不确定性算法在C++中的表达：一个简化模型

Wang Tianxing
<wangtx@novel-tongfang.com>

2004年5月17日

摘要

本文建立了一个简单化的不确定性计算模型，并用 C++ 语言实现了一个支持以这种模型 进行计算的库，并以计算 24 的程序展示了这个模型强大而且优雅的表达能力。

关键词：C++, 不确定, 算法, nondeterministic, algorithm。

目录

1 警告!

本文很长，没几个人有耐心读完！

2 简单地介绍一下不确定性计算

“不确定”这个词，在计算机科学中有严格的定义[turing]。大概的意思是指，在 计算的任意阶段，其后续计算可以是多种可能同时进行，只要有一种可能得到肯定的结果， 那么整个计算就得到肯定的结果。

我们实际使用的计算机当然是确定的，一般程序里用的算法也是确定的。但是，有很多问 题，如果允许用不确定的方式来描述解决的算法，也就是允许用“对每一种情况如何如何” 这种方式写算法，而不是总要写出详细的循环或递归，那么算法的描述将会简化很多。

几乎所有的算法都含有不确定性成分。我们这里要描述的方法适用于不确定性成分比较多 的场合，例如输入有不确定性，计算的规则里有多种可能性，或者计算的结果可能有多个 等等。

例如各种需要用穷举进行搜索的算法，其中的穷举和搜索过程就是典型的不确定计算模式。 又例如语法分析程序，其中有很多匹配规则有多种可能性，如果允许歧义的话结果也可能 是多个。

允许算法里采用不确定性描述并不能解决原来解决不了的问题。我们的目的只是更简单地 解决能解决的问题。

3 一个看起来没有一点用的模型

一个“计算”可以抽象为：

一个“规则”作用于一个“输入数据”上，得到一个“输出数据集合”。

上面就是我们的不确定性计算模型，够简单吧！请注意，输出的是一个“集合”，因此可 以表达不确定性计算，即可能没有结果，可能一个结果，也可能得到多个结果。

我们必须把上面那一句话写成 C++ 语言的程序，否则就是在说空话，浪费大家的时间了。 在上面的话里有三个重要概念，即“规则”、“输入数据”和“输出数据集合”，我们为 它们在 C++ 里建立三个“概念”[concept]，分别叫做 Rule, Universe, 和 Multiverse[multiverse]：

Universe ：一个普通的“值”类型，即 CopyConstructible & Assignable，
也就是可以放到标准容器中的类型。

Multiverse ：用于接受多个 Universe 对象，继承自 Multiverse 类：

```
struct Multiverse { virtual void result(Universe& u) = 0; };
```

在具体的 Multiverse 实现的 result() 成员函数中，可以修改传给它的 Universe 参数。

Rule ：一个类类型，其中定义了一个静态成员函数 apply()：

```
struct Rule { static void apply(Universe& u, Multiverse& m); };
```

这里，u 是规则的输入，m 用于接受规则的输出。规则每得到一个结果，就用这个结 果做参数调用一次 m.result()。请注意，apply() 可以改变它的 u 参数，同样 m 的 result() 函数也能改变它的参数，因此有多个结果时要注意恰当地使用拷贝。

Universe, Multiverse 和 Rule 三个概念就描述了不确定性计算在 C++ 里的一个模型。

我们说这个模型是简化了的，是因为一个 Rule 作用于一个 Universe 得到的 Multiverse 是 Universe 的集合，而不允许是其他东西的集合：

$$Rule :: Universe \longrightarrow \{Universe\}$$

这个简化是很大的。更加通用的模型应该是：

$$Rule :: Universe \longrightarrow \{Universe'\}$$

但是，因为这个简化后的模型也很有用，也反映了很多重大问题的解决方法，因此我们用 它作为不确定性计算的入手点在本文中加以介绍。

4 这个模型真的有用吗

只凭这三个概念，确实不能给我们的编程带来多少帮助。

一般的形而上程序设计(metaprogramming, [meta])都是这样进行的：建立几个抽 象概念，然后想当然地在这些概念上写一大堆没准儿有用的代码，然后在真正的解决实际 问题的过程中再来看这些东西有没有用，有用就用，没用就写新的，顺便扩充一下原来的 库。

我们这里也不例外。我们建立了上面的简单模型后，就要对着这个模型先编出一堆很可能 有用的程序，具体点说就是一些通用的规则和规则的组合程序。

我们使用这个模型以及相应的支持库写出来的程序的一般结构是：

- 定义一个能够表达要解决的问题的 Universe 类型。
- 定义若干基本的规则。
- 使用规则组合方法对基本规则进行组合，最后得到一条解决问题的终极规则。
- 定义一个 Multiverse 的派生类型，在它的 result() 函数里对结果进行 最后的审判，决定一个结果是该上天堂还是该下地狱。
- 把输入数据做成一个 Universe，声明一个最终审判用的对象(上面定义的 Multiverse的派生类型)，对这个 Universe 和 Multiverse 调用那条终极规则，系统 将尝试所有可能的情况，对所有可能的结果进行最终审判，得到最终结果。

因此，我们现在最重要的工作就是写一堆看起来可能很有用的规则组合方法。在写这些代 码时，上面的三个概念将是核心，甚至是我们仅有的依据，因此要充分认识“概念”的重 要性。

5 在解决具体问题前先写一些通用的程序

现在我们打算对着那三个没多少内容的概念绞尽脑汁写出一些希望很有用的程序。

从哪里下手呢？我们看到，Universe 贯穿在整个计算过程中，变不出任何花样，而 Multiverse 作为 Universe 的集合也就没有多少花样了，因此重点是在规则上打主意。 规则上看起来也没多少花样，不过有一些，那就是规则的组合。最基本的组合方式有两种， 即顺序式和选择式。

在搞规则组合前，我们先给出两个显得很无聊，但是也的确非常通用的规则：

zero ：对任意输入，都不输出任何结果：

$$zero(u) = \{\}$$

one ：对任意输入，都把输入直接当作结果输出：

$$one(u) = \{u\}$$

它们的实现很简单：

```
struct zero {
    template<typename Universe>
    static void apply(Universe&, Multiverse<Universe>&) {}
};

struct one {
    template<typename Universe>
    static void apply(Universe& u, Multiverse<Universe>& m) { m.result(u); }
};
```

