arXiv:2304.05039v1 [cs.LO] 11 Apr 2023

执行轨迹与归约序列

Gilles Dowek*

1 引言

Y. Gurevich [6] 提出了一种定义算法概念的方法，将算法定义为一组执行轨迹。这个定义使我们能够在引入程序概念之前引入算法概念，并且不依赖于任何特定的编程语言。它还使我们能够区分

- 程序是什么：一段文本，
- 程序做什么：一种算法，
- 程序所计算的内容：一个将输入值映射为输出值的函数。

本文主张，这种将算法视为执行轨迹集合的概念，在某种程度上独立于抽象状态机的概念——尽管它与后者十分契合。它可以在更一般的小步操作语义框架中重新表述 [8]。

在小步操作语义的背景下重新表述这一想法，可以定义执行轨迹的概念，不仅适用于命令式程序，也适用于函数式程序（包括高阶函数式程序），从而为关于将算法定义为执行轨迹集合和递归方程的争议带来新的认识 [7, 3]。

2 将算法视为执行轨迹集合

2.1 执行轨迹集合

考虑一种算法：通过不断将一个自然数除以 2，直到得到奇数，从而计算其最大奇数因子。

该算法可以定义为轨迹集合

$$\langle 12\rangle, \langle 6\rangle, \langle 3\rangle$$
$$\langle 8\rangle, \langle 4\rangle, \langle 2\rangle, \langle 1\rangle$$
$$\langle 7\rangle$$
$$\dots$$

以这种方式定义的算法不必是确定性的。例如，通过将一个自然数除以任意一个属于 2 的幂的因子，直到得到奇数，从而计算其最大奇数因子的非确定性算法，可以定义为以下序列的集合

$$\langle 12\rangle, \langle 3\rangle$$
$$\langle 12\rangle, \langle 6\rangle, \langle 3\rangle$$
$$\langle 8\rangle, \langle 4\rangle, \langle 1\rangle$$
$$\langle 8\rangle, \langle 4\rangle, \langle 2\rangle, \langle 1\rangle$$
$$\langle 7\rangle$$
$$\dots$$

*Inria 和 ´巴黎-萨克雷高等师范学´校，威尔逊总统大道 61 号，94235 卡尚 Cedex，法国，gilles.dowek@ens-paris-saclay.fr。

2.2 两种算法的同一性

此定义还给出了两个算法的同一性概念 [2]。例如，第 2.1 节中计算自然数的最大奇数因子的两个算法彼此不同，并且也不同于包含这些序列的那个算法

$$\langle 12\rangle, \langle 12, 1\rangle, \langle 12, 2\rangle, \langle 12, 4\rangle, \langle 3, 4\rangle, \langle 3\rangle$$

$$\langle 8\rangle, \langle 8, 1\rangle, \langle 8, 2\rangle, \langle 8, 4\rangle, \langle 8, 8\rangle, \langle 1, 8\rangle, \langle 1\rangle$$

$$\langle 7\rangle, \langle 7, 1\rangle, \langle 7, 1\rangle, \langle 7\rangle$$

$$\ldots$$

3 从小步操作语义到执行轨迹

并非所有序列集合都是算法。例如，包含自然数序列 $\langle\langle n_1\rangle, \langle n_2\rangle\rangle$ 的集合，其中 $n_2 = 1$ 如果 n_1 是终止程序的G¨odel数而$n_2 = 0$ 否则不是，因为执行轨迹这一概念预设了可有效计算性的概念。

这种有效性的概念可以在小步操作语义框架中定义。

3.1 函数式语言的小步操作语义

小步操作语义最初是为函数式编程语言定义的。

例如，要定义程序 fun x → x * x + 1 在值 7 上的执行结果，我们首先构造项 (fun x → x * x + 1) 7，它同时包含程序和输入值，然后通过重写规则一步一步地将其归约为不可再归约的项 50；这些重写规则定义了语言语义。

例如，使用 β-归约规则以及加法和乘法的显然规则，我们得到以下归约序列

$$\texttt{(fun x} \to \texttt{x * x + 1) 7} \longrightarrow \texttt{7 * 7 + 1} \longrightarrow \texttt{49 + 1} \longrightarrow \texttt{50}$$

