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

1. while $\text{rem}(y, x) \neq 0$ do
2.     $z \leftarrow x$;
3.     $x \leftarrow \text{rem}(y, x)$;
4.     $y \leftarrow z$;
5. end
6. return $x$

$\text{rem}(a, b)$ is the remainder when $a$ is divided by $b$. 
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 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

1. while $\text{rem}(y, x) \neq 0$ do
2.     $z \leftarrow x$;
3.     $x \leftarrow \text{rem}(y, x)$;
4.     $y \leftarrow z$;
5. end
6. return $x$

$\text{rem}(a, b)$ 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

1. while $\text{rem}(y, x) \neq 0$ do
2.     $z \leftarrow x$;
3.     $x \leftarrow \text{rem}(y, x)$;
4.     $y \leftarrow z$;
5. end
6. return $x$

$\text{rem}(a, b)$ 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

1. while \( \text{rem}(y, x) \neq 0 \) do
2. \[ z \leftarrow x; \]
3. \[ x \leftarrow \text{rem}(y, x); \]
4. \[ y \leftarrow z; \]
5. end
6. return \( x \)

\( \text{rem}(a, b) \) is the remainder when \( a \) is divided by \( b \).
An algorithm is the set of its own execution traces

**Algorism** The Euclidean algorithm

1. while \( \text{rem}(y, x) \neq 0 \) do
2. \( z \leftarrow x; \)
3. \( x \leftarrow \text{rem}(y, x); \)
4. \( y \leftarrow z; \)
5. end
6. return \( x \)

\( \text{rem}(a, b) \) is the remainder when \( a \) is divided by \( b \).
An algorithm is the set of its own execution traces

**Algorism** The Euclidean algorithm

1. while \( \text{rem}(y, x) \neq 0 \) do
2. \[ z \leftarrow x; \]
3. \[ x \leftarrow \text{rem}(y, x); \]
4. \[ y \leftarrow z; \]
5. end
6. return \( x \)

\( \text{rem}(a, b) \) is the remainder when \( a \) is divided by \( b \).
An algorithm is the set of its own execution traces

**Algorism** The Euclidean algorithm

1. while \( \text{rem}(y, x) \neq 0 \) do
2.     \( z \leftarrow x; \)
3.     \( x \leftarrow \text{rem}(y, x); \)
4.     \( y \leftarrow z; \)
5. end
6. return \( x \)

\( \text{rem}(a, b) \) is the remainder when \( a \) is divided by \( b \).
An algorithm is the set of its own execution traces

**Algorism** The Euclidean algorithm

1. while rem(y, x) ≠ 0 do
2.     z ← x;
3.     x ← rem(y, x);
4.     y ← z;
5. end
6. return x

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

⟨x: 4, y: 6⟩,
⟨x: 4, y: 6⟩,
⟨x: 4, y: 6, z: 4⟩,
⟨x: 2, y: 6, z: 4⟩,
⟨x: 2, y: 4, z: 4⟩,
⟨x: 2, y: 4, z: 4⟩,
⟨x: 2⟩
An algorithm is the set of its own execution traces

**Algorism** The Euclidean algorithm

1. while \( \text{rem}(y, x) \neq 0 \) do
2. \( z \leftarrow x; \)
3. \( x \leftarrow \text{rem}(y, x); \)
4. \( y \leftarrow z; \)
5. end
6. return \( x \)

\( \text{rem}(a, b) \) 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

What does it mean for an algorithm to be constructed from 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.

\[
\begin{align*}
\text{rem}(y, x) &\neq 0 \\
\langle x : 4, y : 6 \rangle, \langle x : 4, y : 6 \rangle, \langle x : 4, y : 6, z : 4 \rangle, \langle x : 2, y : 6, z : 4 \rangle, \langle x : 2, y : 4, z : 4 \rangle \\
\end{align*}
\]

\[
\begin{align*}
z &\leftarrow x \\
x &\leftarrow \text{rem}(y, x) \\
y &\leftarrow z \\
\end{align*}
\]
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.

\[
\begin{align*}
\text{rem}(y,x) \neq 0 \\
\{ \langle x : 4, y : 6 \rangle, \langle x : 4, y : 6 \rangle, \langle x : 4, y : 6, z : 4 \rangle, \langle x : 2, y : 6, z : 4 \rangle, \langle x : 2, y : 4, z : 4 \rangle \} \\
\text{rem}(y,x) \\
\{ \langle x : 2, y : 4, z : 4 \rangle \}
\end{align*}
\]

**Question**

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

Consider trace points: partial execution traces. Length 1 trace points are inputs. The same set of operations is applied to all inputs.

\[
\{ z \} \text{rem}(y, x) \neq 0
\]

\[
\langle x: 4, y: 6 \rangle, \langle x: 4, y: 6 \rangle, \ldots
\]

\[
\text{rem}(y, x) = 0
\]

\[
\{ z \} \text{rem}(y, x) = 0
\]

\[
\langle x: 2, y: 8 \rangle, \langle x: 2, y: 8 \rangle, \ldots
\]

