# Enumerability, effectivity, decidability Markov algorithms

András Máté

10.10.2025

# The calculus $\mathbf{H}_3$

# The calculus $\mathbf{H}_3$

$\mathbf{H}_2$ (over an alphabet $\mathcal{A}_{cc}$ plus 9 auxiliary letters) derives strings with the intended meanings "$a$ is a calculus", "$b$ is a string of the alphabet of $a$", "$a$ derives $b$". ($a$ and $b$ are *translations, codes* or if you want, *names* of a calculus resp. word in $\mathcal{A}_{cc}$.)

# The calculus $\mathbf{H}_3$

$\mathbf{H}_2$ (over an alphabet $\mathcal{A}_{cc}$ plus 9 auxiliary letters) derives strings with the intended meanings "$a$ is a calculus", "$b$ is a string of the alphabet of $a$", "$a$ derives $b$". ($a$ and $b$ are *translations, codes* or if you want, *names* of a calculus resp. word in $\mathcal{A}_{cc}$.)

The calculus $\mathbf{H}_3$ is an extension of $\mathbf{H}_2$. It renders numerals to every $\mathcal{A}_{cc}$-string. (This is in effect a Gödel numbering.)
Numerals: strings consisting of $\alpha$-s only.

# The calculus $\mathbf{H}_3$

$\mathbf{H}_2$ (over an alphabet $\mathcal{A}_{cc}$ plus 9 auxiliary letters) derives strings with the intended meanings "$a$ is a calculus", "$b$ is a string of the alphabet of $a$", "$a$ derives $b$". ($a$ and $b$ are *translations, codes* or if you want, *names* of a calculus resp. word in $\mathcal{A}_{cc}$.)

The calculus $\mathbf{H}_3$ is an extension of $\mathbf{H}_2$. It renders numerals to every $\mathcal{A}_{cc}$-string. (This is in effect a Gödel numbering.)
Numerals: strings consisting of $\alpha$-s only.

First step: introduce a lexicographic ordering of $\mathcal{A}_{cc}$-strings.
New auxiliary letter: $F$ for the relation 'follows'.
I. e., $xFy$ should mean that the string $y$ follows $x$ in the lexicographic ordering.
Base: $\alpha$ follows the empty word.
Inductive rules define the follower of a string according to its last letter.

# Lexicographic ordering

# Lexicographic ordering

26. $F\alpha$
27. $x\alpha Fx\beta$
28. $x\beta Fx\xi$
29. $x\xi Fx \gg$
30. $x \gg Fx*$
31. $xFy \rightarrow x * Fy\alpha$

# Lexicographic ordering

26. $F\alpha$
27. $x\alpha Fx\beta$
28. $x\beta Fx\xi$
29. $x\xi Fx \gg$
30. $x \gg Fx*$
31. $xFy \rightarrow x * Fy\alpha$

From the language radix axioms it follows that:
Every $\mathcal{A}_{cc}$-string has one and only one follower;
Except of the empty string, each string is the follower of one and only one string.
The empty string is not a follower of anything.
I. e., strings with the empty string as 0 and this follower-relation as the successor-function fulfil axioms of primitive Peano arithmetics without mathematical induction.

# Gödel numbering of $\mathcal{A}_{cc}$-strings

# Gödel numbering of $\mathcal{A}_{cc}$-strings

Now we can add the (Gödel-)numbering to our calculus on the trivial way.
$G$ is a a new auxiliary letter, intended meaning of $xGy$: $y$ is the ordinal number of $x$ in the lexicographic ordering.

# Gödel numbering of $\mathcal{A}_{cc}$-strings

Now we can add the (Gödel-)numbering to our calculus on the trivial way.
$G$ is a a new auxiliary letter, intended meaning of $xGy$: $y$ is the ordinal number of $x$ in the lexicographic ordering.
Basis: the ordinal number of the empty string is the empty string itself.
Inductive rule: to get the number of the follower of a string $x$ we need to add an $\alpha$ to the number of $x$.

# Gödel numbering of $\mathcal{A}_{cc}$-strings

Now we can add the (Gödel-)numbering to our calculus on the trivial way.
$G$ is a a new auxiliary letter, intended meaning of $xGy$: $y$ is the ordinal number of $x$ in the lexicographic ordering.
Basis: the ordinal number of the empty string is the empty string itself.
Inductive rule: to get the number of the follower of a string $x$ we need to add an $\alpha$ to the number of $x$.

32. $G$
33. $xFy \rightarrow xGz \rightarrow yGz\alpha$

# Gödel numbering of $\mathcal{A}_{cc}$-strings

Now we can add the (Gödel-)numbering to our calculus on the trivial way.
$G$ is a a new auxiliary letter, intended meaning of $xGy$: $y$ is the ordinal number of $x$ in the lexicographic ordering.
Basis: the ordinal number of the empty string is the empty string itself.
Inductive rule: to get the number of the follower of a string $x$ we need to add an $\alpha$ to the number of $x$.

