The Grand Unified Theory of Computation (Part 2)

Vahan Mkrtchyan [1]

[1] Lane Department of Computer Science and Electrical Engineering
West Virginia University

April 22, 2015

Outline

Outline

Outline

Alonzo Church and the λ-Calculus

Alonzo Church and the λ-Calculus

The λ-Calculus

Alonzo Church and the λ-Calculus

The λ-Calculus

The logician Alonzo Church has come up with another definition of a computation and a computable function.

Alonzo Church and the λ-Calculus

The λ-Calculus

The logician Alonzo Church has come up with another definition of a computation and a computable function. The system he has invented is called λ-Calculus.

Alonzo Church and the λ-Calculus

The λ-Calculus

The logician Alonzo Church has come up with another definition of a computation and a computable function. The system he has invented is called λ-Calculus.

Getting used to the new format

Alonzo Church and the λ-Calculus

The λ-Calculus

The logician Alonzo Church has come up with another definition of a computation and a computable function. The system he has invented is called λ-Calculus.

Getting used to the new format

Instead of writing $f(x) = x^2$,

Alonzo Church and the λ-Calculus

The λ-Calculus

The logician Alonzo Church has come up with another definition of a computation and a computable function. The system he has invented is called λ-Calculus.

Getting used to the new format

Instead of writing $f(x) = x^2$, we will write $f = \lambda x.x^2$.

Alonzo Church and the λ-Calculus

The λ-Calculus

The logician Alonzo Church has come up with another definition of a computation and a computable function. The system he has invented is called λ-Calculus.

Getting used to the new format

Instead of writing $f(x) = x^2$, we will write $f = \lambda x.x^2$.

In other words, $\lambda x.x^2$ is a function that takes a variable x and returns x^2.

Alonzo Church and the λ-Calculus

The λ-Calculus

The logician Alonzo Church has come up with another definition of a computation and a computable function. The system he has invented is called λ-Calculus.

Getting used to the new format

Instead of writing $f(x) = x^2$, we will write $f = \lambda x.x^2$.

In other words, $\lambda x.x^2$ is a function that takes a variable x and returns x^2.

If we replace a number to the right of this function,

Alonzo Church and the λ-Calculus

The λ-Calculus

The logician Alonzo Church has come up with another definition of a computation and a computable function. The system he has invented is called λ-Calculus.

Getting used to the new format

Instead of writing $f(x) = x^2$, we will write $f = \lambda x.x^2$.

In other words, $\lambda x.x^2$ is a function that takes a variable x and returns x^2.

If we replace a number to the right of this function, it substitutes that number for the variable x:

Alonzo Church and the λ-Calculus

The λ-Calculus

The logician Alonzo Church has come up with another definition of a computation and a computable function. The system he has invented is called λ-Calculus.

Getting used to the new format

Instead of writing $f(x) = x^2$, we will write $f = \lambda x.x^2$.

In other words, $\lambda x.x^2$ is a function that takes a variable x and returns x^2.

If we replace a number to the right of this function, it substitutes that number for the variable x:

$$(\lambda x.x^2)7 = 49$$

Alonzo Church and the λ-Calculus

The λ-Calculus

The logician Alonzo Church has come up with another definition of a computation and a computable function. The system he has invented is called λ-Calculus.

Getting used to the new format

Instead of writing $f(x) = x^2$, we will write $f = \lambda x.x^2$.

In other words, $\lambda x.x^2$ is a function that takes a variable x and returns x^2.

If we replace a number to the right of this function, it substitutes that number for the variable x:

$$(\lambda x.x^2)7 = 49$$

So far, this is just a strange shift in notation.

Alonzo Church and the λ-Calculus

Alonzo Church and the λ-Calculus

The λ-Calculus

Alonzo Church and the λ-Calculus

The λ-Calculus

The way, the λ-Calculus treats the process of computation is different.

Alonzo Church and the λ-Calculus

The λ-Calculus

The way, the λ-Calculus treats the process of computation is different.

Consider the addition function,

Alonzo Church and the λ-Calculus

The λ-Calculus

The way, the λ-Calculus treats the process of computation is different.

Consider the addition function, which we write

$$\lambda xy.x + y.$$

Alonzo Church and the λ-Calculus

The λ-Calculus

The way, the λ-Calculus treats the process of computation is different.

Consider the addition function, which we write

$$\lambda xy.x + y.$$

We will apply this to 3 and 5.

Alonzo Church and the λ-Calculus

The λ-Calculus

The way, the λ-Calculus treats the process of computation is different.

Consider the addition function, which we write

$$\lambda xy.x + y.$$

We will apply this to 3 and 5. Rather than writing

$$(\lambda xy.x + y)(3, 5)$$

Alonzo Church and the λ-Calculus

The λ-Calculus

The way, the λ-Calculus treats the process of computation is different.

