Algorithms and Execution Traces

Declan Thompson
declan@stanford.edu

16 March 2023

Outline

1. The big picture
2. Challenges
3. Main contributions

Algorism The Euclidean algorithm

while $\texttt{rem}(y, x) \neq 0$ **do**

  $z \leftarrow x$;

  $x \leftarrow \texttt{rem}(y, x)$;

  $y \leftarrow z$;

end

return x

`rem(a,b)` is the remainder when `a` is divided by `b`.

An account of named algorithms

An account of named algorithms

1. An algorithm is the set of its own execution traces.

An account of named algorithms

1. An algorithm is the set of its own execution traces.
2. Algorithms are constructed out of basic operations.

An account of named algorithms

1. An algorithm is the set of its own execution traces.
2. Algorithms are constructed out of basic operations.
3. Algorithms can be described by finite texts (*algorisms*).

An algorithm is the set of its own execution traces

An algorithm is the set of its own execution traces

Algorism The Euclidean algorithm

while $\texttt{rem}(y, x) \neq 0$ **do**
    $z \leftarrow x$;
    $x \leftarrow \texttt{rem}(y, x)$;
    $y \leftarrow z$;
end
return x

rem(a,b) is the remainder when a is divided by b.

$$\langle x : 4, y : 6 \rangle$$

An algorithm is the set of its own execution traces

Algorism The Euclidean algorithm

```
while rem(y, x) ≠ 0 do
    z ← x;
    x ← rem(y, x);
    y ← z;
end
return x
```

rem(a,b) is the remainder when a is divided by b.

$\langle x : 4, y : 6 \rangle,$

$\langle x : 4, y : 6 \rangle$

An algorithm is the set of its own execution traces

Algorism The Euclidean algorithm

```
while rem(y, x) ≠ 0 do
    z ← x;
    x ← rem(y, x);
    y ← z;
end
return x
```

rem(a,b) is the remainder when a is divided by b.

$\langle x : 4, y : 6 \rangle,$

$\langle x : 4, y : 6 \rangle,$

$\langle x : 4, y : 6, z : 4 \rangle$

An algorithm is the set of its own execution traces

Algorism The Euclidean algorithm

```
while rem(y, x) ≠ 0 do
    z ← x;
    x ← rem(y, x);
    y ← z;
end
return x
```

rem(a,b) is the remainder when a is divided by b.

$\langle x : 4, y : 6 \rangle$,

$\langle x : 4, y : 6 \rangle$,

$\langle x : 4, y : 6, z : 4 \rangle$,

$\langle x : 2, y : 6, z : 4 \rangle$

An algorithm is the set of its own execution traces

Algorism The Euclidean algorithm

```
while rem(y, x) ≠ 0 do
    z ← x;
    x ← rem(y, x);
    y ← z;
end
return x
```

rem(a,b) is the remainder when a is divided by b.

$\langle x : 4, y : 6 \rangle$,

$\langle x : 4, y : 6 \rangle$,

$\langle x : 4, y : 6, z : 4 \rangle$,

$\langle x : 2, y : 6, z : 4 \rangle$,

$\langle x : 2, y : 4, z : 4 \rangle$

An algorithm is the set of its own execution traces

Algorism The Euclidean algorithm

```
while rem(y, x) ≠ 0 do
    z ← x;
    x ← rem(y, x);
    y ← z;
end
return x
```

rem(a,b) is the remainder when a is divided by b.

$\langle x : 4, y : 6\rangle,$

$\langle x : 4, y : 6\rangle,$

$\langle x : 4, y : 6, z : 4\rangle,$

$\langle x : 2, y : 6, z : 4\rangle,$

$\langle x : 2, y : 4, z : 4\rangle,$

$\langle x : 2, y : 4, z : 4\rangle$

An algorithm is the set of its own execution traces

Algorism The Euclidean algorithm

while $\texttt{rem}(y, x) \neq 0$ **do**
    $z \leftarrow x$;
    $x \leftarrow \texttt{rem}(y, x)$;
    $y \leftarrow z$;
end
return x

`rem(a,b)` is the remainder when `a` is divided by `b`.

