# Execution traces and reduction sequences

Gilles Dowek

# Execution traces and reduction sequences

Gilles Dowek*

## 1 Introduction

Y. Gurevich [6] proposes a definition of the notion of algorithm as a set of execution traces. This definition permits to introduce the notion of algorithm before that of program, and independently of any specific programming language. It also permits to distinguish

- what a program is: a text,
- what a program does: an algorithm,
- what a program computes: a function mapping input values to output values.

In this note, we defend that this notion of algorithm as a set of execution traces is somewhat independent of the notion of abstract state machine—although it fits very well with it. It can be reformulated in the more general framework of small step operational semantics [8].

Reformulating this idea in the context of small step operational semantics permits to define a notion of execution trace, not only for imperative programs, but for also for functional programs, including higher-order ones, shedding new light on the controversy on the definition of the notion of algorithm as sets of execution traces and as recursive equations [7, 3].

## 2 Algorithms as sets of execution traces

### 2.1 Sets of execution traces

Consider an algorithm that computes the largest odd divisor of a natural number by dividing it by 2 until an odd number is reached.

This algorithm can be defined as the set of traces

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

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

$$\langle 7\rangle$$

$$\ldots$$

The algorithms defined in this way need not be deterministic. For instance, the non deterministic algorithm computing the largest odd divisor of a natural number by dividing it by any divisor that is a power of 2 until an odd number is reached can be defined as the set of sequences

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

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

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

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

$$\langle 7\rangle$$

$$\ldots$$

*Inria and École normale supérieure de Paris-Saclay, 61, avenue du Président Wilson, 94235 Cachan Cedex, France, gilles.dowek@ens-paris-saclay.fr.

## 2.2 The identity of two algorithms

This definition also yields a notion of identity of two algorithms [2]. For instance the two algorithms of Section 2.1 computing the largest odd divisor of a natural number are different, and are different from that containing the sequences

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

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

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

...

# 3 From small step operational semantics to execution traces

Not all sets of sequences are algorithms. For instance, the set containing the sequences of natural numbers $\langle\langle n_1\rangle, \langle n_2\rangle\rangle$, such that $n_2 = 1$ if $n_1$ is the Gödel number of a terminating program and $n_2 = 0$ otherwise is not, because the notion of execution trace presuposes a notion of effectivity.

This notion of effectivity can be defined in the framework of small step operational semantics.

## 3.1 Small step operational semantics for functional languages

Small step operational semantics has first been defined for functional programming languages.

For instance, to define the result of the execution of the program `fun x → x * x + 1` on the value 7, we first build the term `(fun x → x * x + 1) 7` that contains both the program and the input value and then reduce it, step by step, to the irreducible term `50`, with rewrite rules defining the semantics of the language.

For instance, with the $\beta$-reduction rule and obvious rules for the addition and multiplication, we have the following reduction sequence

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

Again, not all sets of sequences are sets of reduction sequences. A set of sequences is a set of reduction sequences, if there exists a computable function, computing the next element of a sequence, from the previous one—and, in the non deterministic case, the finite set of possible next elements from the previous one.

**Definition 1 (Set of reduction sequences)** *A set $R$ of sequences is a set of reduction sequences if there exists a set $I$ of initial states and computable function step such that $R$ is the set of all sequences $s_1, ..., s_n$ such that $s_1$ is an element of $I$, $s_n$ is such that $step(s_n) = \varnothing$ and for all $i$, $s_{i+1} \in step(s_i)$.*

The computable function *step* defines the granularity of the reduction. For instance, a $\beta$-reduction step can be considered as atomic, or as a sequence of steps whose length depends on the size of the body of the abstraction, like in the calculus of explicit substitutions [1]. In the same way, the assignment of a variable can be considered as an atomic step or as a sequence of steps whose length depends, logarithmically, on the size of the memory.

This choice of a granularity affects the length of the reduction sequences, hence the complexity of the algorithms. The complexity of algorithms is thus relative to the choice of such as function *step*.