请注意，为了这些规则可以运用于任何应用，我们把 Universe 变为模板参数了，对应的， Multiverse 的概念中也应该改为需要继承 Multiverse<Universe>：

```
template<typename Universe>
struct Multiverse { virtual void result(Universe& u) = 0; };
```

现在我们想规则的组合：顺序式和选择式。

所谓顺序式，就是顺序运用多条规则。比如两条规则 P 和 Q 的顺序式组合，就是对对一 个 Universe 运用 P 后的结果集中的每一个结果，再运用 Q，把所有这些结果并到一起， 作为 P 和 Q 的组合的结果。我们把顺序式合成叫做乘法，相应的运算结果叫做 product<P,Q>。请注意，这个乘法一般是不可交换的。可以看出， one 与任何规则 相乘得到的还是那条规则，zero 与任何规则相乘都得到 zero，这也是它们的名称的由来。

我们用 P Q 表示规则 P 和规则 Q 的乘积，那么：

$$(P\ Q)(u) = \backslash union\ \{x\ \backslash in\ P(u)\}\ Q(x)$$

即 P 与 Q 的乘积作用于 u 的效果，相当于对 P 作用于 u 的结果集中的每个 x 计算 Q(x)，然后把它们并起来。显然规则间的乘法运算适合结合律，因此在多个规则相乘时我 们可以省去括号，例如 P Q R S 表示 P, Q, R, 和 S 的乘积。

我们通过引入一个辅助的 Multiverse 来解决这个问题。这个 applying_multiverse<> 在构造时记住另一个 Multiverse，当来了一个 Universe 时，运用它的 Rule 再次对这 个数据进行加工，结果放到它记着的那个 Multiverse 中：

```
template<typename Universe, typename Rule>
struct applying_multiverse: Multiverse<Universe> {
    applying_multiverse(Multiverse<Universe>& m): m_(m) {}
    void result(Universe& u) { Rule::apply(u, m_); }
private:
    Multiverse<Universe>& m_;
};

template<typename P, typename Q>
struct product {
    template<typename Universe>
    static void apply(Universe& u, Multiverse<Universe>& m)
    {
        applying_multiverse<Universe, Q> tm(m);
        P::apply(u, tm);
    }
};
```

所谓选择式，就是多条规则，每条都可以用一用，把所有得到的结果放在一起，作为选择 式合成规则的结果。我们把这个合成方式叫做加法，并用 sum<P,Q> 表示 P 与 Q 的 选择式合成。显然 zero 加任何规则都得到那条规则。

我们用 $P \mid Q$ 表示规则 P 和规则 Q 的选择式合成，那么：

$$(P \mid Q)(u) = P(u)\ \backslash union\ Q(u)$$

即 P 与 Q 的和作用于 u 的效果，相当于 P 与 Q 单独作用于 u 的结果的并集。显然规 则的加运算适合结合律和交换律，因此多个规则相加时可以省去括号，例如用 $P \mid Q \mid R$ 表示 P, Q 和 R 的和。另外，乘法对加法有分配律，即：

$$(P \mid Q)\ R = P\ R \mid Q\ R$$

$$R\ (P \mid Q) = R\ P \mid R\ Q$$

这里我们也约定了乘法的优先级高于加法。

sum<> 的实现很简单，只要把 P 和 Q 的结果放到一起就行了：

```
template<typename P, typename Q>
struct sum {
    template<typename Universe>
    static void apply(Universe& u, Multiverse<Universe>& m)
    {
        Universe v(u);
        P::apply(u, m);
        Q::apply(v, m);
    }
};
```

请注意，为了让 P 和 Q 都作用于同一个输入上，我们对输入作了一个拷贝，因为我们约 定了规则是可以修改它的 Universe 参数的，如果我们不做拷贝，传给 Q 的就可能是被 P 改过的东西，不正确。

在实际应用中，经常有不止两个规则顺序或选择式合成的情况。多个规则的合成显然能够 用上面给出的两个规则的合成来实现，例如：

```
template<typename P, typename Q, typenme R>
struct sum3 : sum<P, sum<Q, R> > {};

template<typename P, typename Q, typename R, typename S>
struct sum4 : sum<P, sum<Q, sum<R, S> > > {};

// some rules:
struct r1 {...};        struct r2 {...};
struct r3 {...};        struct r4 {...};

// sum them up:
struct four_rules: sum4<r1, r2, r3, r4> {};

// or(worse):
typedef sum4<r1, r2, r3, r4> four_rules;
```

请仔细注意上面的规则合成方式。这些规则的合成有个特点，就是只是类型的组合，没有 一行是可执行的指令式程序。虽然基本的规则少不了指令式程序，就是 if、for、赋值语 句之类的东西，但是在高级的规则合成层次上，将看不到用于执行的语句，而只有类型的 构造，就象上面的 sum3<>, sum4<>, four_rules 等一样。

注意到只有类型模板实例化和继承，而且被定义类型在功能上等价于它继承的类型，我们 用等式来表示上面的类型间的关系有时比用 C++ 要简便很多：

$$\begin{aligned} sum3<P,Q,R> &= P \mid Q \mid R \\ sum4<P,Q,R,S> &= P \mid Q \mid R \mid S \\ four\_rules &= r1 \mid r2 \mid r3 \mid r4 \end{aligned}$$

类似地，我们同样也可以定义 product3<>, product4<>, 等等。我们在最后给出的库 代码里用了一个稍微不同的方法，直接实现了支持最多到 26 个参数的 sum<> 和 product<>：

```
template<typename A=one, typename B=one, typename C=one, typename D=one,
         typename E=one, typename F=one, typename G=one, typename H=one,
         typename I=one, typename J=one, typename K=one, typename L=one,
         typename M=one, typename N=one, typename O=one, typename P=one,
         typename Q=one, typename R=one, typename S=one, typename T=one,
         typename U=one, typename V=one, typename W=one, typename X=one,
         typename Y=one, typename Z=one>
struct product {
    template<typename Universe>
    static void apply(Universe& u, Multiverse<Universe>& m)
    {
        applying_multiverse<Universe,
                            product<B, C, D, E, F, G, H, I, J, K, L, M, N, O,
                                    P, Q, R, S, T, U, V, W, X, Y, Z> > tm(m);
        A::apply(u, tm);
    }
};


template<>
struct product<one, one, one, one, one, one, one, one, one, one, one, one, one,
               one, one, one, one, one, one, one, one, one, one, one, one, one
>: one {};
```