$\langle x : 4, y : 6 \rangle,$

$\langle x : 4, y : 6 \rangle,$

$\langle x : 4, y : 6, z : 4 \rangle,$

$\langle x : 2, y : 6, z : 4 \rangle,$

$\langle x : 2, y : 4, z : 4 \rangle,$

$\langle x : 2, y : 4, z : 4 \rangle,$

$\langle x : 2 \rangle$

An algorithm is the set of its own execution traces

Algorism The Euclidean algorithm

while $\mathtt{rem}(y, x) \neq 0$ **do**
  $z \leftarrow x$;
  $x \leftarrow \mathtt{rem}(y, x)$;
  $y \leftarrow z$;
end
return x

rem(a,b) is the remainder when a is divided by b.

$\langle x : 4, y : 6\rangle$,
$\langle x : 4, y : 6\rangle$,
$\langle x : 4, y : 6, z : 4\rangle$,
$\langle x : 2, y : 6, z : 4\rangle$,
$\langle x : 2, y : 4, z : 4\rangle$,
$\langle x : 2, y : 4, z : 4\rangle$,
$\langle x : 2\rangle$

Key idea

Model algorithms as sets of sequences over appropriate universes of objects.

Algorithms are constructed out of basic operations

Algorithms are constructed out of basic operations

- Each algorithm is associated with a *purpose* and appears in a *context*.

Algorithms are constructed out of basic operations

- Each algorithm is associated with a *purpose* and appears in a *context*.
- A purpose in context provides a set of *basic operations*.

Algorithms are constructed out of basic operations

- Each algorithm is associated with a *purpose* and appears in a *context*.
- A purpose in context provides a set of *basic operations*.
- An algorithm for a given purpose, in a given context, is *constructed* out of basic operations.

Algorithms are constructed out of basic operations

- Each algorithm is associated with a *purpose* and appears in a *context*.
- A purpose in context provides a set of *basic operations*.
- An algorithm for a given purpose, in a given context, is *constructed* out of basic operations.

$$\overbrace{\langle x:4, y:6\rangle, \mathrlap{\underbrace{}_{z\leftarrow x}}\langle x:4, y:6\rangle}^{\texttt{rem}(y,x)\neq 0}, \overbrace{\langle x:4, y:6, z:4\rangle, \mathrlap{\underbrace{}_{y\leftarrow z}}\langle x:2, y:6, z:4\rangle}^{x\leftarrow\texttt{rem}(y,x)}, \overbrace{\langle x:2, y:4, z:4}^{\texttt{rem}(}$$

Algorithms are constructed out of basic operations

- Each algorithm is associated with a *purpose* and appears in a *context*.
- A purpose in context provides a set of *basic operations*.
- An algorithm for a given purpose, in a given context, is *constructed* out of basic operations.

$$\langle x:4, y:6\rangle \overset{\texttt{rem}(y,x)\neq 0}{,} \langle x:4, y:6\rangle \underset{z\leftarrow x}{,} \langle x:4, y:6, z:4\rangle \overset{x\leftarrow\texttt{rem}(y,x)}{,} \langle x:2, y:6, z:4\rangle \underset{y\leftarrow z}{,} \langle x:2, y:4, z:4 \quad \overset{\texttt{rem}(}{}$$

Question

What does it mean for an algorithm to be constructed from basic operations?

Control Equivalence

Control Equivalence

Consider *trace points*: partial execution traces.

Control Equivalence

Consider *trace points*: partial execution traces.

- Length 1 trace points are inputs.
- The same set of operations is applied to all inputs.

Control Equivalence

Consider *trace points*: partial execution traces.

- Length 1 trace points are inputs.
- The same set of operations is applied to all inputs.

$$\underbrace{\langle x:4, y:6\rangle, \langle x:4, y:6\rangle}_{\texttt{rem}(y,x)\neq 0}, \ldots$$

$$\overbrace{\langle x:2, y:8\rangle, \langle x:2, y:8\rangle}^{\texttt{rem}(y,x)=0}, \ldots$$

Control Equivalence

Consider *trace points*: partial execution traces.