If \( \sigma \) and \( \tau \) haven't yet been distinguished, then the same set of operations is applied to each.

\[
\{ \text{extensions of } \sigma, \tau \} = \{ \sigma, \tau \} \circ p
\]

Define a control equivalence relation on the trace points of an algorithm: \( \sigma \leftrightarrow \tau \) iff "the same thing happens next" in \( \sigma \) and \( \tau \). A control equivalence describes how an algorithm is constructed from operations given by a purpose in context.
Control Equivalence

Consider *trace points*: partial execution traces.

\[
\{ z \mid \text{rem}(y, x) \neq 0 \} \ni \langle x: 4, y: 6 \rangle, \langle x: 4, y: 6 \rangle, \ldots
\]

If \( \sigma \) and \( \tau \) haven't yet been distinguished, then the same set of operations is applied to each.

\[
\{ \text{extensions of } \sigma, \tau \} = \{ \sigma, \tau \} \circ p
\]

Define a control equivalence relation on the trace points of an algorithm: \( \sigma \leftrightarrow \tau \) iff "the same thing happens next" in \( \sigma \) and \( \tau \).
Consider *trace points*: partial execution traces.

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

\[
\{ z \} \rem (y, x) \neq 0
\langle x: 4, y: 6 \rangle, \langle x: 4, y: 6 \rangle, \ldots
\]
\[
\{ \langle x: 2, y: 8 \rangle, \langle x: 2, y: 8 \rangle, \ldots \}
\]

If \( \sigma \) and \( \tau \) haven't yet been distinguished, then the same set of operations is applied to each.

\[
\{ \text{extensions of } \sigma, \tau \} = \{ \sigma, \tau \} \circ p
\]

Define a control equivalence relation on the trace points of an algorithm: \( \sigma \leftrightarrow \tau \) iff "the same thing happens next" in \( \sigma \) and \( \tau \).
Control Equivalence

Consider *trace points*: partial execution traces.

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

\[
\{ (x: 4, y: 6), (x: 4, y: 6), \ldots \} \\
\{ (x: 2, y: 8), (x: 2, y: 8), \ldots \}
\]

\[
\text{rem}(y, x) \neq 0 \\
\text{rem}(y, x) = 0
\]

If \( \sigma \) and \( \tau \) haven't yet been distinguished, then the same set of operations is applied to each.

\[
\{ \text{extensions of } \sigma, \tau \} = \{ \sigma, \tau \} \circ p
\]

Define a control equivalence relation on the trace points of an algorithm: \( \sigma \leftrightarrow \tau \) iff "the same thing happens next" in \( \sigma \) and \( \tau \).
Control Equivalence

Consider *trace points*: partial execution traces.

- Length 1 trace points are inputs.
- The same set of operations is applied to all inputs.
- If $\sigma$ and $\tau$ haven’t yet been distinguished, then the same set of operations is applied to each.

\[
\langle x : 4, y : 6 \rangle, \langle x : 4, y : 6 \rangle, \ldots
\]

\[
\text{if } \text{rem}(y, x) \neq 0
\]

\[
\langle x : 2, y : 8 \rangle, \langle x : 2, y : 8 \rangle, \ldots
\]

\[
\text{if } \text{rem}(y, x) = 0
\]
Control Equivalence

Consider *trace points*: partial execution traces.

- Length 1 trace points are inputs.
- The same set of operations is applied to all inputs.
- If \( \sigma \) and \( \tau \) haven't yet been distinguished, then the same set of operations is applied to each.

\[
\begin{cases}
\langle x : 4, y : 6 \rangle, \langle x : 4, y : 6 \rangle, \ldots \\
\langle x : 2, y : 8 \rangle, \langle x : 2, y : 8 \rangle, \ldots \\
\end{cases}
\]

\[
\text{If } \text{rem}(y,x) \neq 0 \text{, then } \langle x : 4, y : 6 \rangle, \langle x : 4, y : 6 \rangle, \ldots
\]

\[
\text{If } \text{rem}(y,x) = 0 \text{, then } \langle x : 2, y : 8 \rangle, \langle x : 2, y : 8 \rangle, \ldots
\]

\[
\{\text{extensions of } \sigma, \tau\} = \{\sigma, \tau\} \circ P
\]
Control Equivalence

Consider *trace points*: partial execution traces.

- Length 1 trace points are inputs.
- The same set of operations is applied to all inputs.
- If $\sigma$ and $\tau$ haven’t yet been distinguished, then the same set of operations is applied to each.

\[
\begin{align*}
\langle x : 4, y : 6 \rangle, \langle x : 4, y : 6 \rangle, \ldots \\
\langle x : 2, y : 8 \rangle, \langle x : 2, y : 8 \rangle, \ldots
\end{align*}
\]

If $\sigma$ and $\tau$ haven’t yet been distinguished, then the same set of operations is applied to each.