同样，并非所有序列集合都是归约序列集合。若存在一个可计算函数，能够根据序列的前一个元素计算出下一个元素（在非确定性情况下，则计算出前一个元素之后所有可能的下一个元素组成的有限集合），那么该序列集合就是归约序列集合。

定义 1（归约序列集合）如果存在一个初始状态集合 I 和可计算函数 step，使得 R 是所有序列 $s_1, ..., s_n$ 的集合，并且这些序列满足：s_1 是 I 的一个元素，s_n 满足 step(s_n) $= \varnothing$，且对于所有 i，$s_{i+1} \in$step(s_i)，那么序列集合 R 就是归约序列集合。

可计算函数 step 定义了规约的粒度。例如，一个 β-规约步骤可以被视为原子的，或者被视为一系列步骤，其长度取决于抽象体的大小，就像显式替换演算 [1] 中那样。同样，变量赋值可以被视为一个原子步骤，或者被视为一系列步骤，其长度以对数方式取决于内存的大小。

这种粒度的选择会影响归约序列的长度，进而影响算法的复杂度。因此，算法的复杂度取决于对这类函数步骤的选择。

3.2 命令式语言的小步操作语义

一个命令式程序，例如

```
while even(x) do x := x / 2
```

$$\langle \text{跳过;r},\rho\rangle \longrightarrow \langle \text{r},\rho\rangle$$

$$\langle \text{x := t;r},\rho\rangle \longrightarrow \langle \text{r},\ \rho + (\text{x} = [\![\text{t}]\!]_\rho)\rangle$$

$$\langle (\text{p;q});\text{r},\rho\rangle \longrightarrow \langle \text{p;(q;r)},\rho\rangle$$

若 $[\![\text{t}]\!]_\rho =$ 为 true，则

$$\langle \texttt{if t then p else q;r},\rho\rangle \longrightarrow \langle \texttt{p;r},\rho\rangle$$

若 $[\![\text{t}]\!]_\rho =$ 为 false，则

$$\langle \text{若 t 为真，则执行 p，否则执行 q;r},\rho\rangle \longrightarrow \langle \text{q;r},\rho\rangle$$

$$\langle \text{while t do p;r},\rho\rangle \longrightarrow \langle (\text{if t then p;while t do p else skip});\text{r},\rho\rangle$$

图1：简单命令式语言的小步操作语义

单独来看，无法通过小步操作语义规则进行归约。但该程序与某个状态（例如 $\langle \text{x} = 12\rangle$ ）组成的有序对，可以归约为由空程序和状态 $\langle \text{x} = 3\rangle$ 组成的有序对。简单命令式语言的小步操作语义如图1所示，其中 r 可以为空，此时 p;r 就是 p。不可约项是程序为空的项。例如，该程序在状态 $\langle \text{x} = 12\rangle$ 下的归约序列为序列

$\langle$当 even(x) 时执行 x := x / 2, $\langle \text{x} = 12\rangle\rangle \longrightarrow \langle$如果 even(x)，则执行 x := x /2；当 even(x) 时执行 x := x / 2，否则跳过 $\langle \text{x} = 12\rangle\rangle \longrightarrow \langle$x := x /2；当 even(x) 时执行 x := x / 2, $\langle \text{x} = 12\rangle\rangle \longrightarrow \langle$当 even(x) 时执行 x := x / 2, $\langle \text{x} = 6\rangle\rangle \longrightarrow \langle$如果 even(x)，则执行 x := x /2；当 even(x) 时执行 x := x / 2，否则跳过 $\langle \text{x} = 6\rangle\rangle \longrightarrow \langle$x := x /2；当 even(x) 时执行 x := x / 2, $\langle \text{x} = 6\rangle\rangle \longrightarrow \langle$当 even(x) 时执行 x := x / 2, $\langle \text{x} = 3\rangle\rangle \longrightarrow \langle$如果 even(x)，则执行 x := x /2；当 even(x) 时执行 x := x / 2，否则跳过 $\langle \text{x} = 3\rangle\rangle \longrightarrow \langle$跳过 $\langle \text{x} = 3\rangle\rangle \longrightarrow \langle, \langle \text{x} = 3\rangle\rangle$