32. $G$
33. $xFy \rightarrow xGz \rightarrow yGz\alpha$

Our hypercalculus $\mathbf{H}_3$ now consists of the rules 1-33. and it suffices to prove at least one important incompleteness result.

# Autonomous numerals

# Autonomous numerals

Be $\mathbf{C}$ an arbitrary calculus.
The translation of $\mathbf{C}$ into our language is some $\mathcal{A}_{cc}$-word $a$.
$\mathbf{H}_3$ derives $Ka$.
There is a numeral $c$ s.t. $\mathbf{H}_3$ derives $aGc$, i. e. the Gödel number of $\mathbf{C}$ is $c$.

# Autonomous numerals

Be $\mathbf{C}$ an arbitrary calculus.
The translation of $\mathbf{C}$ into our language is some $\mathcal{A}_{cc}$-word $a$.
$\mathbf{H}_3$ derives $Ka$.
There is a numeral $c$ s.t. $\mathbf{H}_3$ derives $aGc$, i. e. the Gödel number of $\mathbf{C}$ is $c$.

Does $\mathbf{C}$ derive a string whose translation is $c$?
Be $\mathbf{C}$ a calculus with this property (deriving its own Gödel number).
Then $\mathbf{H}_3$ derives $aDc$, too.
Let us call such $c$-s autonomous numbers.

# Autonomous numerals

Be $\mathbf{C}$ an arbitrary calculus.
The translation of $\mathbf{C}$ into our language is some $\mathcal{A}_{cc}$-word $a$.
$\mathbf{H}_3$ derives $Ka$.
There is a numeral $c$ s.t. $\mathbf{H}_3$ derives $aGc$, i. e. the Gödel number of $\mathbf{C}$ is $c$.

Does $\mathbf{C}$ derive a string whose translation is $c$?
Be $\mathbf{C}$ a calculus with this property (deriving its own Gödel number).
Then $\mathbf{H}_3$ derives $aDc$, too.
Let us call such $c$-s autonomous numbers.
Let us extend $\mathbf{H}_3$ to define autonomous numbers.
New auxiliary letter: $A$ with the intended meaning "autonomous". New rule:

# Autonomous numerals

Be $\mathbf{C}$ an arbitrary calculus.
The translation of $\mathbf{C}$ into our language is some $\mathcal{A}_{cc}$-word $a$.
$\mathbf{H}_3$ derives $Ka$.
There is a numeral $c$ s.t. $\mathbf{H}_3$ derives $aGc$, i. e. the Gödel number of $\mathbf{C}$ is $c$.

Does $\mathbf{C}$ derive a string whose translation is $c$?
Be $\mathbf{C}$ a calculus with this property (deriving its own Gödel number).
Then $\mathbf{H}_3$ derives $aDc$, too.
Let us call such $c$-s autonomous numbers.
Let us extend $\mathbf{H}_3$ to define autonomous numbers.
New auxiliary letter: $A$ with the intended meaning "autonomous". New rule:

$$34. \quad xDy \to xGy \to Ay$$

# Our Gödel-like theorem

# Our Gödel-like theorem

The numbers are the strings of the one-letter alphabet $\mathcal{A}_0 = \{\alpha\}$, so their class is $\mathcal{A}_0^\circ$ and it can be defined inductively. The class of autonomous numerals, in class theoretic notation:

$$\boldsymbol{Aut} = \{x : x \in \mathcal{A}_0^\circ \wedge \mathbf{H}_3 \mapsto Ax\}$$

By adding a release rule to $\mathbf{H}_3$ deleting $A$, we gain a definition of $\boldsymbol{Aut}$ by a canonical calculus.

# Our Gödel-like theorem

The numbers are the strings of the one-letter alphabet $\mathcal{A}_0 = \{\alpha\}$, so their class is $\mathcal{A}_0^\circ$ and it can be defined inductively. The class of autonomous numerals, in class theoretic notation:

$$\boldsymbol{Aut} = \{x : x \in \mathcal{A}_0^\circ \wedge \mathbf{H}_3 \mapsto Ax\}$$

By adding a release rule to $\mathbf{H}_3$ deleting $A$, we gain a definition of $\boldsymbol{Aut}$ by a canonical calculus.

We prove that the string class $\mathcal{A}_0^\circ - \boldsymbol{Aut}$ (the class of non-autonomous numerals) cannot be defined inductively.

# Our Gödel-like theorem

The numbers are the strings of the one-letter alphabet $\mathcal{A}_0 = \{\alpha\}$, so their class is $\mathcal{A}_0^\circ$ and it can be defined inductively. The class of autonomous numerals, in class theoretic notation:

$$\boldsymbol{Aut} = \{x : x \in \mathcal{A}_0^\circ \wedge \mathbf{H}_3 \mapsto Ax\}$$

By adding a release rule to $\mathbf{H}_3$ deleting $A$, we gain a definition of $\boldsymbol{Aut}$ by a canonical calculus.

