.next.element=o3;
p.header.next.next.element=o2;
p.header.next.next.next.element=o1;
```

为了解决这个问题，我们需要知道表示是如何被解释的：也就是说，如何把 `LinkedList` 的实例看成元素的一个抽象序列。这就是抽象函数所提供的。我们实现的抽象函数如下：

```
A(p)=
    if p.size = 0 then
        < > (the empty list)
    else
        <p.header.next.element, p.header.next.next.element, ...>
        (元素的序列索引是 0... size-1, 其中第 i 个元素的索引就是 p.nexti+1)
```

## 6.5 抽象和具体对象

在考虑抽象类型时，把对象想象成两个不同的领域是有帮助的。在具体的领域，我们有真正实现的对象。在抽象的领域，我们有精确的对象与抽象类型的规格方式相对来描述其值。

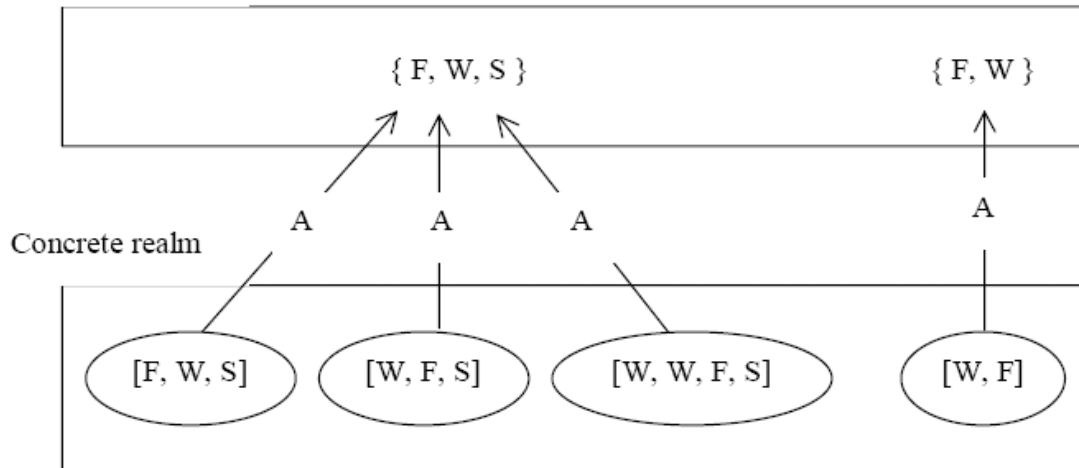
假定我们为处理大学的课程登记而创建了一个程序。对于一个给定的课程，我们需要选择课程属于哪个学期：*秋季、冬季、春季和夏季*。采用 MIT 的风格，我们将把它们记为：*F、W、S、U*。我们所需要的是一个类型 *SeasonSet*，它的值是季度的集合，我们将假设我们已

经有了一个类型 `Season`，这就允许我们以如下方式写代码：

```
if(course.seasons.contains(Season.S)) ...
```

有很多方法来表示我们的类型。我们可以懒惰地使用 `java.util.ArrayList`，这就允许我们以简单包装器的方式来写大多数的方法。抽象和类体领域可能看起来是这样的：

### Abstract realm



下面椭圆形标记为 `[F,W,S]` 代表一个具体对象，它包含一个数组列表，其中第一个元素为 `F`，第二个为 `W`，第三个为 `S`。上面椭圆形标记为 `{F,W,S}` 代表一个抽象集合，它包含三个元素 `F`、`W` 和 `S`。注意相同的抽象集合 `{F,W,S}` 可能有多种表示，如，它可以是 `[W,F,S]`，顺序不重要；或 `[W,W,F,S]`，如果表示不变式允许复制的话；（当然，有很多种抽象集合和具体对象我们没有表示出来；图中仅仅是给出了一个例子。）

两个领域之间的关系就是一个函数，因为每个具体对象至多可被解释成一个抽象值。函数可以是部分的，因为一些具体对象——那些违反了表示不变式的——没有解释。这个函数就是抽象函数，在图中被箭头标记为 `A`。

假定我们的 `SeasonSet` 类有一个属性 `eltlist` 保持了 `ArrayList`。那么我们可以以如下方式写下抽象函数：

$$A(s) = \{s.eltlist.elts[i] \mid 0 \leq i < size(s.eltlist)\}$$

也就是所，集合包含了列表的多有元素。

不同的表示具有不同的抽象函数。另一个展示 `SeasonSet` 类的方法是使用带有 4 个布尔值的数组。以下就是一个抽象函数，