Consider the addition function, which we write

$$\lambda xy.x + y.$$

We will apply this to 3 and 5. Rather than writing

$$(\lambda xy.x + y)(3, 5)$$

we write

$$((\lambda xy.x + y)3)5$$

Alonzo Church and the λ-Calculus

The λ-Calculus

The way, the λ-Calculus treats the process of computation is different.

Consider the addition function, which we write

$$\lambda xy.x + y.$$

We will apply this to 3 and 5. Rather than writing

$$(\lambda xy.x + y)(3, 5)$$

we write

$$((\lambda xy.x + y)3)5 = (\lambda y.3 + y)5$$

Alonzo Church and the λ-Calculus

The λ-Calculus

The way, the λ-Calculus treats the process of computation is different.

Consider the addition function, which we write

$$\lambda xy.x + y.$$

We will apply this to 3 and 5. Rather than writing

$$(\lambda xy.x + y)(3, 5)$$

we write

$$((\lambda xy.x + y)3)5 = (\lambda y.3 + y)5 = 8.$$

Alonzo Church and the λ-Calculus

The λ-Calculus

The way, the λ-Calculus treats the process of computation is different.

Consider the addition function, which we write

$$\lambda xy.x + y.$$

We will apply this to 3 and 5. Rather than writing

$$(\lambda xy.x + y)(3, 5)$$

we write

$$((\lambda xy.x + y)3)5 = (\lambda y.3 + y)5 = 8.$$

This is already a computation.

Alonzo Church and the λ-Calculus

The λ-Calculus

The way, the λ-Calculus treats the process of computation is different.

Consider the addition function, which we write

$$\lambda xy.x + y.$$

We will apply this to 3 and 5. Rather than writing

$$(\lambda xy.x + y)(3, 5)$$

we write

$$((\lambda xy.x + y)3)5 = (\lambda y.3 + y)5 = 8.$$

This is already a computation. We have the object $\lambda y.3 + y$.

Alonzo Church and the λ-Calculus

The λ-Calculus

The way, the λ-Calculus treats the process of computation is different.

Consider the addition function, which we write

$$\lambda xy.x + y.$$

We will apply this to 3 and 5. Rather than writing

$$(\lambda xy.x + y)(3, 5)$$

we write

$$((\lambda xy.x + y)3)5 = (\lambda y.3 + y)5 = 8.$$

This is already a computation. We have the object $\lambda y.3 + y$. In other words, having fixed $x = 3$, we have a function on the other variable y,

Alonzo Church and the λ-Calculus

The λ-Calculus

The way, the λ-Calculus treats the process of computation is different.

Consider the addition function, which we write

$$\lambda xy.x + y.$$

We will apply this to 3 and 5. Rather than writing

$$(\lambda xy.x + y)(3, 5)$$

we write

$$((\lambda xy.x + y)3)5 = (\lambda y.3 + y)5 = 8.$$

This is already a computation. We have the object $\lambda y.3 + y$. In other words, having fixed $x = 3$, we have a function on the other variable y, which adds 3 to it.

Alonzo Church and the λ-Calculus

Alonzo Church and the λ-Calculus

The λ-Calculus

Alonzo Church and the λ-Calculus

The λ-Calculus

This is a powerful shift in how we think about performing a computation.

Alonzo Church and the λ-Calculus

The λ-Calculus

This is a powerful shift in how we think about performing a computation.

Rather than modifying variables and passing their values up to functions,

Alonzo Church and the λ-Calculus

The λ-Calculus

This is a powerful shift in how we think about performing a computation.

Rather than modifying variables and passing their values up to functions, we evaluate functions from the top down,

Alonzo Church and the λ-Calculus

The λ-Calculus

This is a powerful shift in how we think about performing a computation.

Rather than modifying variables and passing their values up to functions, we evaluate functions from the top down, modifying what they will do with the rest of their data when they get it.

Alonzo Church and the λ-Calculus

The λ-Calculus

This is a powerful shift in how we think about performing a computation.

Rather than modifying variables and passing their values up to functions, we evaluate functions from the top down, modifying what they will do with the rest of their data when they get it.

The notation λxy is really a shorthand, which works like this:

Alonzo Church and the λ-Calculus

The λ-Calculus

This is a powerful shift in how we think about performing a computation.

Rather than modifying variables and passing their values up to functions, we evaluate functions from the top down, modifying what they will do with the rest of their data when they get it.

The notation λxy is really a shorthand, which works like this:

$$\lambda xy.x + y = \lambda x.(\lambda y.x + y).$$

Alonzo Church and the λ-Calculus

The λ-Calculus

This is a powerful shift in how we think about performing a computation.

Rather than modifying variables and passing their values up to functions, we evaluate functions from the top down, modifying what they will do with the rest of their data when they get it.

The notation λxy is really a shorthand, which works like this:

$$\lambda xy.x + y = \lambda x.(\lambda y.x + y).$$

Thus addition is a function on one variable x,

Alonzo Church and the λ-Calculus

The λ-Calculus

This is a powerful shift in how we think about performing a computation.