这里，虽然参数个数是有限的，但是太多了，所以我们用了类型上的递归，并用一个完全特例化来结束递归。

sum<> 的实现类似，具体的实现就请你自己看代码了。

我们在这一节里写的两条规则和两种规则的组合方法没有涉及到任何具体应用，因此它们必 然是非常通用的。

6 我们写的通用规则和组合方法已经很强大了

现在我们要说的是，最初看似简单而且抽象的一句话，现在已经发展到相当不简单的程度 了。如果把针对要解决的问题定义的规则加上 zero 和 one 称为规则集，那么：

用 sum<> 和 product<> 进行组合，可以构造出规则集上的上下文无关语言。

上下文无关语言，也就是 Chomsky type-2 语言[chomsky]，是一个相当丰富的 结构，例如常用的程序设计语言的语法一般都尽量用上下文无关语法来描述。

sum<> 和 product<> 之所以有这样的能力，是因为一个前面我们还没有使用过的 能力，就是定义规则时我们可以用递归：

```
template<typename R>
struct star: sum<one, product<R, star> > {};
```

或者用简化写法：

$$star\texttt{<R>} \;=\; one \mid R\ star\texttt{<R>}$$

可以用递归，是因为 C++ 里允许在定义一个类时，在其基类中使用这个类本身。

上面通过递归用 sum<> 和 product<> 定义的 star<>，就是正则表达式或 BNF 中 常用的星号运算，也就是把一个规则运用 0 次，1 次，或任意多次。这样，我们的系统就 是强于正则语法(Chomsky type-3 语言)的，因为我们可以自己实现 star<>，而在正则 语法中 star<> 是一个预先给定的操作，无法用其他操作定义。

现在，zero, one, sum<>, product<> 和 star<> 构成了规则上的一个克林代数 ([kleene]Kleene Algebra，也就是描述正则语言的系统。Kleene 的正确的音译 应该是克雷尼，但是数学书上都译成克林了)，而且整个系统强于克林代数。

需要注意的是，在规则组合时不能使用左递归，否则程序运行的时候就要无限递归了。而 我们大家都知道，不用左递归并不会限制可表达的语言结构，因为左递归都可以用固定的 方法消除掉。例如，任何直接左递归都可以化为：

$$X \;=\; u \mid X\ v$$

的形式，其中 u 和 v 左端没有 X。而这个式子可以用等价的没有左递归的：

$$X \;=\; u\ \texttt{v*}$$

来表示。这里，我们用了 v* 作为 star<v> 的简写。

7 也许真的很强大了，不过到底有什么用啊

前面我们没有考虑具体应用而写了一堆号称能表达很复杂的结构的东西。不是想卖关子， 实在是因为我觉得不把这个模型发展到一定程度就不能用。当然，发展到一定程度之后如 果还不去具体应用，那也真是浪费大家的时间了。

现在我们打算用前面的一堆东西去解决一个具体问题：算 24。新闻组里老是有人问算 24 的程序怎么写，很多人也玩过。所谓算 24，就是给定几个整数(标准的是 4 个)，然后只 允许用四则运算，算出 24 来。例如，给定 3，3，8，8 四个数，那么 8/(3-8/3) 就是 一个答案。

我们选择个问题来解决，当然是因为这个问题适合于用我们的工具解决(哪个作者不是挑 拿手的问题去解决)。解决这个问题时，需要构造有一定结构的东西，就是四则运算表达 式，然后判断它是否满足要求。这个构造过程就具有不确定的特征，因为我们只能尝试各 种可能(某些天才也许能运用一些数论避免盲目构造，不过用计算机的人都是懒得当天才)， 也许找不到答案，也许能找到多个答案。而且，要构造的表达式有一定结构上的复杂性。 因此，解决这个问题多少能显示一点我们的不确定计算模型的力量。

前面我们已经说过用我们的库解决问题的步骤，这里再重复一下：我们需要定义一个表示 我们处理的对象的 Universe 类型，实现一组对 Universe 进行基本操作的原始规则，用 前面开发的规则组合方法把原始规则组合成解决问题的终极规则，还要写一个最后判定结 果的 Multiverse，再把这条终极规则作用到输入数据上，在最后审判的地方得到我们想 要的结果。

在开始我们的不确定计算之前，我们先写一个很传统的类：Value。虽然算 24 一般输入 和结果(24)都是整数，但是中间结果可能是分数，因此我们需要一个分数计算的类。最后 输出结果时，我们要的是计算的式

子而不是结果(因为结果早就知道了！)，所以我们还需 要把算式本身记录下来。我们把分数运算和记录算式的功能都放到 Value 类中了，我知 道这样不好，不过因为这不是我们的重点，我们只需要它能用就行了：

```
struct Value {
    explicit Value(int a): x(a), y(1)
    {
        char buf[64];
        sprintf(buf, "%d", a);
        s = buf;
    }

    Value(int a, int b, std::string const& c): x(a), y(b), s(c) {}

    Value operator+(Value const& r) const
    { return Value(x*r.y+y*r.x, y*r.y, op("+", r)); }

    Value operator-(Value const& r) const
    { return Value(x*r.y-y*r.x, y*r.y, op("-",r)); }


    Value operator*(Value const& r) const
    { return Value(x*r.x, y*r.y, op("*",r)); }

    Value operator/(Value const& r) const
    { return Value(x*r.y, r.y ? y*r.x : 0, op("/", r)); }

    bool valid() const { return y != 0; }

    bool operator==(Value const& r) const
    { return valid() && r.valid() && x * r.y == r.x * y; }

    int x, y;
    std::string s;

private:
    std::string op(char const* sop, Value const& r) const
    { return "(" + s + sop + r.s + ")"; }
};

typedef Value(Value::*Operator)(Value const&) const;
```

一个 Value 类的对象表示的值是 x/y，得到这个值的表达式是 s。需要留意一下， Value 中对 y==0 的情况有一些特殊的处理。

另外我们定义了一个 Value 的成员函数指针类型 Operator，以后将用它引用 Value 提 供的四则运算。

下面我们开始写和我们的不确定计算模型有关的代码。

8 先来设计计算 24 用的 Universe

之所以把表达问题数据的结构称为宇宙(Universe)，是因为所有的状态信息都必须存在这 个结构中。当然也有一些和量子论里的多世界解释的类比因素。

一个 Universe 必须能够表示计算开始，中间过程，以及最终结果的每一个状态。因此需 要一个存储给定的整数的成员，我们用一个整数的向量来表示：

```
std::vector<int> nums;
```