3.3 从归约序列到执行轨迹

命令式程序的执行轨迹这一概念可以很容易地从这样的归约序列中定义：如果归约序列是

$$\langle \text{p}_1, \rho_1\rangle \longrightarrow \langle \text{p}_2, \rho_2\rangle \longrightarrow \ \ldots\ \longrightarrow \langle \text{p}_\text{n}, \rho_n\rangle$$

t当 p_1 在 ρ_1 中的执行轨迹，是通过删除每个中的程序而得到的序列。 h pair 并擦除状态中的变量名。

例如，该程序的执行轨迹

```
while even(x) do x := x / 2
```

在状态 $\langle \text{x} = 12\rangle$ 中是

$$\langle 12\rangle, \langle 12\rangle, \langle 12\rangle, \langle 6\rangle, \langle 6\rangle, \langle 6\rangle, \langle 3\rangle, \langle 3\rangle, \langle 3\rangle, \langle 3\rangle$$

更一般地说，执行轨迹是归约序列的任意投影。这排除了计算不可计算函数的算法，并保证了有效性。

定义 2（投影）设 proj 是一个可计算函数。若 s 是序列 s_1,..., s_n，我们将 proj(s) 记作对该序列中各元素逐点应用函数 proj：proj(s_1) ..., proj(s_n)。

若 R 是一个序列集合，我们将 proj(R) 记作对 R 中各元素逐点应用函数 proj。

我们称集合 A 是集合 R 的投影，如果存在一个可计算函数 proj，使得 A =proj(R)。

3.4 加速

执行轨迹集合

$$\langle 12\rangle, \langle 12\rangle, \langle 12\rangle, \langle 6\rangle, \langle 6\rangle, \langle 6\rangle, \langle 3\rangle, \langle 3\rangle, \langle 3\rangle, \langle 3\rangle$$

$$\langle 8\rangle, \langle 8\rangle, \langle 8\rangle, \langle 4\rangle, \langle 4\rangle, \langle 4\rangle, \langle 2\rangle, \langle 2\rangle, \langle 2\rangle, \langle 1\rangle, \langle 1\rangle, \langle 1\rangle, \langle 1\rangle$$

$$\langle 7\rangle, \langle 7\rangle, \langle 7\rangle, \langle 7\rangle$$

与第 2.1 节中的执行轨迹集合不同。

但是，序列

$$\langle 12\rangle, \langle 6\rangle, \langle 3\rangle$$

是序列的子序列

$$\langle 12\rangle, \langle 12\rangle, \langle 12\rangle, \langle 6\rangle, \langle 6\rangle, \langle 6\rangle, \langle 3\rangle, \langle 3\rangle, \langle 3\rangle, \langle 3\rangle$$

并且连续省略步骤的数量上限为 3。

这引出了对一个序列集合的加速概念的定义。

定义3（加速）一组序列 B 是一组序列 A 的加速，当且仅当存在一个数 k，使得 B 中的每个序列都是 A中某个序列的子序列，包含相同的首元素和末元素，并且连续省略元素的数量受 k 限制。

加速会保持有效性，复杂度也会保持不变，因为执行轨迹的长度最多乘以或除以常数 k。

现在我们可以正式定义算法这一概念。

定义 4（算法）算法是一组序列，它是归约序列集合的投影的加速。

例如，包含所有由程序构成的对的初始集合

```
while even(x) do x := x / 2
```

以及某个自然数 n对应的状态 $\langle \mathrm{x} = n\rangle$，还有图1中定义的步函数，共同定义了一个归约序列集合。投影函数 proj 擦除程序和变量名，定义了一种包含这些轨迹的算法。

$$\langle 12\rangle, \langle 12\rangle, \langle 12\rangle, \langle 6\rangle, \langle 6\rangle, \langle 6\rangle, \langle 3\rangle, \langle 3\rangle, \langle 3\rangle, \langle 3\rangle$$