## 3.2 Small step operational semantics for imperative languages

An imperative program, such as

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

$$\langle \texttt{skip;r}, \rho \rangle \longrightarrow \langle \texttt{r}, \rho \rangle$$

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

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

If $[\![\texttt{t}]\!]_\rho = \texttt{true}$ then

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

If $[\![\texttt{t}]\!]_\rho = \texttt{false}$ then

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

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

Figure 1: Small step operational semantics of a simple imperative language

cannot, alone, be reduced by small step operational semantics rules. But, the ordered pair formed with this program and a state such as $\langle \texttt{x} = 12 \rangle$ can be reduced to the pair formed with the empty program and the state $\langle \texttt{x} = 3 \rangle$. The small step operational semantics of a simple imperative language is given in Figure 1, considering that `r` may be empty, in which case `p;r` is just `p`. Irreducible terms are those whose program is empty. For instance, the reduction sequence of this program in the state $\langle \texttt{x} = 12 \rangle$ is the sequence

$\langle$`while even(x) do x := x / 2`$, \langle \texttt{x} = 12 \rangle\rangle$
$\longrightarrow \langle$`if even(x) then x := x /2; while even(x) do x := x / 2 else skip`$, \langle \texttt{x} = 12 \rangle\rangle$
$\longrightarrow \langle$`x := x /2; while even(x) do x := x / 2`$, \langle \texttt{x} = 12 \rangle\rangle$
$\longrightarrow \langle$`while even(x) do x := x / 2`$, \langle \texttt{x} = 6 \rangle\rangle$
$\longrightarrow \langle$`if even(x) then x := x /2; while even(x) do x := x / 2 else skip`$, \langle \texttt{x} = 6 \rangle\rangle$
$\longrightarrow \langle$`x := x /2; while even(x) do x := x / 2`$, \langle \texttt{x} = 6 \rangle\rangle$
$\longrightarrow \langle$`while even(x) do x := x / 2`$, \langle \texttt{x} = 3 \rangle\rangle$
$\longrightarrow \langle$`if even(x) then x := x /2; while even(x) do x := x / 2 else skip`$, \langle \texttt{x} = 3 \rangle\rangle$
$\longrightarrow \langle$`skip`$, \langle \texttt{x} = 3 \rangle\rangle$
$\longrightarrow \langle, \langle \texttt{x} = 3 \rangle\rangle$

### 3.3 From reduction sequences to execution traces

The notion of execution trace of an imperative program can easily be defined from such a reduction sequence: if the reduction sequence is

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

then the execution trace of $\texttt{p}_1$ in $\rho_1$ is the sequence obtained by dropping the program in each pair and errasing variables names in the states.

For instance, the execution trace of the program

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

in the state $\langle \texttt{x} = 12 \rangle$ is

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

More generally, a execution trace is any projection of a reduction sequence. This rules out algorithms computing non computable functions and guarantees effectiveness.

**Definition 2 (Projection)** *Let proj be a computable function. If $s$ is a sequence $s_1, ..., s_n$ we write proj($s$) for the pointwise application of the function proj to the elements of this sequence: proj($s_1$), ..., proj($s_n$).*

*If $R$ is a set of sequences, we write $proj(R)$ for the pointwise application of the function proj to the elements of $R$.*

*We say that a set $A$ is a projection of a set $R$ if there exists a computable function proj such that $A = proj(R)$.*

## 3.4 Speeding up

The set of execution traces

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

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

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

is different from the set of execution traces of Section 2.1.

But, the sequence

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

is a subsequence of the sequence

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

and the number of consecutive omitted steps is bounded by 3.

This leads to define a notion of speed up of a set of sequences.

**Definition 3 (Speed up)** *A set of sequences $B$ is a speed up of a set of sequences $A$ if there exists a number $k$ such that each sequence of $B$ is a subsequence of one of $A$, containing the same first and last elements and such that the number of consecutive omitted elements is bounded by $k$.*

Effectiveness is preserved by a speed up, and complexity also as the length of execution traces is at most multiplied or divided by the constant $k$.

We can now give a formal definition of the notion of algorithm.

**Definition 4 (Algorithm)** *An algorithm is a set of sequences that is a speed up of a projection of a set of reduction sequences.*

For instance, the initial set containing all the pairs formed with the program

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

and a state $\langle \mathtt{x} = n\rangle$ for some natural number $n$, and the *step* function defined in Figure 1 define a set of reduction sequence. The projection function *proj* erasing the program and the variable names define an algorithm containing the traces

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

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

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

that has a a speed up the algorithm containing the traces

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

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

$$\langle 7\rangle$$

$$\ldots$$

So, both algorithms are expressed by the program

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

Thus, this program does not express a single algorithm, but several depending on the granularity, defined by the *step* function and by the speed up, at which this program is observed.

# 4 Functional languages

The very idea of functional programming is that programs do not do anything, but just compute something. So it is often assumed that the notion of execution trace—what the program does—makes no sense for functional languages.

The absence of a notion of execution trace for functional programs is probably at the root of the controversy [3] on the definition of the notion of algorithm as sets of execution traces [6] and as recursive equations [7].

But, as we have seen, the functional languages have a small step operational semantics, and, as we shall see, this small step operational semantics permits to extend the notion of execution trace to functional programs.