在计算过程中，我们需要保存一个表达式的子表达式的值以及运算符本身，而且子表达式 可能嵌套，因此我们用一个 Value 的栈和一个 Operator 的栈来表示：

```
std::stack<Value> vals;
std::stack<Operator> ops;
```

当然，上面这个需要在一开始也许并不明显，不过开始写规则后就会明显了，我们这里为 了叙述上的方便把它提前引进了。这样我们得到如下的 Universe 结构：

```
struct Universe {
        std::vector<int> nums;
        std::stack<Value> vals;
        std::stack<Operator> ops;
};
```

需要再次说明的是，Universe 里到底需要什么东西，除了初始输入数据，剩下的实在是 要在写规则时才清楚。也许真该晚一点再说 Universe 的内容。

9 如何对所有的结果进行最后的审判

在写计算规则之前，让我们先把最后的目标确定下来：我们现在写最后对结果进行判定的类。

我们把进行最后审判的场所伪装成一个普普通通的 Multiverse，这样各个计算的结果就 可以不知不觉地被引进去，在那里接受审判。我们把这个 Multiverse 叫做 Destiny：

```
typedef ::smonde::Multiverse<Universe> Multiverse;

struct Destiny: Multiverse {
    void result(Universe& u)
    {
        if (u.nums.size() != 0) return;
        Value const& v = u.vals.top();
        if (v == Value(24)) std::cout << v.s << "\n";
    }
};
```

前面一直没说，我们前面写的 zero, one, sum<>, product<>, Multiverse<> 等都是放在一个叫 smonde(a simplified model of nondeterministic evaluations) 里的，因此我们这里用一个 typedef 让 Multiverse 更便于使用，以后写规则时还要反复 用它。

上面实现的 result() 里对接受审判的 Universe 进行了两个判断。第一个是看给定的数 字是否用完了，因为不把给的数字全用上是不允许的。第二个判断是检查 vals 栈顶的值 是不是我们想要的 24。这里，我们有一个约定，就是每构造出一个表达式，就把它的值 (也包括表达式形式)放在 vals 栈顶。另外，我们的规则保证，全部输入数据用完后，一 定是在 vals 中得到唯一的一个表达式的值。因此 result() 做的事就是，如果一个表达 式用完了所有输入数字，而且结果是 24，就把这个表达式显示到 cout。

在我们这个算法里，24 这个数字显然没有什么特殊的，完全可以是任何整数(或有理数)， 因此我们可以把 24 改成 Destiny 的构造参数，让我们的程序能很容易地算任何整数结 果：

```
struct Destiny : Multiverse {
    Destiny(int g): goal_(g) {}

    void result(Universe& u)
    {
        if (u.nums.size() != 0) return;
        Value const& v = u.vals.top();
        if (v == Value(goal_)) std::cout << v.s << "\n";
    }
private:
        int goal_;
};
```

虽然已经不限于计算 24，但是我们还将继续把这个程序称为计算 24 的程序。

10 先写主程序，以便对程序的总体结构有个了解

这里我们只给出一个具体的用 3, 3, 8, 8 算 24 的程序片断：

```
Universe u;

u.nums.push_back(3);            u.nums.push_back(3);
u.nums.push_back(8);            u.nums.push_back(8);

Destiny d(24);

expr::apply(u, d);
```

Universe 和 Destiny 在前面已经定义过了，expr 就是我们将要构造的搜索所有表达式 的终极规则。从输入数据构造的 Universe u 开始，在终极规则 expr 的作用下，所有用 那些输入数据的四则运算构成的表达式都将进入 Destiny d，在那里那些满足我们的要求 的表达式将被挑出来进行显示。

实际的程序中输入数据是从 std::cin 读入的，并且没有算 24 和 4 个整数的限制， 这里就省略了。

11 现在写和计算 24 有关的基本规则

这里要写的基本规则都是构造终极的 expr 要用到的，和我们的具体的目的以及 Universe 的结构相关的。所谓“基本”在这里指的是必须手工写 apply() 函数的那些规 则。我们把那些通过使用 sum<>, product<> 等构造工具从其他规则构造出来的规则 称为“构造性规则”，将留到下一节再写出来。

规则 NUM ：从输入数据中取一个数放到 vals 栈顶。

这条规则就具有所谓的不确定性，因为如果有多个输入数据，“取一个数”这个操作就有 多种做法。因此我们在实现 NUM 时就要穷尽所有的可能：

```
struct NUM {
    static void apply(Universe& u, Multiverse& m)
    {
        for (size_t i=0; i < u.nums.size(); \texttt{++}i) {
            Universe v(u);
            v.vals.push(Value(v.nums[i]));
            v.nums[i] = v.nums.back();
            v.nums.pop_back();
            m.result(v);
        }
    }
};
```

在上面的实现中，我们是对 nums 中的每个数，把它移到 vals 栈顶来完成“取一个数” 操作的。被我们取过数后的 nums 里的数的顺序有点乱，不过因为不违反我们的目的，打 乱输入数据的顺序是被允许的。

这就是我们写的第一条应用规则。请仔细体会一下我们前面一直在说的不确定性， Universe 的可拷贝性，以及 Multiverse 作为 Universe 的集合等概念在其中所起的作 用。

规则模板 OP<op> ：如果还可以再用一个二元运算符去构造二元运算表达式，
那么就把 op 放到 ops 栈顶。

这是一条规则模板，有一个 Operator op 做参数。OP<op> 也有一点不确定性，就是能 继续构造表达式时，放一个 op 到 ops 上作为结果，不能构造时将得不到任何结果。

这条规则模板有点费解之处，就是“还可以用一个运算符构造表达式时”这句话。这个问 题也许要等到看到使用这条规则的其他规则才能清楚，不过这里先勉强解释一下：给定若 干个输入数据后，在构造表达式的过程中并不是可以无限制地使用二元运算符的。例如， 如果给了你四个数，你最多能用 3 个二元运算符，多了你也用不上。我们要做是否还能 用二元运算符的判断，实际上是为了后面避免左递归，因为一条规则如果不从输入中取走 一点东西的话，它就相当于空规则(讲语法的书里叫 epsilon，我们这里叫 one) 。空规 则开头，后面如果紧跟着进行递归的规则的话，就出现了左递归。我们在 OP<op> 中对 剩下的输入的个数、已经生成的子表达式的个数、已经用了的二元运算符的个数进行判断， 实际上相当于实实在在地从输入中“拿走”了一个运算符，因为把 op 放到 ops 栈顶是 会影响以后的判断的。下面是 OP<> 的实现：

```
template<Operator Op>
struct OP {
    static void apply(Universe& u, Multiverse& m)
```

```
        {
            if(u.nums.size()+u.vals.size() > u.ops.size()+1) {
                u.ops.push(Op);
                m.result(u);
            }
        }
    };
```

