编程语言的博弈语义

尹昊萱，本科毕业于清华大学计算机专业，获得逻辑学交叉学科课程证书，现于牛津大学计算机系读博。

一段程序（program）的含义是什么？

这看起来并不是一个值得研究的问题，毕竟我们每个人学习写程序的时候都是从学习编程语言的“含义”开始的。比如说，“4”这个程序的含义就是数值 4，“2+3”这个程序的含义就是对 2 和 3 这两个数值求和，“f(x)=x+1”这个程序的含义则是一个函数：输入一个值，输出这个值+1。

然而，这种解释方式也存在不令人满意之处：有许多本质上含义相同的程序，按照这种逐字逐句的方法解读出来的含义却不同。例如，“4”的含义是数值 4，“2*2”的含义是对 2 和 2 这两个数值相乘的结果，“let f(x)=x+1 in f(3)”的含义则是先定义一个+1函数，再计算这个函数输入为 3 时的值。然而，这三个程序在本质上都是一回事：它们都表示数值 4。

读到这里，你可能会产生两点疑问。第一，为什么说上面的三个程序本质相同呢？有的人可能认为，只要两个程序做的事情不同，它们本质上就是不同的。例如，第一个程序没有做数值运算，第二个程序只做了乘法，第三个程序只做了加法。第二，如果想让上面三个程序本质相同，是不是直接把程序运算后的结果当作它的含义就行了呢？至少在上面的例子中，三个程序运算得到的结果都是 4，因此可以说它们的含义是相同的。

为了回答这两个问题，我们考虑两个稍微复杂一些的C++程序：

```
class Pos {
    int x=0;
public:
    void count() {++x;}
    int get() {return x;}
};
```

```
class Neg {
    int x=0;
public:
    void count() {--x;}
    int get() {return -x;}
};
```

如果你没有学过面向对象编程，可以把上面的两个程序形象地理解成两个遥控器 Pos 和 Neg。每个遥控器上各有一块屏幕和两个按钮，两个按钮上分别写着 count 和 get。遥控器 Pos 的内部构造如下：它存储着一个初始值为 0 的变量，每次按下 count 键，就把变量的值加一；每次按下 get 键，就在屏幕上显示变量当前的值。遥控器 Neg 的内部构造是：它存储着一个初始值为 0 的变量，每次按下 count 键，就把变量的值减一；每次按下 get 键，就在屏幕上显示变量当前的值的相反数。

现在，我的问题是：给你一个遥控器，你可以随便把玩它，但是不能拆开来看，你能分辨出手里的遥控器是哪一种吗？

例如，你可能想做如下的事情：先按一下 get 键，观察屏幕上显示的数字，再按三下 count 键，再按一下 get 键，观察屏幕上显示的数字。那么，无论你手里拿的是哪种遥控器，发生的事情都可以表示成下面的序列：

```
get !0 count count count get !3
```

这个序列里的每个元素是一个行为（action），以感叹号“!”开头的是程序的行为（此处即显示数字），不以“!”开头的是你的行为（此处即按下某个按钮）。[1]

> [1] 严格来说，count 也是一个询问的行为，因此每次按下 count 都应该有一个类型为 unit（或 void）的返回值，不过这里为了简洁而省去了。

既然在两个遥控器上发生的事情是同样的，我们自然也就无法区分手里拿的到底是哪一个。你可能还想多试几次，以不同的顺序和次数来按按钮，但很显然，无论你怎样按，两个遥控器的表现总是相同的。

于是，我们可以很自然地说：

如果两个遥控器用起来是一样的，那么它们的含义就是相同的。

插一句题外话，这似乎与语言哲学里维特根斯坦“Meaning as Use”的观点也有共通之处：

> For a *large* class of cases of the employment of the word ‘meaning’—though not for all—this word can be explained in this way: the meaning of a word is its use in the language. (*Philosophical Investigations* 43)

当然，我们还要把“用起来”这一说法变得更精确一些——总不能每次都把程序当成遥控器。我们在写程序的时候是怎么“用”一段程序的呢？最简单的理解是，我们“调用”它暴露出来的接口。例如此前的测试，“先按一下 get 键，观察屏幕上显示的数字，再按三下 count 键，再按一下 get 键，观察屏幕上显示的数字”，就可以看作如下代码：

```
cout << a.get();
a.count();
a.count();
a.count();
cout << a.get();
```

现在我们可以回答最开始提出的两个问题了。第一，两个程序明明做的事情不一样，为什么说它们是等价的呢？因为我们采用的是外部而非内部的视角。只要两个程序从外面看起来做的事情是一样的，我们就认为它们是等价的，而不去关心它们内部究竟是怎么做这些事情的。[2] 第二，能不能直接把程序运算后的结果当成它的含义呢？在这个例子中我们可以看到，Pos 和 Neg 两个类都已经是最简了，没有一个能简化到另外一个，但是它们仍然可以是等价的。

> [2] 当然，内部和外部是两种不同的视角，在解决不同的问题上有不同的优势，没有对错之分。

在上面的例子里，我们比较的是两个类。事实上，任何两段程序都可以用提供**上下文（context）**的方法进行比较。比如，“f(x)=x+x”和“f(x)=x*2”这两段程序在一般情况下都是等价的。为了验证这一点，我们考虑对这两段程序的“使用”。例如：

```
int x = 1;
int y = _(x);
cout << y;
```