- Length 1 trace points are inputs.
- The same set of operations is applied to all inputs.

$$\underbrace{\langle x:4, y:6\rangle, \langle x:4, y:6\rangle}_{\texttt{rem}(y,x)\neq 0}, \ldots$$

$$\overbrace{\langle x:2, y:8\rangle, \langle x:2, y:8\rangle}^{\texttt{rem}(y,x)=0}, \ldots$$

- If σ and τ haven't yet been distinguished, then the same set of operations is applied to each.

Control Equivalence

Consider *trace points*: partial execution traces.

- Length 1 trace points are inputs.
- The same set of operations is applied to all inputs.

$$\underbrace{\langle x:4, y:6\rangle, \langle x:4, y:6\rangle}_{\texttt{rem}(y,x)\neq 0}, \ldots$$

$$\overbrace{\langle x:2, y:8\rangle, \langle x:2, y:8\rangle}^{\texttt{rem}(y,x)=0}, \ldots$$

- If σ and τ haven't yet been distinguished, then the same set of operations is applied to each.

$$\{\text{extensions of } \sigma, \tau\} = \{\sigma, \tau\} \circ \mathrm{P}$$

Control Equivalence

Consider *trace points*: partial execution traces.

- Length 1 trace points are inputs.
- The same set of operations is applied to all inputs.

$$\underbrace{\langle x:4, y:6\rangle, \langle x:4, y:6\rangle}_{\texttt{rem}(y,x)\neq 0}, \ldots$$

$$\overbrace{\langle x:2, y:8\rangle, \langle x:2, y:8\rangle}^{\texttt{rem}(y,x)=0}, \ldots$$

- If σ and τ haven't yet been distinguished, then the same set of operations is applied to each.

$$\{\text{extensions of } \sigma, \tau\} = \{\sigma, \tau\} \circ \mathrm{P}$$

Define a *control equivalence* relation on the trace points of an algorithm:

$$\sigma \leftrightarrows \tau \qquad \text{iff} \qquad \text{“the same thing happens next” in } \sigma \text{ and } \tau$$

A control equivalence describes how an algorithm is constructed from operations given by a purpose in context.

Algorithms can be described by finite texts

- Algorithms are describable by finite texts.

Algorithms can be described by finite texts

- Algorithms are describable by finite texts.
- At the same time, algorithms are language independent.

Algorithms can be described by finite texts

- Algorithms are describable by finite texts.
- At the same time, algorithms are language independent.

A *finite control equivalence* is a control equivalence with a finite number of equivalence classes, each using a finite stock of operations.

Algorithms can be described by finite texts

- Algorithms are describable by finite texts.
- At the same time, algorithms are language independent.

A *finite control equivalence* is a control equivalence with a finite number of equivalence classes, each using a finite stock of operations.

Thesis

An algorithm is a set of execution traces equipped with a finite control equivalence.

Outline

Two perspectives on algorithms (at least)

Two perspectives on algorithms (at least)

An orthodox definition

An algorithm is a finite sequence of instructions that:

- is unambiguous;
- is deterministic;
- uses fixed finitary operations;
- is guaranteed to solve its task;
- takes a finite amount of time.

Two perspectives on algorithms (at least)

An orthodox definition

An algorithm is a finite sequence of instructions that:

- is unambiguous;
- is deterministic;
- uses fixed finitary operations;
- is guaranteed to solve its task;
- takes a finite amount of time.

- Standard accounts of algorithms start from computability theory.

Two perspectives on algorithms (at least)

An orthodox definition

An algorithm is a finite sequence of instructions that:

- is unambiguous;
- is deterministic;
- uses fixed finitary operations;
- is guaranteed to solve its task;
- takes a finite amount of time.

Consequences of my thesis

An algorithm is a non-syntactic object that:

- may be under-determined;
- need not be deterministic;
- may use arbitrary operations;
- may be incorrect;
- could never halt (or more!).

- Standard accounts of algorithms start from computability theory.

Two perspectives on algorithms (at least)

An orthodox definition

An algorithm is a finite sequence of instructions that:

- is unambiguous;
- is deterministic;
- uses fixed finitary operations;
- is guaranteed to solve its task;
- takes a finite amount of time.