上面的判断说的是：如果剩下的输入数据的个数加构造出来的子表达式的个数大于正在处 理的运算符的个数加一的话，就还可以再用一个运算符。我实在希望这件事情是很明显的， 可是我们解释这个事情花的篇幅已经比程序本身长很多了！

规则 REDUCE ：从 vals 栈顶取下两个操作数，从 ops 栈顶取下一个运算符， 用这个运算符对那两个操作数进行运算，把结果放到 vals 栈顶。

使用这条规则的环境必须保证确实有两个操作数以及一个运算符在那里。这一条规则可以 说是完全确定的，不存在任何不确定因素。我们的不确定计算模型当然已经包含了确定的 计算。REDUCE 可以如下实现：

```
struct REDUCE {
    static void apply(Universe& u, Multiverse& m)
    {
        Value v2 = u.vals.top();    u.vals.pop();
        Value v1 = u.vals.top();    u.vals.pop();
        Operator op = u.ops.top();  u.ops.pop();
        Value v = (v1.*op)(v2);
        u.vals.push(v);
        m.result(u);
    }
};
```

NUM, OP<>, 和 REDUCE 就是我们要手写的全部基本规则和规则模板，剩下的规则我们将 用构造的方法做出。

12 从基本规则开始构造出终极规则

规则 any_op ：如果还可以的话，把一个运算符放到 ops 栈顶。

这里所谓的“如果还可以的话”就是指上面解释 OP<> 时说的还可以用二元运算符的那个 条件。这里的“把一个运算符放到栈顶”是一个不确定的操作，即可以放四则运算的任一 个。使用 OP<> 规则模板以及通用的规则组合运算，可以这样定义 any_op：

```
struct any_op : sum<
    OP<&Value::operator +>,     OP<&Value::operator ->,
    OP<&Value::operator *>,     OP<&Value::operator />
> {};
```

希望大家已经习惯于读这样的程序了！用简化表示可以写为：

$$any\_op \;=\; OP\langle\text{'+'}\rangle \mid OP\langle\text{'-'}\rangle \mid OP\langle\text{'*'}\rangle \mid OP\langle\text{'/'}\rangle$$

现在我们来定义终极规则 expr：

规则 expr ：取一些输入数据和运算符，构造成一个表达式，把结果放到 vals 栈顶。

有了随便取一个数的 NUM，随便取一个运算符的 any_op，还有把取来的数和运算符进行 运算的 REDUCE，我们就可以构造出我们一直期待的终极规则 expr：

```
struct expr: sum<NUM, product<any_op, expr, expr, REDUCE> > {};
```

是的，你没看错，就是这么简单！用我们前面一直提到的简化符号可以写成：

$$expr \ = \ NUM \mid any\_op \ expr \ expr \ REDUCE$$

熟悉编译原理的朋友一眼就能看出，上面就是很普通的前缀形式的表达式的产生式， REDUCE 没有语法效果，只是执行一个语义动作。也请注意，这条规则的强大来源于递归 的使用。

对这条规则，有两点需要说明。

首先是 expr 的语义。expr 有两个分支，这两个分支都应该满足我们对 expr 这个规则 的语义要求。第一个分支 NUM，是从输入中取一个数放到 vals 栈顶，因此满足 expr 的 语义。第二个分支 any_op expr expr REDUCE，是从输入任取一个运算符，再取两个表达 式，再运行 REDUCE。我们知道 expr 是把一个表达式的值放到 vals 栈顶，因此运行 REDUCE 时，我们已经取到了一个运算符和两个子表达式的值，而 REDUCE 把它们进行运 算，把结果放到 vals 栈上，因此也满足我们对 expr 的要求，即消耗一些输入，把一个 表达式的值放到 vals 栈顶。

另一点需要说明的就是我们前面说过的 any_op/OP<> 中对“是否还能再取一个操作符” 的判断。现在我们可以看清楚，如果 any_op 不对输入造成任何影响的话，接下来的 expr 规则就可能重新走入这个分支，但是却没有任何实际的进展，因此会造成无限递归。 这也是我们用前缀形式而不用中缀形式的原因，因为中缀形式本来就是我们的系统不支持 的直接左递归。

OP<> 不直接对 nums 造成影响，但是它对 ops 造成影响，而在判断时，OP<> 是使用 了 ops 等的状态的，因此我们可以把 nums，vals，和 ops 合在一起理解为广义的 “输入”， OP<> 是对这个广义的输入造成影响的，而且是消耗式的，这保证了我们的 程序的进展。

13 看看我们费了很大的劲写出来的程序

完了，写完了，费了很大的劲，总算把这个程序写完了。我们写出来的程序是这样的：

smonde.h：通用的不确定计算支持库
24-s1.cpp：计算 24 的程序

在这个程序中，我们对前面给出的 Destiny 稍微作了一些修改，增加了一个 seen_ 成员 变量，用于记录已经输出过的答案，来避免显示看起来完全相同的答案。这些看起来完全 相同的答案只有当输入数据中有相同的数时才会出现，因此通过修改 NUM 规则，避免重 复枚举相同的数值，是可以避免这种完全相同的重复的。我们实际上用了一个懒人的解决 办法，修改 NUM 规则的解决办法就留作练习了。(事实上，这个懒人的做法很可能效率更 高！)

现在，让我们把程序编译一下并运行。我们输入：

```
24 3 3 8 8 e
```

程序输出：

```
(8/(3-(8/3)))
```

很不错！得到的答案完全正确！再运行一下，这次输入：

```
24 1 5 5 5e
```

程序输出：

```
(5*(5-(1/5)))
((5-(1/5))*5)
```

输出的答案仍然是正确的，不过这两个答案看起来没有实质区别。

现在我们稍微回顾一下。我们费了很大的劲，解决了一个小问题。在解决这个小问题时我 们用了一种以前没有用过的程序设计模式。费这么大劲，值得吗？在本文的剩下的篇幅里， 我们将尽力去说明，这么做是值得的。