We prove that the string class $\mathcal{A}_0^\circ - \boldsymbol{Aut}$ (the class of non-autonomous numerals) cannot be defined inductively.

**Theorem**: There is no canonical calculus $\mathbf{C}$ over some $B \supseteq \mathcal{A}_{cc}$ s.t. for any string $x$,

$$\mathbf{C} \mapsto x \Leftrightarrow x \in \mathcal{A}_0^\circ - \boldsymbol{Aut}$$

# Proof of the theorem

# Proof of the theorem

Let us assume toward contradiction that we have a calculus $\mathbf{C}$ with the Gödel number $g$ s.t for every non-autonomous numeral $c$, $\mathbf{C} \mapsto c$, and there is no autonomous numeral $d$ for that $\mathbf{C} \mapsto d$.

# Proof of the theorem

Let us assume toward contradiction that we have a calculus $\mathbf{C}$ with the Gödel number $g$ s.t for every non-autonomous numeral $c$, $\mathbf{C} \mapsto c$, and there is no autonomous numeral $d$ for that $\mathbf{C} \mapsto d$.

Suppose that $\mathbf{C} \mapsto g$. In this case, $\mathbf{C}$ is an autonomous calculus, $g$ is an autonomous number, therefore $\mathbf{C}$ does not derive $g$. Contradiction.

# Proof of the theorem

Let us assume toward contradiction that we have a calculus $\mathbf{C}$ with the Gödel number $g$ s.t for every non-autonomous numeral $c$, $\mathbf{C} \mapsto c$, and there is no autonomous numeral $d$ for that $\mathbf{C} \mapsto d$.

Suppose that $\mathbf{C} \mapsto g$. In this case, $\mathbf{C}$ is an autonomous calculus, $g$ is an autonomous number, therefore $\mathbf{C}$ does not derive $g$. Contradiction.

Suppose that $\mathbf{C}$ does not derive $g$. In this case, $\mathbf{C}$ is not an autonomous calculus, $g$ is a non-autonomous number, therefore $\mathbf{C} \mapsto g$. Contradiction again, q.e.d.

# Proof of the theorem

Let us assume toward contradiction that we have a calculus $\mathbf{C}$ with the Gödel number $g$ s.t for every non-autonomous numeral $c$, $\mathbf{C} \mapsto c$, and there is no autonomous numeral $d$ for that $\mathbf{C} \mapsto d$.

Suppose that $\mathbf{C} \mapsto g$. In this case, $\mathbf{C}$ is an autonomous calculus, $g$ is an autonomous number, therefore $\mathbf{C}$ does not derive $g$. Contradiction.

Suppose that $\mathbf{C}$ does not derive $g$. In this case, $\mathbf{C}$ is not an autonomous calculus, $g$ is a non-autonomous number, therefore $\mathbf{C} \mapsto g$. Contradiction again, q.e.d.

This theorem is Gödel-like because it shows that no inductive definition can be given for the notion "non-autonomous calculus" just like Gödel's first incompleteness theorem shows that no inductive definition can be given for the notion "arithmetical truth". And this proof uses an analogue of the Liar Paradox, too.

# Enumerability

# Enumerability

In general, if we have a calculus to define some string class, we have an effective process to enumerate its members. We can enumerate the derivations in the calculus: first, the one-member derivations, then the two-member ones, etc. For any given $n$, the set of the $n$-member derivations is finite.

# Enumerability

In general, if we have a calculus to define some string class, we have an effective process to enumerate its members. We can enumerate the derivations in the calculus: first, the one-member derivations, then the two-member ones, etc. For any given $n$, the set of the $n$-member derivations is finite.

The enumeration of derivations produces an enumeration of the derivable strings too. This informal consideration shows that inductively defined classes are effectively enumerable, i. e., we have a procedure that enumerates all of its members. What about the conversion of this claim? Is every effectively enumerable class inductively definable? We don't have an answer yet.

# Enumerability and decidability

# Enumerability and decidability

If we had a calculus for the non-autonomous numerals we would have an enumeration of the non-autonomous numerals, too. In that case we could decide about any given numeral $n$ whether it is an autonomous numeral or not.

# Enumerability and decidability

If we had a calculus for the non-autonomous numerals we would have an enumeration of the non-autonomous numerals, too. In that case we could decide about any given numeral $n$ whether it is an autonomous numeral or not.

Imagine that a printing machine prints the autonomous numbers in the order of enumeration and another one the non-autonomous numbers. After a finite time, $n$ will occur as an output of either the first or the second machine and therefore we have a *decision procedure* for the membership of the class.