Consequences of my thesis

An algorithm is a non-syntactic object that:

- may be under-determined;
- need not be deterministic;
- may use arbitrary operations;
- may be incorrect;
- could never halt (or more!).

- Standard accounts of algorithms start from computability theory.
- Examples abound of algorithms which violate the standard conditions.

Example argument: executive finiteness

If σ is an algorithm's execution trace then $|\sigma| < \omega$

Example argument: executive finiteness

If σ is an algorithm's execution trace then $|\sigma| < \omega$

Enumeration algorithms allow infinite traces

Example argument: executive finiteness

If σ is an algorithm's execution trace then $|\sigma| < \omega$

The number of steps between inputs/outputs in algorithm's execution traces is always finite

Enumeration algorithms allow infinite traces

Example argument: executive finiteness

If σ is an algorithm's execution trace then $|\sigma| < \omega$

Enumeration algorithms allow infinite traces

The number of steps between inputs/outputs in algorithm's execution traces is always finite

Any sufficiently powerful model of computation contains instances entering infinite loops

Example argument: executive finiteness

If σ is an algorithm's execution trace then $|\sigma| < \omega$

Enumeration algorithms allow infinite traces

The number of steps between inputs/outputs in algorithm's execution traces is always finite

Any sufficiently powerful model of computation contains instances entering infinite loops

(1) Those aren't algorithms!

Example argument: executive finiteness

If σ is an algorithm's execution trace then $|\sigma| < \omega$

Enumeration algorithms allow infinite traces

The number of steps between inputs/outputs in algorithm's execution traces is always finite

Any sufficiently powerful model of computation contains instances entering infinite loops

(1) Those aren't algorithms!

This is a terminological dispute

Example argument: executive finiteness

If σ is an algorithm's execution trace then $|\sigma| < \omega$

Enumeration algorithms allow infinite traces

The number of steps between inputs/outputs in algorithm's execution traces is always finite

Any sufficiently powerful model of computation contains instances entering infinite loops

(1) Those aren't algorithms!

This is a terminological dispute

(2) If σ is an algorithm's execution trace then $|\sigma| \leq \omega$

Example argument: executive finiteness

If σ is an algorithm's execution trace then $|\sigma| < \omega$

Enumeration algorithms allow infinite traces

The number of steps between inputs/outputs in algorithm's execution traces is always finite

Any sufficiently powerful model of computation contains instances entering infinite loops

(1) Those aren't algorithms!

This is a terminological dispute

(2) If σ is an algorithm's execution trace then $|\sigma| \leq \omega$

Transfinitary algorithms appear in some theoretical contexts

Example argument: executive finiteness

If σ is an algorithm's execution trace then $|\sigma| < \omega$

Enumeration algorithms allow infinite traces

The number of steps between inputs/outputs in algorithm's execution traces is always finite

Any sufficiently powerful model of computation contains instances entering infinite loops

(1) Those aren't algorithms!

This is a terminological dispute

(2) If σ is an algorithm's execution trace then $|\sigma| \leq \omega$

Transfinitary algorithms appear in some theoretical contexts

Allowing algorithms with traces of arbitrary length gives a more general theory and avoids unnecessary overheads.

Example argument: executive finiteness

If σ is an algorithm's execution trace then $|\sigma| < \omega$

Enumeration algorithms allow infinite traces

The number of steps between inputs/outputs in algorithm's execution traces is always finite

Any sufficiently powerful model of computation contains instances entering infinite loops

(1) Those aren't algorithms!

This is a terminological dispute

(2) If σ is an algorithm's execution trace then $|\sigma| \leq \omega$

Transfinitary algorithms appear in some theoretical contexts

Allowing algorithms with traces of arbitrary length gives a more general theory and avoids unnecessary overheads.

Principle Algorithms can have execution traces of any ordinal length.

Example argument: executive finiteness

If σ is an algorithm's execution trace then $\lvert\sigma\rvert < \omega$	*Enumeration algorithms allow infinite traces*
The number of steps between inputs/outputs in algorithm's execution traces is always finite	*Any sufficiently powerful model of computation contains instances entering infinite loops*
(1) Those aren't algorithms!	*This is a terminological dispute*
(2) If σ is an algorithm's execution trace then $\lvert\sigma\rvert \leq \omega$	*Transfinitary algorithms appear in some theoretical contexts*

Allowing algorithms with traces of arbitrary length gives a more general theory and avoids unnecessary overheads.