```
[ true, false, true, false ]
```

可以映射为 `{F,S}`，假定数组元素的顺序为 `F`、`W`、`S`、`U`。这个顺序是通过抽象函数转换的信息，可以写成：

$$A(s) = (if\ s.boolarr[0]\ then\ \{F\}\ else\ \{\})\ U \\ (if\ s.boolarr[1]\ then\ \{S\}\ else\ \{\})\ U$$

```
(if s.boolarr[2] then {S} else {}) U
(if s.boolarr[3] then {U} else {})
```

我们同样可以选择一个不同的抽象函数，它以不同的顺序来定位季度：

```
A(s) =
  (if s.boolarr[0] then {S} else {}) U
  (if s.boolarr[1] then {U} else {}) U
  (if s.boolarr[2] then {F} else {}) U
  (if s.boolarr[3] then {W} else {})
```

从这个例子中我们可以得到的一个重要的教训就是选择一个表示比命名属性和选择它们的类型更为重要。抽象函数告诉我们：同样的布尔数组可以解释成不同的方式。同样地，在链式列表的例子中，抽象函数告诉我们条目的次序是如何与元素的次序对应起来的。把抽象函数想象成理所当然的是初学者的一个普遍错误，因为你总可以从代码的声明总猜想到什么。不幸的是，这不总是正确的：例如需要仔细阅读链式列表的代码来发现第一个条目是虚拟条目。

## 6.6 例子：CNF 中的布尔公式

让我们看一个带有抽象函数的简单表示的例子。一个布尔公式是一个创建于命题（符号可以代表的值是 true 和 false）和逻辑操作的精确的公式。例如，公式如下：

```
CourseSix => sixOneSeventy
```

使用两个命题，*CourseSix* 和 *sixOneSeventy*，和逻辑暗示操作。它表示：如果 *courseSix* 为真，那么 *sixOneSeventy* 也为真。如果布尔值的某个分配到使公式正确的命题，那么一个布尔公式就可以满足。这个公式是满足的是因为我们可以设置 *CourseSix* 为 false 或我们可以设置两个命题都是 true。

一个决定公式是否可以满足的算法，称为 *SAT solver*。*SAT solver* 有很多应用，他们的技术在过去十年里提高了很多。他们可应用在设计工具中检测设计的约束，在计划者手上用于发现计划，在测试工具中用于发现哪些暴露特殊类错误的测试，等等。一个 *SAT solver* 也可以用于检测一个证明。假定我们确认以下：

```
CourseSix => sixOneSeventy
```

而且

```
sixOneSeventy => lateNights
```

那么

```
CourseSix => lateNights
```

这是一个的简单推理，当然，让我们看一下如何使用 *SAT solver* 来检查它。我们简单地把前提结合到结论的否定中：

```
(CourseSix => sixOneSeventy)(sixOneSeventy => lateNights)(! CourseSix => lateNights)
```

把这个公式展示给 *solver*。*solver* 将发现它是不可满足的，将证明前提正确的情况结论

不正确：也就是说，证明是有效的。

大多数 *SAT solver* 使用一个布尔公式的表示，称为 *conjunctive normal form*，或者简称为 *CNF*。*CNF* 中的公式是一个子句集合；每个子句是一个字的集合；一个字是一个命题或它的否定。公式被解释成子句的连接，每个子句被解释成字的分离。对于 *CNF*，一个更有用的名字是 *product of sums*，它清楚地使得最远的操作者就是一个产品（例如，连接）。

例如，*CNF* 公式

$$\{\{a\},\{\!b,c\}\}$$

与下面的公式是等价的

$$a \wedge (\neg b \vee c)$$

以上的公式可以以 *CNF* 表示为：

$$\{\{\neg \text{courseSix}, \text{sixONEseventy}\}, \{\neg \text{sixOneSeventy}, \text{lateNights}\}, \{\text{courseSix}\}, \{\neg \text{lateNights}\}\}$$

现在让我们考虑我们真样可以建立一个保持 *CNF* 公式的抽象数据类型。假定我们已经有了一个代表字的类 *Literal*。下面是一个使用 Java 库 *ArrayList* 类的推理表示。