# Enumerability and decidability

If we had a calculus for the non-autonomous numerals we would have an enumeration of the non-autonomous numerals, too. In that case we could decide about any given numeral $n$ whether it is an autonomous numeral or not.

Imagine that a printing machine prints the autonomous numbers in the order of enumeration and another one the non-autonomous numbers. After a finite time, $n$ will occur as an output of either the first or the second machine and therefore we have a *decision procedure* for the membership of the class.

The converse of the claim is obvious: if we have a decision procedure then we can enumerate both the set and its complement. Therefore we have an enumeration procedure both for a string class $\mathcal{B}$ over an alphabet $\mathcal{A}$ and its complement $\mathcal{A}^\circ - \mathcal{B}$ if and only if we have a decision procedure for $\mathcal{B}$.

# Enumerability and decidability

If we had a calculus for the non-autonomous numerals we would have an enumeration of the non-autonomous numerals, too. In that case we could decide about any given numeral $n$ whether it is an autonomous numeral or not.

Imagine that a printing machine prints the autonomous numbers in the order of enumeration and another one the non-autonomous numbers. After a finite time, $n$ will occur as an output of either the first or the second machine and therefore we have a *decision procedure* for the membership of the class.

The converse of the claim is obvious: if we have a decision procedure then we can enumerate both the set and its complement. Therefore we have an enumeration procedure both for a string class $\mathcal{B}$ over an alphabet $\mathcal{A}$ and its complement $\mathcal{A}^\circ - \mathcal{B}$ if and only if we have a decision procedure for $\mathcal{B}$.

Our next task is to make precise and formally defined the notions used above: 'procedure', 'effective enumeration'.

# The open question

# The open question

We know that the string class $\mathcal{A}_0^\circ - \boldsymbol{Aut}$ is not inductively definable. Does it mean that it is not effectively enumerable, either?

# The open question

We know that the string class $\mathcal{A}_0^\circ - \boldsymbol{Aut}$ is not inductively definable. Does it mean that it is not effectively enumerable, either?

Generalization: If a string class is not inductively definable, dores it imply that the class is not effectively enumerable, either?

# The open question

We know that the string class $\mathcal{A}_0^\circ - \boldsymbol{Aut}$ is not inductively definable. Does it mean that it is not effectively enumerable, either?

Generalization: If a string class is not inductively definable, dores it imply that the class is not effectively enumerable, either?

Contrapositive form of the above (generalized) question: Is it true that an effectively enumerable class is always inductively definable?

# The open question

We know that the string class $\mathcal{A}_0^\circ - \boldsymbol{Aut}$ is not inductively definable. Does it mean that it is not effectively enumerable, either?

Generalization: If a string class is not inductively definable, dores it imply that the class is not effectively enumerable, either?

Contrapositive form of the above (generalized) question: Is it true that an effectively enumerable class is always inductively definable?

If the answer is 'yes', then the class of autonomous numerals is not decidable (although it is enumerable).

# The open question

We know that the string class $\mathcal{A}_0^\circ - \boldsymbol{Aut}$ is not inductively definable. Does it mean that it is not effectively enumerable, either?

Generalization: If a string class is not inductively definable, dores it imply that the class is not effectively enumerable, either?

Contrapositive form of the above (generalized) question: Is it true that an effectively enumerable class is always inductively definable?

If the answer is 'yes', then the class of autonomous numerals is not decidable (although it is enumerable).

But to establish such an answer, we need a (formal) notion of effective procedure.

# Procedures, algorithms

# Procedures, algorithms

A *procedure* or *algorithm* is a set of commands that you should perform in a prescribed sequence in order to solve a task of some type (class of tasks). Some well-known sorts of procedures:

# Procedures, algorithms

A *procedure* or *algorithm* is a set of commands that you should perform in a prescribed sequence in order to solve a task of some type (class of tasks). Some well-known sorts of procedures:

*Operations.* Example: multiplication of numerals. Given any pair of numerals, produce a numeral which denotes the product of the two numbers.

# Procedures, algorithms

A *procedure* or *algorithm* is a set of commands that you should perform in a prescribed sequence in order to solve a task of some type (class of tasks). Some well-known sorts of procedures:

*Operations.* Example: multiplication of numerals. Given any pair of numerals, produce a numeral which denotes the product of the two numbers.

*Decision procedures.* Example: Given a string from $\mathcal{A}^{\circ}_{Language(FOL)}$, decide whether it is a formula of $Language(FOL)$ or not.

# Procedures, algorithms

A *procedure* or *algorithm* is a set of commands that you should perform in a prescribed sequence in order to solve a task of some type (class of tasks). Some well-known sorts of procedures:

*Operations.* Example: multiplication of numerals. Given any pair of numerals, produce a numeral which denotes the product of the two numbers.