## 4.1 Recursive equations

Consider, for instance the functional program, that computes the largest odd factor of a natural number, defined by the recursive equation

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

or the rewrite rule

$$\texttt{f(x)} \longrightarrow \texttt{if(even(x),f(/(x,2)),x)}$$

with obvious rules to compute parity, division and test.

Then, the term `f(12)` reduces to the term `3`. But, in small step operational semantics, the reduction sequence cannot just be

$\texttt{f(12)}$
$\longrightarrow \texttt{if(even(12),f(/(12,2)),12)}$
$\longrightarrow \texttt{if(true,f(/(12,2)),12)}$
$\longrightarrow \texttt{f(/(12,2))}$
$\longrightarrow \texttt{f(6)}$
$\longrightarrow \texttt{if(even(6),f(/(6,2)),6)}$
$\longrightarrow \texttt{if(true,f(/(6,2)),6)}$
$\longrightarrow \texttt{f(/(6,2))}$
$\longrightarrow \texttt{f(3)}$
$\longrightarrow \texttt{if(even(3),f(/(3,2)),3)}$
$\longrightarrow \texttt{if(false,f(/(3,2)),3)}$
$\longrightarrow \texttt{3}$

because the fact that `f(12)` reduces to `if(even(12),f(/(12,2)),12)` is not part of the semantics of the language, and depends on the program. So, like for imperative languages, we need to introduce a pair formed with a program—a set of recursive equations—and a term, and if we write `p` for the set of recursive equations above, the reduction sequence is then

$\langle \texttt{p,f(12)} \rangle$
$\longrightarrow \langle \texttt{p,if(even(12),f(/(12,2)),12)} \rangle$
$\longrightarrow \langle \texttt{p,if(true,f(/(12,2)),12)} \rangle$
$\longrightarrow \langle \texttt{p,f(/(12,2))} \rangle$
$\longrightarrow \ldots$

Thus, the sequence of terms

$\texttt{f(12)}$
$\longrightarrow \texttt{if(even(12),f(/(12,2)),12)}$
$\longrightarrow \texttt{if(true,f(/(12,2)),12)}$
$\longrightarrow \texttt{f(/(12,2))}$
$\longrightarrow \ldots$

that contains program names, `f`, `/`, ... but no programs, is the analog of sequence of states in imperative languages.

In the definition of a trace for imperative programs we have dropped variables names. Exactly in the same way, we can drop program names and get the execution trace

```
<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,<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 Lambda-calculus style languages

When functional programs are defined with recursive equations or with rewrite rules, it is possible to separate the state from the program, and thus to define a notion of execution trace. But, in other functional languages, such as the lambda-calculus or the rho-calculus [4], the program and the data are less easy to separate.

We consider an extension of the lambda-calculus where the `fun` operation is recursive, called `fixfun` in [5], and the reduction rule

$$\texttt{(fun f x} \to \texttt{t) u} \longrightarrow \texttt{(u/x,fun f x} \to \texttt{t/f)t}$$

with obvious rules to compute parity, division and test.

Then, the term `(fun f x` $\to$ `(if (even x) (f (/ x 2)) x))` `12` reduces to the term `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
```

To separate the programs from the data, we replace every abstraction by the constant `<fun>` and get the execution trace

```
(<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 (<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
```

Note that, with minor notation changes, this execution trace is the same as that in the the language of Section 4.1.

# 5 When do two functional programs express the same algorithm?

This notion of execution trace permits to define algorithms as a set of execution traces and define when two functional programs execute the same algorithm. For this definition, although syntactically different, the two programs computing the identity function on Booleans

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

and

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

express the same algorithm

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


# Acknowledgements

Many thanks to the members of the working group Tarmac for enlightening discussions.


# References


[1] M. Abadi, L. Cardelli, P-L. Curien, and J-J. Levy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[2] A. Blass, N. Dershowitz, and Y. Gurevich. When are two algorithms the same? *Bullettin of Symbolic logic*, 12(2):145–168, 2009.

[3] A. Blass and Y. Gurevich. Algorithms vs. machines. *Bullettin of the European Association for Theoretical Computer Science*, 77:96–118, 2002.

[4] H. Cirstea and C. Kirchner. The rewriting calculus - part I. *Logic Journal of the IGPL*, 9(3):339–375, 2001.

[5] G. Dowek and J.-J. Lévy. *Introduction to the Theory of Programming Languages*. Springer, 2010.

[6] Y. Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 2000.

[7] Y.N. Moschovakis. What is an algorithm? *Mathematics unlimited*, pages 919–936, 2001.

[8] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.