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

\[
\sigma \leftrightarrow \tau \quad \text{iff} \quad \text{“the same thing happens next” in } \sigma \text{ and } \tau
\]

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

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

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!).
### Two perspectives on algorithms (at least)

<table>
<thead>
<tr>
<th>An orthodox definition</th>
<th>Consequences of my thesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>An algorithm is a finite sequence of instructions that:</td>
<td></td>
</tr>
<tr>
<td>■ is unambiguous;</td>
<td></td>
</tr>
<tr>
<td>■ is deterministic;</td>
<td></td>
</tr>
<tr>
<td>■ uses fixed finitary operations;</td>
<td></td>
</tr>
<tr>
<td>■ is guaranteed to solve its task;</td>
<td></td>
</tr>
<tr>
<td>■ takes a finite amount of time.</td>
<td></td>
</tr>
</tbody>
</table>

| 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)

<table>
<thead>
<tr>
<th>An orthodox definition</th>
<th>Consequences of my thesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>An algorithm is a finite sequence of instructions that:</td>
<td>An algorithm is a non-syntactic object that:</td>
</tr>
<tr>
<td>■ is unambiguous;</td>
<td>■ may be under-determined;</td>
</tr>
<tr>
<td>■ is deterministic;</td>
<td>■ need not be deterministic;</td>
</tr>
<tr>
<td>■ uses fixed finitary operations;</td>
<td>■ may use arbitrary operations;</td>
</tr>
<tr>
<td>■ is guaranteed to solve its task;</td>
<td>■ may be incorrect;</td>
</tr>
<tr>
<td>■ takes a finite amount of time.</td>
<td>■ could never halt (or more!).</td>
</tr>
</tbody>
</table>

- Standard accounts of algorithms start from computability theory.
- Examples abound of algorithms which violate the standard conditions.
Example argument: executive finiteness

If $\sigma$ is an algorithm's execution trace then $|\sigma| < \omega$
Example argument: executive finiteness

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

Enumeration algorithms allow infinite traces

Those aren’t algorithms!

This is a terminological dispute

Any sufficiently powerful model of computation contains instances entering infinite loops

(1) Those aren’t algorithms!

(2) If $\sigma$ 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.

Desideratum

An explanation of when executive finiteness does hold.
Example argument: executive finiteness

If $\sigma$ 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*

---

*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.
Example argument: executive finiteness

If $\sigma$ 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*

*Any sufficiently powerful model of computation contains instances entering infinite loops*
Example argument: executive finiteness

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

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

(1) Those aren’t algorithms!

*Enumeration algorithms allow infinite traces*

*Any sufficiently powerful model of computation contains instances entering infinite loops*
Example argument: executive finiteness

If \( \sigma \) is an algorithm’s execution trace then \( |\sigma| < \omega \)

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

(1) Those aren’t algorithms!

Enumera\(\text{tion algorithms allow infinite traces}

Any sufficiently powerful model of computation contains instances entering infinite loops

This is a terminological dispute
Example argument: executive finiteness

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

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

(1) Those aren’t algorithms!

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

Enumeration algorithms allow infinite traces

Any sufficiently powerful model of computation contains instances entering infinite loops

This is a terminological dispute
If $\sigma$ is an algorithm's execution trace then $|\sigma| < \omega$

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

(1) Those aren't algorithms!

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

**Enumeration algorithms allow infinite traces**

Any sufficiently powerful model of computation contains instances entering infinite loops

This is a terminological dispute

Transfinite algorithms appear in some theoretical contexts
Example argument: executive finiteness

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

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

(1) Those aren’t algorithms!

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

Enumeration algorithms allow infinite traces

Any sufficiently powerful model of computation contains instances entering infinite loops

This is a terminological dispute

Transfinite algorithms appear in some theoretical contexts

Allowing algorithms with traces of arbitrary length gives a more general theory and avoids unnecessary overheads.
If $\sigma$ is an algorithm’s execution trace then $|\sigma| < \omega$

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

(1) Those aren’t algorithms!

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

Enumeration algorithms allow infinite traces

Any sufficiently powerful model of computation contains instances entering infinite loops

This is a terminological dispute

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 $\sigma$ is an algorithm’s execution trace then $|\sigma| < \omega$

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

(1) Those aren’t algorithms!

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

Enumeration algorithms allow infinite traces

Any sufficiently powerful model of computation contains instances entering infinite loops

This is a terminological dispute

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

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

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 \textit{algorithm} familiar from computability theory (the \textit{effective procedures perspective})
   - The sense of \textit{algorithm} used when considering \textit{named} algorithms (the \textit{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 \textit{principles} that algorithms satisfy
   - A set of \textit{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