*Decision procedures.* Example: Given a string from $\mathcal{A}^{\circ}_{Language(FOL)}$, decide whether it is a formula of $Language(FOL)$ or not.

*Enumeration procedure for a given sequence (of strings).* Example: from any string of the alphabet $\mathcal{A}_{cc}$, produce the next string in the lexicographic ordering.

# Markov algorithms

# Markov algorithms

Ways to formalize the notion of (finite, effectively performable) procedure: Turing machines, recursive functions, lambda calculus etc. We will use Markov algorithms.

# Markov algorithms

Ways to formalize the notion of (finite, effectively performable) procedure: Turing machines, recursive functions, lambda calculus etc. We will use Markov algorithms.

A calculus tells us what we are *allowed* to do, an algorithm prescribes what we *must* do.

# Markov algorithms

Ways to formalize the notion of (finite, effectively performable) procedure: Turing machines, recursive functions, lambda calculus etc. We will use Markov algorithms.

A calculus tells us what we are *allowed* to do, an algorithm prescribes what we *must* do.

Markov algorithms transform strings of a given alphabet into other strings. Every step of the algorithm is a substitution of a string by another string, prescribed by the commands of the algorithm and their order.

# Markov algorithms

Ways to formalize the notion of (finite, effectively performable) procedure: Turing machines, recursive functions, lambda calculus etc. We will use Markov algorithms.

A calculus tells us what we are *allowed* to do, an algorithm prescribes what we *must* do.

Markov algorithms transform strings of a given alphabet into other strings. Every step of the algorithm is a substitution of a string by another string, prescribed by the commands of the algorithm and their order.

Markov algorithm (or normal algorithm) over an alphabet $\mathcal{A}$ (not containing the characters '$\rightarrow$' and '$\cdot$') is a finite, nonempty sequence $N$ of $\mathcal{A}$-commands.

# Markov algorithms

Ways to formalize the notion of (finite, effectively performable) procedure: Turing machines, recursive functions, lambda calculus etc. We will use Markov algorithms.

A calculus tells us what we are *allowed* to do, an algorithm prescribes what we *must* do.

Markov algorithms transform strings of a given alphabet into other strings. Every step of the algorithm is a substitution of a string by another string, prescribed by the commands of the algorithm and their order.

Markov algorithm (or normal algorithm) over an alphabet $\mathcal{A}$ (not containing the characters '$\to$' and '$\cdot$') is a finite, nonempty sequence $N$ of $\mathcal{A}$-commands.

An $\mathcal{A}$-command is a string of the form $\ulcorner a \to b \urcorner$ or $\ulcorner a \to \cdot b \urcorner$ where $a$ (the input of the command) and $b$ (the output) are $\mathcal{A}$-strings. Commands of the latter form are called stop commands.

# Application of a command resp. algorithm to a word

# Application of a command resp. algorithm to a word

The command $C = a \to b$ resp. $a \to \cdot b$ is applicable to a string $f$ if its input $a$ occurs as a sub-string in $f$, i.e. $f = u^\frown a^\frown v$, where $u$ and $v$ can be any string over $\mathcal{A}$.

# Application of a command resp. algorithm to a word

The command $C = a \to b$ resp. $a \to \cdot b$ is applicable to a string $f$ if its input $a$ occurs as a sub-string in $f$, i.e. $f = u^\frown a^\frown v$, where $u$ and $v$ can be any string over $\mathcal{A}$.

The application of $C$ to $f$ is the substitution of the *first* occurrence of $a$ in $f$ by $b$. The result: $C(f)$.

# Application of a command resp. algorithm to a word

The command $C = a \to b$ resp. $a \to \cdot b$ is applicable to a string $f$ if its input $a$ occurs as a sub-string in $f$, i.e. $f = u^\frown a^\frown v$, where $u$ and $v$ can be any string over $\mathcal{A}$.

The application of $C$ to $f$ is the substitution of the *first* occurrence of $a$ in $f$ by $b$. The result: $C(f)$.

Steps of the application of an algorithm $N$ to a string $f_0$ (informally):

# Application of a command resp. algorithm to a word

The command $C = a \to b$ resp. $a \to \cdot b$ is applicable to a string $f$ if its input $a$ occurs as a sub-string in $f$, i.e. $f = u^\frown a^\frown v$, where $u$ and $v$ can be any string over $\mathcal{A}$.

The application of $C$ to $f$ is the substitution of the *first* occurrence of $a$ in $f$ by $b$. The result: $C(f)$.

Steps of the application of an algorithm $N$ to a string $f_0$ (informally):

1. If no command in $N$ is applicable to $f_0$, then $f_0$ blocks $N$, in symbols, $N(f) = \sharp$ ($\sharp \notin \mathcal{A}$).