```
public class Formula{
    private ArrayList clauses;
    ...
}
```

子句的属性就是一个 *ArrayList*，它的元素是字 *ArrayList* 本身。

我们的表示不变式可以是：

$$R(f) = f.\text{clauses} \neq \text{null} \ \&\& \\ \text{all } c: f.\text{clauses}.\text{elts} \mid \\ c \text{ instanceof } \text{ArrayList} \ \&\& \ c \neq \text{null} \ \&\& \\ \text{all } l: c.\text{elts} \mid c \text{ instanceof } \text{Literal} \ \&\& \ c \neq \text{null}$$

我们已经使用了规格说明属性 *elts* 来代表 *ArrayList* 的元素。表示不变式表明 *ArrayList* 子句的元素是非空的 *ArrayList*，每个包含非空字的元素。

以下就是抽象函数：

$$A(f) = \text{true} \wedge C(f.\text{clauses}.\text{elts}[0]) \wedge \dots \wedge C(f.\text{clauses}.\text{elts}[(\text{size}(f.\text{clauses}) - 1)])$$

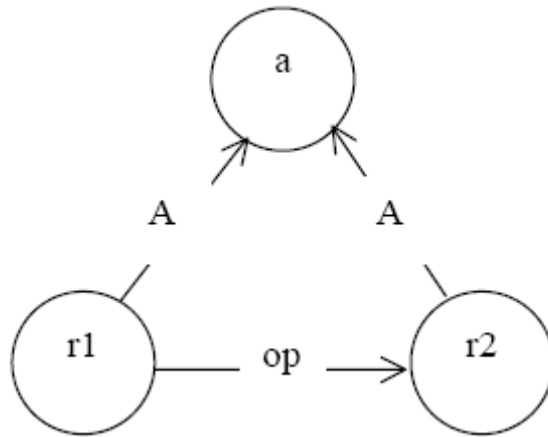
其中， $C(c) = \text{false} \vee c.\text{elts}[0] \vee \dots \vee c.\text{elts}[0]$

注意我们已经介绍了一个辅助函数 *C*，它把子句抽象成了公式。看看这个定义，我们可以解决边界情况的含义。假定 *f.clauses* 是一个空的 *ArrayList*，那么因为第一行右边的结合，所以 *A(f)* 将是正确的。假定 *f.clauses* 包含一个单字句 *c*，*c* 是一个空的 *ArrayList*，那么 *C(c)* 将是错误的，*A(f)* 也将是错误的。有两个基本的布尔值：*true* 由空子句集表示，*false* 又包含空子句的集合表示。

## 6.7 善意的影响

什么是一个观察者（observer）操作？在我们的关于表示独立性和数据抽象的介绍中，我们把它定义为一个不能改变对象的操作。下面我们给出一个更大的定义。

一个操作可以改变一个类型的对象，因此表示的属性也会发生变化。这将维护它所定义的抽象数值。我们可以以下图来描述这种现象：



操作 *op* 的执行改变了 *r1* 到 *r2* 对象的表示，但是 *r1* 和 *r2* 是通过抽象函数 *A* 到抽象数值 *a* 映射的，因此数据类型的客户不能观察到所出现的任何改变。

例如，*LinkedList* 的 *get* 方法可以缓存获取的最后一个元素，因此对 *get* 的相同索引的重复调用将会更快。这个写到缓存的内容（这种情况下仅仅是两个属性）改变了表示，但是它对于对象的数值不会产生任何影响，因为它可以通过类型的调用操作来观察。客户不知道是否有一个检索被缓存（除了注意到性能的提高）。

一般的，我们可以允许操作者来改变表示，只要抽象数值被保存了。我们将必须确保表示不变式不被破坏。如果我们作为 *checkRep* 方法来编写不变式，我们应该在观察者的开始和结尾处插入。

## 6.8 总结

为什么我们要使用表示不变式？因为不变式可以真正地减少工作量：

- 它使得模块推理变为可能。没有表示不变式的文档记录，你在添加新的方法时，可能不得不阅读所有的方法来理解如何进行。
- 它可以帮助捕捉错误。通过把不变式作为一个运行确认来实现，你可以发现一些采用其它方法很难发现的错误。

抽象函数指定了一定抽象数据类型如何作为抽象数值被表示。使用表示不变式，我们能够以模块的风格推导一个类型操作的正确性。

事实上，抽象函数写起来比表示不变式要困难。书写表示不变式总是值得的，你应该坚持不懈的做这件事情。书写一个抽象函数也总是很有用的，即使写的不规范。但是有时抽象

域很难界定，而且书写一个详细的抽象函数所花的额外工作也不值得。这需要你去判断。