在开始新的旅程前，让我们先看一看我们前面走过的路是多么的短吧！这里是一个几乎和 smonde.h + 24-s1.cpp 等价的算 24 的程序，它也可以接受任意个输入，也可以算出任 意的整数(只要你稍微改一下)，甚至于要容易理解得多： 24-s0.cpp

这么看来，我们是用了一万四千五百字，285 行程序，解决了一个用 60 来行程序就能解 决的问题。这多少让人有点沮丧。

不过，也不必太沮丧，因为这毕竟只是一个小程序，这么小的程序基本上是什么也证明不 了的，反正我们总可以这么说。

下面，我们打算解决一个复杂一点的问题，而且试图借此表明，用我们的系统来解决复杂 问题，确实比用一些专门的算法有更大的可扩展性，而且得到的算法具有更好的可读性。

我们要解决的新问题是：给定几个整数，找到用这几个数的四则运算得到 24 的方法。这 就是我们前面已经解决的问题！不过，我们现在再加上一个限制，就是输出的答案中不能 包含本质上相同的答案。

下面我们就修改 24-s1.cpp，让它只输出“本质上不同”的答案，改好之后，我们就对修 改的容易等大加夸赞，然后声称在 24-s0.cpp 的基础上，类似的修改几乎是不可能的， 以至于我们根本就不屑于一试，这样我们就可以“证明”，用 smonde 确实是很好的！

14 什么是本质上不同的答案

首先我们必须精确定义“本质上不同”。这事情是见仁见智的，我们这里只打算给一个能 够排除大多数人都承认的“本质上相同”的答案的定义。

让我们先来看几个具体例子，以便从中发现“本质上相同”的特征。先看我们的程序输入 24 1 5 5 5 时的输出：

```
(5*(5-(1/5)))
((5-(1/5))*5)
```

大家一般都认为这两个答案是本质上相同的，因为只要运用一下乘法交换律就可以将一个 答案变换为另一个。就是说，我们都同意，使用交换律可以互相变换的答案是“本质上相 同的”。同样地，大家可能也都同意，运用加法或乘法的结合律可以互相变换的答案也是 “本质上相同的”。

再来看这几个式子：

```
1-2+3-4
1+3-2-4
3+1-4-2
```

我不知道这叫什么律了，反正把加减法(或乘除法)组成的式子象上面那样重新排列组合， 看起来是一件很简单的事情，因此我们也觉得它们是本质上相同的。

现在我们就把经过象上面那样的重新排列组合可以互相转换的答案称为“本质上相同的”， 否则就是“本质上不同的”。

这当然还不是一个严格的定义。现在我们打算从每个“本质上相同的”答案集合里挑出一 个唯一的代表。对选择公理的信徒来说，可以挑出代表是他们的信仰，是不需要讨论的事 情。而对于我们构造主义者来说，只有给出了具体的挑选办法后，才会承认真的可以从每 个等价类里挑出一个当代表。

大致上我们可以这样挑代表：对于加减法(乘除法类似)的表达式，我们总是优先选择前 面是加法，后面是减法的那些式子，也就是类似于 a+b+c-d-e 形式的，这样的式子也可 以写作 (a+b+c)-(d+e)，对于这样形式的式子里的两个子表达式，以及只有加法情况下的 一个表达式，我们优选相加的所有子表达式的“字符串值”是递增的那些，就是说，我们 选 (1+2+3)-(4+5)，而不选 (2+1+3)-(5+4)或其他。这些相加的子表达式可能是数字，也 可能是乘除法构成的表达式。所谓“字符串值”就是子表达式写出来的样子，而不是算出 来的值，例如，2+1/3+4*5 里的子表达式的“字符串值”分别是“2”，“1/3”，和 “4*5”，因此我们会选择等价的 1/3+2+4*5 作为代表。这里“字符串值”的使用是非常 任意的，其实只是为了达到选出来的结果唯一的目的，具体选出哪个无关紧要，因为都是等价的。

我们把从一个等价类里如上挑选出的代表称为这个等价类的“范式”。我们不去严格定义 挑选的过程，而是定义“范式”的结构。下面我们用 $|E_i|$ 表示表达式 E_i 的字符 串值，并且省略了表示运算符结合的括号。这个定义是递归的：

满足下列条件之一的表达式称为“范式”：

数字(NUM) ：由一个数字构成的表达式。

加法范式(addform) ：形如 $E_1 + E_2 + \cdots E_n, n \geq 2$，其中每个 E_i 都是一个数字，或者一个乘法范式，或者一个除法范式。而且，如果 $i < j$， 那么 $|E_i| \leq |E_j|$。

减法范式(subform) ：形如 $E_1 - E_2$，E_i 是一个数字，或者一个乘法 范式，或者一个除法范式，或者一个加法范式。

乘法范式(mulform) ：形如 $E_1 * E_2 * \cdots E_n, n \geq 2$，其中每个 E_i 都是一个数字，或者一个加法范式，或者一个减法范式。而且，如果 $i < j$， 那么 $|E_i| \leq |E_j|$。

除法范式(divform) ：形如 E_1/E_2，E_i 是一个数字，或者一个加法 范式，或者一个减法范式，或者一个乘法范式。

现在我们可以宣称：不同的范式代表本质上不同的答案，而且两个本质上不同的答案也一 定有不同的范式代表它们。因此，我们只要找出所有范式形式的答案，就可以避免答案的 重复，也不会有遗漏。

15 解决这个新问题

我们注意到，在范式的定义中，加法范式和乘法范式是很类似的，减法范式和除法范式也 是很类似的，实际上如果把其中的“加”字和“乘”字互换，同时把“减” 字和“除” 字互换，同时把相应的运算符也互换，对应的规则也就互换了。因此，我们打算用两个模 板 nform1<> 和 nform2<> 来实现这两种结构，把那些需要互换的东西当成模板参数。 而且注意到在加减法范式中，乘除法范式总是一起出现的，不必区分，我们可以把加减法 范式合为一个，称为 addsubform，同样把乘除法合为一个 muldivform：