# Application of a command resp. algorithm to a word

The command $C = a \to b$ resp. $a \to \cdot b$ is applicable to a string $f$ if its input $a$ occurs as a sub-string in $f$, i.e. $f = u^\frown a^\frown v$, where $u$ and $v$ can be any string over $\mathcal{A}$.

The application of $C$ to $f$ is the substitution of the *first* occurrence of $a$ in $f$ by $b$. The result: $C(f)$.

Steps of the application of an algorithm $N$ to a string $f_0$ (informally):

1. If no command in $N$ is applicable to $f_0$, then $f_0$ blocks $N$, in symbols, $N(f) = \sharp$ ($\sharp \notin \mathcal{A}$).
2. Otherwise, apply the *first* applicable command $C_0$ to $f_0$. The result is $f_1 = C_0(f_0)$.

# Application of a command resp. algorithm to a word

The command $C = a \to b$ resp. $a \to \cdot b$ is applicable to a string $f$ if its input $a$ occurs as a sub-string in $f$, i.e. $f = u^\frown a^\frown v$, where $u$ and $v$ can be any string over $\mathcal{A}$.

The application of $C$ to $f$ is the substitution of the *first* occurrence of $a$ in $f$ by $b$. The result: $C(f)$.

Steps of the application of an algorithm $N$ to a string $f_0$ (informally):

1. If no command in $N$ is applicable to $f_0$, then $f_0$ blocks $N$, in symbols, $N(f) = \sharp$ ($\sharp \notin \mathcal{A}$).
2. Otherwise, apply the *first* applicable command $C_0$ to $f_0$. The result is $f_1 = C_0(f_0)$.
3. If $C_0$ was a stop command, then $N$ applies to $f_0$ and transforms it to $f_1$. In symbols, $N(f_0) = f_1$.

# Application of a command resp. algorithm to a word

The command $C = a \to b$ resp. $a \to \cdot b$ is applicable to a string $f$ if its input $a$ occurs as a sub-string in $f$, i.e. $f = u^\frown a^\frown v$, where $u$ and $v$ can be any string over $\mathcal{A}$.

The application of $C$ to $f$ is the substitution of the *first* occurrence of $a$ in $f$ by $b$. The result: $C(f)$.

Steps of the application of an algorithm $N$ to a string $f_0$ (informally):

1. If no command in $N$ is applicable to $f_0$, then $f_0$ blocks $N$, in symbols, $N(f) = \sharp$ ($\sharp \notin \mathcal{A}$).
2. Otherwise, apply the *first* applicable command $C_0$ to $f_0$. The result is $f_1 = C_0(f_0)$.
3. If $C_0$ was a stop command, then $N$ applies to $f_0$ and transforms it to $f_1$. In symbols, $N(f_0) = f_1$.
4. If it was not, then $N$ leads $f_0$ to $f_1$ (in symbols, $N(f_0/f_1)$) and the algorithm continues with step 1, but $f_1$ takes the place of $f_0$. If we arrive to a stop command, then the original string, $f_0$ is transformed into the last result.

# Possible outcomes of the application of an algorithm

# Possible outcomes of the application of an algorithm

If we try to apply an algorithm $N$ to a string $f$, there are three possibilities:

# Possible outcomes of the application of an algorithm

If we try to apply an algorithm $N$ to a string $f$, there are three possibilities:

1. After performing finitely many times the steps above, we arrive to a situation that no command in $N$ applies to our last result. In this case, $N$ does not apply to $f$ or $f$ blocks $N$, $N(f) = \sharp$.

# Possible outcomes of the application of an algorithm

If we try to apply an algorithm $N$ to a string $f$, there are three possibilities:

1. After performing finitely many times the steps above, we arrive to a situation that no command in $N$ applies to our last result. In this case, $N$ does not apply to $f$ or $f$ blocks $N$, $N(f) = \sharp$.
2. After finitely many steps, we arrive to a stop command. If the result of the application of this command was $g$, then $N$ applies to $f$ and transforms it to $g$, $N(f) = g$.

# Possible outcomes of the application of an algorithm

If we try to apply an algorithm $N$ to a string $f$, there are three possibilities:

1. After performing finitely many times the steps above, we arrive to a situation that no command in $N$ applies to our last result. In this case, $N$ does not apply to $f$ or $f$ blocks $N$, $N(f) = \sharp$.
2. After finitely many steps, we arrive to a stop command. If the result of the application of this command was $g$, then $N$ applies to $f$ and transforms it to $g$, $N(f) = g$.
3. We never arrive after finitely many steps to a stop command, nor to a blocking situation. In this case, $N$ runs infinitely on $f$.