$$\langle 8\rangle, \langle 8\rangle, \langle 8\rangle, \langle 4\rangle, \langle 4\rangle, \langle 4\rangle, \langle 2\rangle, \langle 2\rangle, \langle 2\rangle, \langle 1\rangle, \langle 1\rangle, \langle 1\rangle, \langle 1\rangle$$

$$\langle 7\rangle, \langle 7\rangle, \langle 7\rangle, \langle 7\rangle$$

具有一个包含轨迹的加速算法

$$\langle 12\rangle, \langle 6\rangle, \langle 3\rangle$$

$$\langle 8\rangle, \langle 4\rangle, \langle 2\rangle, \langle 1\rangle$$

$$\langle 7\rangle$$

$$\ldots$$

因此，这两种算法都由该程序表示

```
while even(x) do x := x / 2
```

因此，该程序并不表达单一的算法，而是根据观察该程序时所采用的粒度，表达多个算法；这种粒度由步函数和加速定义。

4 函数式语言

函数式编程的核心思想是，程序不会做任何事情，只会计算出某些结果。因此，人们常常认为，执行轨迹这一概念——也就是程序所做的事情——对于函数式语言来说毫无意义。

函数式程序缺乏执行轨迹这一概念，可能正是围绕将算法概念定义为执行轨迹集合 [6] 以及递归方程 [7]的争议 [3] 的根源。

但是，正如我们所看到的，函数式语言具有小步操作语义；而且，正如我们将要看到的，这种小步操作语义可以将执行轨迹的概念扩展到函数式程序。

4.1 递归方程

Consider，例如，计算某个自然数的最大奇因子的函数式程序
实数的数量
器，由递归方程定义

```
f(x) = if(even(x),f(/(x,2)),x)
```

或重写规则

```
f(x) ⟶ if(even(x),f(/(x,2)),x)
```

并配以用于计算奇偶性、除法和测试的显然规则。

然后，项 f(12) 归约为项 3。但是，在小步操作语义中，归约序列不能仅仅是

f(12)⟶ i
f(even(12),f(/(12,2)),12)⟶
if(true,f(/(12,2)),12)⟶ f(/(12,2))
⟶ f(6)⟶ if(
even(6),f(/(6,2)),6)⟶
if(true,f(/(6,2)),6)⟶ f(/(6,2))⟶
f(3)⟶ if(even(3),f(/(3,2)),3)⟶
if(false,f(/(3,2)),3)⟶ 3

因为 f(12) 归约为 if(even(12),f(/(12,2)),12) 这一事实并不属于语言语义，而是取决于程序。因此，与命令式语言一样，我们需要引入一个由程序（一组递归方程）和一个项组成的二元组；若用 p 表示上面这组递归方程，则归约序列为

⟨p,f(12)⟩⟶ ⟨
p,if(even(12),f(/(12,2)),12)⟩⟶ ⟨
p,if(true,f(/(12,2)),12)⟩⟶ ⟨p,f(/(12,2))⟩
⟶ ...

因此，项序列

f(12)⟶ if(even(12),f(/(12,2)),12)
⟶ if(true,f(/(12,2)),12)⟶
f(/(12,2))⟶ ...

其中包含程序名称、f、/、.....但不包含程序，是命令式语言中状态序列的对应物。

I在命令式程序的轨迹定义中，我们省略了变量名称。确切地 ly
在 th 同样，我们可以省略程序名称，得到执行轨迹

<fun>(12),<fun>(<fun>(12),<fun>(<fun>
(12,2)),12),<fun>(true,<fun>(<fun>
(12,2)),12),<fun>(<fun>(12,2)),<fun>(6),<f
un>(<fun>(6),<fun>(<fun>(6,2)),6),<fun>
(true,<fun>(<fun>(6,2)),6),<fun>(<fun>
(6,2)),<fun>(3),<fun>(<fun>(3),<fun>(<fun>
(3,2)),3),<fun>(false,<fun>(<fun>(3,2)),3), 3

4.2 λ演算风格语言

当函数式程序通过递归方程或重写规则定义时，可以将状态与程序分离，从而定义执行轨迹的概念。但在其他函数式语言中，例如 λ演算或 ρ演算，程序与数据则不太容易分离。 [4],