规则 nform1<>, nform2<>, addsubform, muldivform ： 和 expr 一样，留一个值在 vals 栈顶。

先不管 nform1<> 的实现，其他的大致是这样：

```
template<Operator op, typename sform> struct nform1; //实现暂略

template<Operator op, Operator sop, typename sform>
struct nform2: product<OP<sop>,
                       sum<NUM, sform, nform1<op, sform> >,
                       sum<NUM, sform, nform1<op, sform> >,
                       REDUCE
> {};

template<Operator op, Operator sop, typename sform>
struct nform: sum<nform1<op, sform>, nform2<op, sop, sform> > {};

struct addsubform: nform<&Value::operator+, &Value::operator-, muldivform> {};
struct muldivform: nform<&Value::operator*, &Value::operator/, addsubform> {};

struct expr: sum<NUM, addsubform, muldivform> {};
```

看起来很漂亮，不过有个小问题：编译不过去，因为 addsubform 的定义里引用了还没有 定义的 muldivform。把 muldivform 的定义放到前面也不行，因为 muldivform 的定义 里也要用到 addsubform。这是一个鸡和蛋的问题。能解决吗？当然能。如果这个问题没 法解决，也许我就不会写这篇文章了。解决的办法是，用模板：

```
template<typename muldivform>
struct addsubform_: nform<
        &Value::operator+, &Value::operator-, muldivform
> {};

struct muldivform: nform<
        &Value::operator*, &Value::operator/, addsubform_<muldivform>
> {};

struct addsubform: addsubform_<muldivform> {};
```

这个解决办法就是把被依赖的东西当成模板参数，这样就可以把间接递归化为直接递归， 而直接递归是可以直接实现的。对于现在这个具体的小问题，不用 addsubform_<> 模板，直接把 addsubform 的定义抄一遍也是可以解决的：

```
struct muldivform: nform<&Value::operator*, &Value::operator/,
        nform<&Value::operator+, &Value::operator-, muldivform>
> {};

struct addsubform: nform<&Value::operator+, &Value::operator-, muldivform> {};
```

因为有代码结构的重复，这样写是不好的。对于更一般的结构，用模板化才是正确的解决 之道。(不过，对于 Borland C++ 用户，似乎只能用不好的那种形式!)

现在只剩下 nform1<op,sform>，也就是用运算符 op 连起来的不止一个 sform 或 NUM 的“字符串值”不减的串，没有实现了。那样的串的语法是容易实现的，例如：

$$nform1\texttt{<op, S>} = op\ S\ S \mid op\ nform1\texttt{<op, S>}\ S$$

就是一个满足要求的产生式。问题是现在不光是形式上有要求，对那些子表达式的语义值 的大小顺序也有要求。为了对子表达式的大小顺序进行判断，我们需要在 Universe 中引 入新的数据，记录当前表达式的最后一个子表达式的字符串值。因为我们处理的子表达式 结构是递归的，我们这个 Universe 的成员 subs 也必须是一个栈：

```
struct Universe {
        std::vector<int> nums;
        std::stack<Value> vals;
        std::stack<Operator> ops;
        std::stack<std::string> subs;
};
```

在产生 nform1 的第一个子表达式后，就把它的值放到 subs 栈顶，以后每加一个子表达 式，都和 subs 栈顶比较，如果小于 subs 栈顶的值，那么就没有结果，否则把 subs 栈 顶改为新的子表达式的值，并接受这个结果。整个表达式产生完后，把 subs 栈顶的值丢 掉。

由于表达式规则都是把表达式的值放在 vals 栈顶，因此我们定义以下几条规则：

规则 PUSH_SUB ：把 vals 栈顶的表达式的字符串值放到 subs 栈顶。

规则 POP_SUB ：把 subs 栈顶的一个值抛掉。

规则 CHECK_ORDER ：如果 vals 栈顶的表达式的字符串值小于 subs 栈顶 的值，那么没有结果。否则，把 subs 栈顶的值换为 vals 栈顶表达式的 字符串值。

这三条规则的实现是容易的：

```
struct PUSH_SUB {
    static void apply(Universe& u, Multiverse& m)
    {
        u.subs.push(u.vals.top().s);
        m.result(u);
    }
};

struct POP_SUB {
    static void apply(Universe& u, Multiverse& m)
    {
        u.subs.pop();
        m.result(u);
    }
};

struct CHECK_ORDER {
    static void apply(Universe& u, Multiverse& m)
    {
        if (u.vals.top().s < u.subs.top()) return;
        u.subs.top() = u.vals.top().s;
        m.result(u);
    }
};
```

现在来实现 nform1<>。我们用一个 nform1_<> 做主要的事情，nform1<> 做一些辅助的事情。注意，nform1_<> 总是留一个值在 subs 栈顶。

```
template<Operator op, typename S>
struct nform1_ : sum<
        product<OP<op>, S, PUSH_SUB, S, CHECK_ORDER, REDUCE>,
```

```
        product<OP<op>, nform1_, S, CHECK_ORDER, REDUCE>
> {};

template<Operator op, typename sform>
struct nform1 : product<nform1_<op, sum<NUM,sform> >, POP_SUB> {};
```

这样就完成了我们的程序：24-s2.cpp
编译运行一下，输入数据 24 1 5 5 5e，得到输出：

((5-(1/5))*5)

很好，没有以前的“重复”结果了。现在输入 100 13 14 15 16 17e 试试：

(((13+15)*(17-14))+16)
(16-((13+15)*(14-17)))
(((14+15)*(17-13))-16)

在我们对“本质上不同”的定义下，这三个答案都是不同的，但是也许有人会认为第一和 第二个答案是相同的。我们这里不再处理这种“重复”了，留作练习。

16 一些调试程序的方法

用 smonde 库的程序也是标准的 C++ 程序，你以前熟悉的调试方法自然仍然适用。
不过，由于规则的构造方法的大量使用，如果你跟踪程序的话，可能发现很多时候都是在 sum<>::apply() 或者 product<>::apply() 中，造成设置断点的困难。
一种办法是在手工写的规则的 apply() 函数中设置断点，不过这样有时可能难以发现规 则组合的错误。
我们这里介绍一种在规则中间插入检查点的方法。所谓插入检查点，就是在需要检测的地 方插入一条功能上相当于 one 的规则，但是它有一些副作用，例如可以输出一些 Universe 的状态，或者，就为了有地方可以设置断点。
下面是一个可以用于显示执行过程的规则改造模板示例：

```
template<typename Rule>
struct inspect {
    template<typename Universe>
    static void apply(Universe& u, Multiverse<Universe>& m)
    {
        std::cerr   << typeid(Rule).name()
                    << ": nums.size()=" << u.nums.size()
                    << ", vals.size()=" << u.vals.size()
                    << ",  ops.size()=" << u. ops.size()
                    << "\n";
        Rule::apply(u, m);
    }
};
```