# Possible outcomes of the application of an algorithm

If we try to apply an algorithm $N$ to a string $f$, there are three possibilities:

1. After performing finitely many times the steps above, we arrive to a situation that no command in $N$ applies to our last result. In this case, $N$ does not apply to $f$ or $f$ blocks $N$, $N(f) = \sharp$.
2. After finitely many steps, we arrive to a stop command. If the result of the application of this command was $g$, then $N$ applies to $f$ and transforms it to $g$, $N(f) = g$.
3. We never arrive after finitely many steps to a stop command, nor to a blocking situation. In this case, $N$ runs infinitely on $f$.

The first case can be avoided by inserting the command $\varnothing \to \cdot\varnothing$ to the end of the algorithm. It is applicable to any string and does nothing but stops the algorithm.

# Formal definitions of the above notions

# Formal definitions of the above notions

Simultaneous inductive definition of the relations $N(f) = \sharp$ ($f$ blocks $N$), $N(f) = g$ ($N$ transforms $f$ into $g$) and $N(f/g)$ ($N$ leads $f$ to $g$). ($N$ is an algorithm over $\mathcal{A}$, $f$ and $g$ are $\mathcal{A}$-strings and $\sharp \notin \mathcal{A}$.)

# Formal definitions of the above notions

Simultaneous inductive definition of the relations $N(f) = \sharp$ ($f$ blocks $N$), $N(f) = g$ ($N$ transforms $f$ into $g$) and $N(f/g)$ ($N$ leads $f$ to $g$). ($N$ is an algorithm over $\mathcal{A}$, $f$ and $g$ are $\mathcal{A}$-strings and $\sharp \notin \mathcal{A}$.)

# Formal definitions of the above notions

Simultaneous inductive definition of the relations $N(f) = \sharp$ ($f$ blocks $N$), $N(f) = g$ ($N$ transforms $f$ into $g$) and $N(f/g)$ ($N$ leads $f$ to $g$). ($N$ is an algorithm over $\mathcal{A}$, $f$ and $g$ are $\mathcal{A}$-strings and $\sharp \notin \mathcal{A}$.)

- If no command in $N$ is applicable to $f$, then $N(f) = \sharp$.

# Formal definitions of the above notions

Simultaneous inductive definition of the relations $N(f) = \sharp$ ($f$ blocks $N$), $N(f) = g$ ($N$ transforms $f$ into $g$) and $N(f/g)$ ($N$ leads $f$ to $g$). ($N$ is an algorithm over $\mathcal{A}$, $f$ and $g$ are $\mathcal{A}$-strings and $\sharp \notin \mathcal{A}$.)

i. If no command in $N$ is applicable to $f$, then $N(f) = \sharp$.

ii. If $C$ is the first command in $N$ that is applicable to $f$, $C(f) = g$, then
   a. if $C$ is a stop command, then $N(f) = g$;
   b. if $C$ is not a stop command, then $N(f/g)$.

# Formal definitions of the above notions

Simultaneous inductive definition of the relations $N(f) = \sharp$ ($f$ blocks $N$), $N(f) = g$ ($N$ transforms $f$ into $g$) and $N(f/g)$ ($N$ leads $f$ to $g$). ($N$ is an algorithm over $\mathcal{A}$, $f$ and $g$ are $\mathcal{A}$-strings and $\sharp \notin \mathcal{A}$.)

i. If no command in $N$ is applicable to $f$, then $N(f) = \sharp$.
ii. If $C$ is the first command in $N$ that is applicable to $f$, $C(f) = g$, then
    a. if $C$ is a stop command, then $N(f) = g$;
    b. if $C$ is not a stop command, then $N(f/g)$.
iii. If $N(f/g)$ and $N(g/h)$, then $N(f/h)$.

# Formal definitions of the above notions

Simultaneous inductive definition of the relations $N(f) = \sharp$ ($f$ blocks $N$), $N(f) = g$ ($N$ transforms $f$ into $g$) and $N(f/g)$ ($N$ leads $f$ to $g$). ($N$ is an algorithm over $\mathcal{A}$, $f$ and $g$ are $\mathcal{A}$-strings and $\sharp \notin \mathcal{A}$.)

i. If no command in $N$ is applicable to $f$, then $N(f) = \sharp$.
ii. If $C$ is the first command in $N$ that is applicable to $f$, $C(f) = g$, then
    a. if $C$ is a stop command, then $N(f) = g$;
    b. if $C$ is not a stop command, then $N(f/g)$.
iii. If $N(f/g)$ and $N(g/h)$, then $N(f/h)$.
iv. If $N(f/g)$ and $N(g) = h$, then $N(f) = h$.