我们考虑 λ演算的一个扩展，其中 fun 操作是递归的，称为 fixfun， [5], 其归约规则为

```
(fun f x → t) u ⟶ (u/x,fun f x → t/f)t
```

并配有用于计算奇偶性、除法和测试的显然规则。

于是，项 (fun f x → (if (even x) (f (/ x 2)) x)) 12 归约为项 3。 (fun f x → (if (even x) (f (/ x 2)) x)) 12⟶ ((fun f x → (if (even x) (f (/ x 2)) x)) 12)⟶ (if (even 12) ((fun f x → (if (even x) (f (/ x 2)) x)) (/ 12 2)) 12)⟶ (if true ((fun f x → (if (even x) (f (/ x 2)) x)) (/ 12 2)) 12)⟶ ((fun f x → (if (even x) (f (/ x 2)) x)) (/ 12 2)))⟶ ((fun f x → (if (even x) (f (/ x 2)) x)) 6)⟶ (if (even 6) ((fun f x → (if (even x) (f (/ x 2)) x)) (/ 6 2)) 6)⟶ (if true ((fun f x → (if (even x) (f (/ x 2)) x)) (/ 6 2)) 6)⟶ ((fun f x → (if (even x) (f (/ x 2)) x)) (/ 6 2))⟶ ((fun f x → (if (even x) (f (/ x 2)) x)) 3)⟶ (if (even 3) ((fun f x → (if (even x) (f (/ x 2)) x)) (/ 3 2)) 3)⟶ (if false ((fun f x → (if (even x) (f (/ x 2)) x)) (/ 3 2)) 3)⟶ 3

为将程序与数据分开，我们把每个抽象都替换为常量 <fun> ，从而得到执行轨迹

(<fun> 12), (<fun> (<fun> 12) (<fun>(<fun
> 12 2)) 12), (<fun> true (<fun> (<fun> 12 2))
12), (<fun>(<fun> 12 2))), (<fun> 6), (<fun> (<
fun> 6)(<fun>(<fun> 6 2)) 6), (<fun> true(<f
un> (<fun> 6 2))6), (<fun>(<fun> 6 2)), (<fun
> 3), (<fun> (<fun> 3)(<fun>(<fun> 3 2)) 3), (<
fun> false(<fun> (<fun> 3 2)) 3), 3

请注意，除了一些细微的记号变化外，这条执行轨迹与第 4.1 节语言中的执行轨迹相同。

5 何时两个函数式程序表达相同的算法?

这种执行轨迹概念允许将算法定义为一组执行轨迹，并定义两个函数式程序何时执行相同的算法。根据此定义，尽管在语法上不同，两个计算布尔值上的恒等函数的程序

```
f(false) = false
 f(true) = true
```

and

```
f(x) = x
```

表达相同的算法

```
<fun>(false), false
 <fun>(true), true
```

致谢

衷心感谢工作组 Tarmac 的成员们所进行的启发性讨论。

参考文献

[1] M. Abadi、L. Cardelli、P-L. Curien 和 J-J. Levy。《显式替换》。《函数式编程杂志》，1(4):375–416，1991。[2] A. Blass、N. Dershowitz 和 Y. Gurevich。《何时两个算法相同? 》《符号逻辑通报》，12(2):145–168，2009。[3] A. Blass 和 Y. Gurevich。《算法与机器》。《欧洲理论计算机科学协会通报》，77:96–118，2002。[4] H. Cirstea 和 C. Kirchner。《重写演算——第一部分》。《IGPL 逻辑杂志》，9(3):339–375，2001。[5] G. Dowek 和 J.-J. L´evy。《编程语言理论导论》。Springer，2010。[6] Y. Gurevich。《顺序抽象状态机能够刻画顺序算法》。《ACM 计算逻辑汇刊》，2000。[7] Y.N. Moschovakis。《什么是算法? 》《数学无限》，第919–936页，2001。[8] G. D. Plotkin。《操作语义的结构化方法》。技术报告 DAIMI FN-19，奥胡斯大学计算机科学系，1981。