Principle Algorithms can have execution traces of any ordinal length.

Desideratum An explanation of when executive finiteness *does* hold.

Outline

Main contributions

Main contributions

1. A clear distinction between the two perspectives on algorithms:

2. A careful analysis of the concept of an algorithm from the distinct methods perspective:

3. A formal definition of finite control trace sets for a restricted class of algorithms, with associated results:

Main contributions

1. A clear distinction between the two perspectives on algorithms:
   - The sense of *algorithm* familiar from computability theory (the *effective procedures perspective*)
   - The sense of *algorithm* used when considering *named* algorithms (the *distinct methods perspective*)
2. A careful analysis of the concept of an algorithm from the distinct methods perspective:
3. A formal definition of finite control trace sets for a restricted class of algorithms, with associated results:

Main contributions

1. A clear distinction between the two perspectives on algorithms:
   - The sense of *algorithm* familiar from computability theory (the *effective procedures perspective*)
   - The sense of *algorithm* used when considering *named* algorithms (the *distinct methods perspective*)
2. A careful analysis of the concept of an algorithm from the distinct methods perspective:
   - A trace-based framework within which the analysis proceeds
3. A formal definition of finite control trace sets for a restricted class of algorithms, with associated results:

Main contributions

1. A clear distinction between the two perspectives on algorithms:
   - The sense of *algorithm* familiar from computability theory (the *effective procedures perspective*)
   - The sense of *algorithm* used when considering *named* algorithms (the *distinct methods perspective*)

2. A careful analysis of the concept of an algorithm from the distinct methods perspective:
   - A trace-based framework within which the analysis proceeds
   - A set of *principles* that algorithms satisfy
   - A set of *desiderata* for any account of algorithms to meet

3. A formal definition of finite control trace sets for a restricted class of algorithms, with associated results:

Main contributions

1. A clear distinction between the two perspectives on algorithms:
   - The sense of *algorithm* familiar from computability theory (the *effective procedures perspective*)
   - The sense of *algorithm* used when considering *named* algorithms (the *distinct methods perspective*)
2. A careful analysis of the concept of an algorithm from the distinct methods perspective:
   - A trace-based framework within which the analysis proceeds
   - A set of *principles* that algorithms satisfy
   - A set of *desiderata* for any account of algorithms to meet
3. A formal definition of finite control trace sets for a restricted class of algorithms, with associated results:
   - Finite control trace sets meet the desiderata and satisfy the principles

Main contributions

1. A clear distinction between the two perspectives on algorithms:
   - The sense of *algorithm* familiar from computability theory (the *effective procedures perspective*)
   - The sense of *algorithm* used when considering *named* algorithms (the *distinct methods perspective*)
2. A careful analysis of the concept of an algorithm from the distinct methods perspective:
   - A trace-based framework within which the analysis proceeds
   - A set of *principles* that algorithms satisfy
   - A set of *desiderata* for any account of algorithms to meet
3. A formal definition of finite control trace sets for a restricted class of algorithms, with associated results:
   - Finite control trace sets meet the desiderata and satisfy the principles
   - Every finite control trace set can be specified by a finite text

Main contributions

1. A clear distinction between the two perspectives on algorithms:
   - The sense of *algorithm* familiar from computability theory (the *effective procedures perspective*)
   - The sense of *algorithm* used when considering *named* algorithms (the *distinct methods perspective*)
2. A careful analysis of the concept of an algorithm from the distinct methods perspective:
   - A trace-based framework within which the analysis proceeds
   - A set of *principles* that algorithms satisfy
   - A set of *desiderata* for any account of algorithms to meet
3. A formal definition of finite control trace sets for a restricted class of algorithms, with associated results:
   - Finite control trace sets meet the desiderata and satisfy the principles
   - Every finite control trace set can be specified by a finite text
   - The recovery of computability theory using finite control trace sets

Thank you