# Formal definitions of the above notions

Simultaneous inductive definition of the relations $N(f) = \sharp$ ($f$ blocks $N$), $N(f) = g$ ($N$ transforms $f$ into $g$) and $N(f/g)$ ($N$ leads $f$ to $g$). ($N$ is an algorithm over $\mathcal{A}$, $f$ and $g$ are $\mathcal{A}$-strings and $\sharp \notin \mathcal{A}$.)

i. If no command in $N$ is applicable to $f$, then $N(f) = \sharp$.
ii. If $C$ is the first command in $N$ that is applicable to $f$, $C(f) = g$, then
    a. if $C$ is a stop command, then $N(f) = g$;
    b. if $C$ is not a stop command, then $N(f/g)$.
iii. If $N(f/g)$ and $N(g/h)$, then $N(f/h)$.
iv. If $N(f/g)$ and $N(g) = h$, then $N(f) = h$.
v. If $N(f/g)$ and $N(g) = \sharp$, then $N(f) = \sharp$.

# Examples

# Examples

**Erase a letter**. Be $a \epsilon \mathcal{A}$. Let us erase every occurrence of $a$ from any string.

# Examples

**Erase a letter**. Be $a \epsilon \mathcal{A}$. Let us erase every occurrence of $a$ from any string.

1. $a \rightarrow \varnothing$
2. $\varnothing \rightarrow \cdot\varnothing$

# Examples

**Erase a letter**. Be $a \epsilon \mathcal{A}$. Let us erase every occurrence of $a$ from any string.

1. $a \rightarrow \varnothing$
2. $\varnothing \rightarrow \cdot\varnothing$

**Erase every letter**.

# Examples

**Erase a letter**. Be $a \epsilon \mathcal{A}$. Let us erase every occurrence of $a$ from any string.

1. $a \rightarrow \varnothing$
2. $\varnothing \rightarrow \cdot\varnothing$

**Erase every letter**.

1. $x \rightarrow \varnothing \qquad x \in \mathcal{A}$
2. $\varnothing \rightarrow \cdot\varnothing$

# Examples

**Erase a letter**. Be $a \epsilon \mathcal{A}$. Let us erase every occurrence of $a$ from any string.

1. $a \rightarrow \varnothing$
2. $\varnothing \rightarrow \cdot\varnothing$

**Erase every letter**.

1. $x \rightarrow \varnothing \qquad x \in \mathcal{A}$
2. $\varnothing \rightarrow \cdot\varnothing$

The letter $x$ is a metalanguage variable for letters and the first command is an usual and obvious abbreviation of $n$ commands, if $\mathcal{A}$ has $n$ members.

# A mirroring algorithm

# A mirroring algorithm

We can use auxiliary letters in algorithms as well as in calculi. It means only that to solve algorithmically some problem concerning the $\mathcal{A}$-strings, we write an algorithm over some $\mathcal{B} \supset \mathcal{A}$ and *we regard* the members of $\mathcal{B} - \mathcal{A}$ auxiliary letters.

# A mirroring algorithm

We can use auxiliary letters in algorithms as well as in calculi. It means only that to solve algorithmically some problem concerning the $\mathcal{A}$-strings, we write an algorithm over some $\mathcal{B} \supset \mathcal{A}$ and *we regard* the members of $\mathcal{B} - \mathcal{A}$ auxiliary letters.

The following algorithm brings any $\mathcal{A}$-string $a_0 a_1 \dots a_n$ into the string $a_0 a_1 \dots a_n \mid a_n a_{n-1} \dots a_0$ ($\mid \notin \mathcal{A}$, and the algorithm uses the auxiliary letters $A$, $C$, too.).

# A mirroring algorithm

We can use auxiliary letters in algorithms as well as in calculi. It means only that to solve algorithmically some problem concerning the $\mathcal{A}$-strings, we write an algorithm over some $\mathcal{B} \supset \mathcal{A}$ and *we regard* the members of $\mathcal{B} - \mathcal{A}$ auxiliary letters.

The following algorithm brings any $\mathcal{A}$-string $a_0a_1 \dots a_n$ into the string $a_0a_1 \dots a_n \mid a_na_{n-1} \dots a_0$ ($\mid \notin \mathcal{A}$, and the algorithm uses the auxiliary letters $A$, $C$, too.).

| | | |
|---|---|---|
| 1. | $Cxy \to yCx$ | $x \in \mathcal{A},\ y \in \mathcal{A} \cup \{\mid\}$ |
| 2. | $Cx \to x$ | $x \in \mathcal{A}$ |
| 3. | $xA \to AxCx$ | $x \in \mathcal{A}$ |
| 4. | $A \to .\varnothing$ | |
| 5. | $\mid x \to x \mid$ | $x \in \mathcal{A}$ |
| 6. | $x \mid \to xA \mid$ | $x \in \mathcal{A}$ |
| 7. | $\varnothing \to \mid$ | |

# Homework

# Homework

Write an algorithm that decides identity of strings of some alphabet $\mathcal{A}$ in the following sense: Let $V$ and $W$ arbitrary $\mathcal{A}$-strings. Your algorithm should transform the string $V \mid W$ into $Y$ if they are the same string, and in $N$ if they are different. ($Y$, $\mid$ and $N$ are auxiliary letters.)