其中，“_”表示插入当前考虑的程序的位置。上面提到的两个程序插入之后都有相同的结果：输出 2。当然，只凭这一点还不足以说明这两个程序是等价的。比如，把“f(x)=x+1”这个程序插入上面的上下文中，输出的结果也是 2，但很显然这个程序并不和之前提到的两个程序等价。只有当两个程序在任何上下文中的表现都相同时，我们才能说它们是等价的。例如，如果把上面三个程序放到下面这个上下文中，

```
int x = 2;
int y = _(x);
cout << y;
```

那么“f(x)=x+x”和“f(x)=x*2”的输出结果都是 4，而“f(x)=x+1”的输出结果则是 3。

到这里，我们可以把前面提出的“用起来”的说法精确化了：

两个程序的含义是一样的，当且仅当它们在所有上下文中的表现是一致的。

我们还可以对上下文做进一步的抽象，因为并不是上下文中的所有操作都与程序有关。在遥控器的例子中，“先按三下 count 键，再按一下 get 键，观察屏幕上显示的数字”和“先按三下 count 键，再跳一支舞，再按一下 get 键，观察屏幕上显示的数字”这两种操作对于我们观察程序来说并没有本质区别，因为操作者跳舞的行为并不与遥控器发生关系。在后一个例子中，

```
int x = 1;
int y = _(x);
cout << y;
```

和

```
int x = 100;
cout << _(1);
```

这两个上下文对于程序来说同样没什么区别。

总而言之，我们既不关心程序内部发生了什么，也不关心上下文自己做了什么，我们只关心程序与上下文之间的互动。在这个意义上，我们也可以把上下文看成是与程序发生交互的环境（environment）。不管程序是之前提到的哪一个，也不管上下文是上面提到的哪一个，它们之间的交互都可以概括成如下的问答。

```
环境：你是什么？
程序：我是一个函数。
环境：我把1输入给你，你的计算结果是什么？
程序：结果是2。
```

这段对话可以像遥控器的例子中一样概括成一个动作的序列：

```
_ !f f(1) !2
```

在第一步中，程序返回的理应是自身的值。但是它并没有直接返回一段具体的程序，而是犹抱琵琶半遮面地给自己取了一个名字叫作 f，把这个名字返回给了环境。接下来环境问什么，程序答什么，但环境永远只能知道程序在某些具体点上的值，而看不到程序本身。这个特性与我们“从外部看”的视角是一致的。只有这样，“f(x)=x+x”和“f(x)=x*2”这两个内部结构不一样的函数才会显得一样。

程序和环境之间的交互还可以更复杂一些。例如，程序“f(g,x)=g(x)+1”与环境“_(g(x)=x+1,1)”之间可能发生如下的对话：

```
环境：你是什么？
程序：我是一个有两个参数的函数，第一个参数是一个有一个参数的函数，第二个参数是一个数。
环境：我把函数g和数字1输入给你，你的计算结果是什么？
程序：我把数字1输入给你提供的函数g，计算结果是什么？
环境：结果是2。
程序：你之前的问题的结果是3。
```

这段对话可以概括成：

```
_ !f f(g,1) !g(1) 2 !3
```

像上面这样的一个动作序列，我们称为程序的一个迹（trace）。要考虑一个程序与所有可能的上下文的交互，本质上就是考虑所有这个程序可能生成的迹。

终于，我们可以说：

一个程序的含义，就是它能生成的所有迹的集合。

最后，读者可能会自然地产生下述想法：给定某程序，是否可以计算其所有的迹？答案显然是否定的，因为一个不平凡的程序与环境之间的互动有无限种可能（例如，可以给一个函数提供无限种参数），所以它的迹的集合是无穷的。一个更合适的问题可能是，是否可以计算给定的程序的所有迹的某种表示，使得我们能够通过比较两个程序的这个表示来判定两个程序是否等价？遗憾的是，这个问题的答案一般仍然是否定的，因为程序的等价性是不可判定的——否则我们就可以通过与一个死循环程序进行比较来判定停机问题。不过，我们也确实可以利用这个方法找到一些可以判定的编程语言片段，这方面最新的研究是

> Bunting, Benedict, and Andrzej S. Murawski. "Operational Algorithmic Game Semantics." *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 2023.

参考资料

如果想对编程语言的博弈语义有更多的了解，

> Abramsky, Samson, and Guy McCusker. "Game semantics." *Computational Logic: Proceedings of the*

> *NATO Advanced Study Institute on Computational Logic, held in Marktoberdorf, Germany, July 29–August 10, 1997*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999. 1-55.

的前半部分含有大量例子和形象的说明，对形成直观理解很有帮助。但是这篇文章后半部分的形式定义需要用到范畴论的知识。如果想要一个形式上最简单的规范定义，可以参考

> Laird, James. "A fully abstract trace semantics for general references." *International Colloquium on Automata, Languages, and Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007.

的前半部分。如果你还想了解其他语言上更复杂的博弈语义定义，可以参考（按照难度递增的顺序）：Laird文章的后半部分、

> Jaber, Guilhem, and Andrzej S. Murawski. "Complete trace models of state and control." *Programming Languages and Systems: 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27–April 1, 2021, Proceedings 30*. Springer International Publishing, 2021.

和

> Koutavas, Vasileios, Yu-Yang Lin, and Nikos Tzevelekos. "Fully Abstract Normal Form Bisimulation for Call-by-Value PCF." *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 2023.

历史上，博弈语义最早在

> Abramsky, Samson, Radha Jagadeesan, and Pasquale Malacaria. "Full abstraction for PCF." *Information and computation* 163.2 (2000): 409-470.

和

> Hyland, J. Martin E., and C-HL Ong. "On full abstraction for PCF: I, II, and III." *Information and computation* 163.2 (2000): 285-408.

两篇文章中被提出，这两篇文章都使用了范畴论。