这个规则改造的结果等价于原来的规则 Rule，不过加了一些副作用，显示了一下当前的 Universe 的状态。假如我们想检查执行规则 NUM 和 REDUCE 前的状态，那么可以如下修 改 expr：

```
struct expr: sum<inspect<NUM>,
                 product<any_op, expr, expr, inspect<REDUCE> >
> {};
```

插入检查代码的方法很多，上面只是给出一种可能，实践当中相信还会有更适用的办法。
象 inspect<> 那样的模板跟踪起来可能不是很方便，因为不同的实例化执行的是同一个 地方的代码。为了跟踪方便，可以专门为某个检查点写一个与 one 等价的规则，用于设 置断点。

17 这个简化模型的优缺点以及下一步的任务

我们从一句对不确定性计算的抽象描述开始，首先把这个描述具体化为 C++ 中的三个概 念：Universe, Multiverse 和 Rule。然后围绕这三个概念，我们写出了基本的规则 zero，one，以及通用的也是基本的规则组

合方法 sum<> 和 product<>，完成了一个 规则上的 Chomsky type-2 语言定义系统。这样就建立起了一个完整的不确定性计算模型 及其支持系统。

我们描述了使用这个不确定性计算模型写出来的程序的一般结构，然后按此写了一个计算 24 的简单程序，初步展示了不确定性算法描述的基本特征。

进一步把算 24 的程序加上只能输出“本质上不同”答案的要求，我们改写了那个程序。 从改写过程，我看到使用不确定性计算模型，系统会有较好的可扩展性，而且这个模型确 实很适合于描述不确定性计算，尤其是问题的解有较复杂的结构的那种。

从上面的实践中，我们看到了这个系统的很多好处：规则组合是声明式的，不需要写可执 行语句。这些声明是真正的等式，可以进行等量代换，虽然本文中我们还没有用到这个特 性。这个模型也有强大的搜索能力，很大程度上可以抵消那种在 C++ 中再实现一个 Prolog 的冲动。这个模型完美地结合了指令式程序设计(imperative programming，写 基本规则)和声明式程序设计(declarative programming，规则组合)方法。规则的构造 方法本身也是可以复用的。

如果把这个模型和量子计算机[quantum]比较的话，它们的计算步骤是相似的。 一个不同点是，这个模型是用反复回溯的办法来尝试所有的可能性，而量子计算机是一次 性算出所有可能的结果。另一个不同点是，在这个模型里对最终结果的判断是轻而易举的， 而在量子计算中，“看到自己想看到的结果”是一件很麻烦的事情。

这个模型也有一些缺点。这些缺点大多和我们做的简化有关：我们要求一条规则不能改变 宇宙的类型。这样做的后果是，Universe 中必须保罗万象，即所有规则可能用到的数据 结构都必须从头到尾存在于 Universe 中。这就大大增加了规则间的耦合性，也就降低了 规则的可复用性。

我们要求 Universe 类型不变后，就把 Multiverse 的类型也限制在一个特定类型 (Multiverse< *Universe* >) 的派生类型上了，然后 Rule 的 apply() 函数的类型也就被 固定下来了。前面我们并没有说明为什么要这么做。以后我们将看到，这个简化让我们避免 了很多复杂的类型问题，因此对于展示不确定计算的基本特征来说是恰当的决定。这种类 型上的简化也让我们付出了效率上的代价：Multiverse 是一个抽象的类型，虽然我们避 免了用 new 来创建 Multiverse 对象，但对 result() 的每次调用仍是一个虚函数调用， 这样就造成了优化的屏障。在大量简单规则进行组合的情况下，这个代价也许是非常高的。

简化模型的两大缺点(Universe 必须包罗万象，Multiverse 是抽象的)正是我们下一次要 解决的核心问题。下次也许还会再多给出几个规则的组合方式。

相信大家都已经看到，我们写出来的不确定性算法程序和语法分析程序非常象(只是有点 象是在倒着运行)。这不是巧合。一方面的(主要的)原因是，产生式系统本来就是一个和 不确定图灵机等价的系统，“顺序”和“选择”也是宇宙间最基本的组合模式。另一方面 的原因是，我本来就是想写一个语法分析程序的构造工具，写完后发现用它算 24 挺方便 的，就又改了方向，从更一般的不确定性计算的角度重新建立了系统的模型，这样更加通 用化以后，也得到了一个更加简单的模型。因此，我们以后要介绍一个语法分析程序的合 成系统，是基于没有简化的不确定性计算模型的。

还有一个问题不知你有没有想过：那些 apply() 函数到底是怎样一直调用到 Destiny 的？ 答案是：它们是一直嵌套和递归调用到 Destiny 的！如果输入数据或者语义性规则多的 话，这是一件很恐怖的事情。例如，对于一个语法分析程序，可能每个输入字符都会引起 不止一层的函数调用！这对于即使是很小的语言可能都是不可接受的。这个问题也有简单 的解决办法，也留到以后说了。

写这么多实在是太累了，事实上当我写到“什么是本质上不同的答案”节时，我用的 Hugs 解释程序就已经“out of heap memory”了，不得不停下来找手册加大 heap memory 的 设置。那些实用中必须解决的问题都留待以后解决吧。现在的系统对于解决算 24 之类的 小问题还算是可以应付的，因此我们也可以感到一丝欣慰。你也累了，歇吧。

18 本文中写的程序代码

这里是和本文有关的全部源程序：

24-s0.cpp： 简单的计算 24 的程序
smonde.h： 简单不确定性运算模型的通用库
24-s1.cpp： 使用 smonde 的算 24 的程序
24-s2.cpp： 使用 smonde 输出本质不同答案的程序

参考资料

[主要是 Wikipedia 的广告!]

[turing] Turing Machine

[concept] David Abrahams 写，Merlin Ran 译，泛型编程技术

[multiverse] Wikipedia, Multiverse

[meta] 邝芷人，哲学讲话.5

[chomsky] Wikipedia, Chomsky Hierarchy

[kleene] Wikipedia, Kleene Algebra

[quantum] Wikipedia, Quantum Computer