Rather than modifying variables and passing their values up to functions, we evaluate functions from the top down, modifying what they will do with the rest of their data when they get it.

The notation λxy is really a shorthand, which works like this:

$$\lambda xy.x + y = \lambda x.(\lambda y.x + y).$$

Thus addition is a function on one variable x, which, when given x, returns the function that adds that x to its input y.

Combinators

Combinators

Strings or Combinators

Combinators

Strings or Combinators

The purest form of the λ-calculus does not have arithmetic operations,

Combinators

Strings or Combinators

The purest form of the λ-calculus does not have arithmetic operations, or even the concept of integers.

Combinators

Strings or Combinators

The purest form of the λ-calculus does not have arithmetic operations, or even the concept of integers.

The basic objects in the λ-calculus are strings, or combinators,

Combinators

Strings or Combinators

The purest form of the λ-calculus does not have arithmetic operations, or even the concept of integers.

The basic objects in the λ-calculus are strings, or combinators, which take the next string to their right and substitute it for each appearance of the first variable marked with λ.

Combinators

Strings or Combinators

The purest form of the λ-calculus does not have arithmetic operations, or even the concept of integers.

The basic objects in the λ-calculus are strings, or combinators, which take the next string to their right and substitute it for each appearance of the first variable marked with λ.

In other words, if x is a variable, and U and V are strings,

Combinators

Strings or Combinators

The purest form of the λ-calculus does not have arithmetic operations, or even the concept of integers.

The basic objects in the λ-calculus are strings, or combinators, which take the next string to their right and substitute it for each appearance of the first variable marked with λ.

In other words, if x is a variable, and U and V are strings, then

$$(\lambda x.U)\,V = U[x \to V],$$

Combinators

Strings or Combinators

The purest form of the λ-calculus does not have arithmetic operations, or even the concept of integers.

The basic objects in the λ-calculus are strings, or combinators, which take the next string to their right and substitute it for each appearance of the first variable marked with λ.

In other words, if x is a variable, and U and V are strings, then

$$(\lambda x.U)\,V = U[x \to V],$$

where $U[x \to V]$ denotes the string we get from U by replacing each appearance of x with V.

Combinators

Strings or Combinators

The purest form of the λ-calculus does not have arithmetic operations, or even the concept of integers.

The basic objects in the λ-calculus are strings, or combinators, which take the next string to their right and substitute it for each appearance of the first variable marked with λ.

In other words, if x is a variable, and U and V are strings, then

$$(\lambda x.U)\,V = U[x \to V],$$

where $U[x \to V]$ denotes the string we get from U by replacing each appearance of x with V.

Each such step is called a reduction.

Combinators

Combinators

Strings or Combinators

Combinators

Strings or Combinators

A computation then consists of a chain of reductions, such as

Combinators

Strings or Combinators

A computation then consists of a chain of reductions, such as

$$(\lambda xy.yxy)ab$$

Combinators

Strings or Combinators

A computation then consists of a chain of reductions, such as

$$(\lambda xy.yxy)ab = (\lambda y.yay)b$$

Combinators

Strings or Combinators

A computation then consists of a chain of reductions, such as

$$(\lambda xy.yxy)ab = (\lambda y.yay)b = bab.$$

Combinators

Strings or Combinators

A computation then consists of a chain of reductions, such as

$$(\lambda xy.yxy)ab = (\lambda y.yay)b = bab.$$

We evaluate strings from left to right, and that in the absence of parentheses, we interpret the string zab as $(za)b$.

Combinators

Strings or Combinators

A computation then consists of a chain of reductions, such as

$$(\lambda xy.yxy)ab = (\lambda y.yay)b = bab.$$

We evaluate strings from left to right, and that in the absence of parentheses, we interpret the string zab as $(za)b$.

The operation of reduction is non-associative,

Combinators

Strings or Combinators

A computation then consists of a chain of reductions, such as

$$(\lambda xy.yxy)ab = (\lambda y.yay)b = bab.$$

We evaluate strings from left to right, and that in the absence of parentheses, we interpret the string zab as $(za)b$.

The operation of reduction is non-associative, meaning that $z(ab) \neq (za)b$ in general.

Combinators

Strings or Combinators

A computation then consists of a chain of reductions, such as

$$(\lambda xy.yxy)ab = (\lambda y.yay)b = bab.$$

We evaluate strings from left to right, and that in the absence of parentheses, we interpret the string zab as $(za)b$.

The operation of reduction is non-associative, meaning that $z(ab) \neq (za)b$ in general.

For example, grouping a and b together in this example gives

$$(\lambda xy.yxy)(ab)$$

Combinators

Strings or Combinators

A computation then consists of a chain of reductions, such as

$$(\lambda xy.yxy)ab = (\lambda y.yay)b = bab.$$

We evaluate strings from left to right, and that in the absence of parentheses, we interpret the string zab as $(za)b$.

The operation of reduction is non-associative, meaning that $z(ab) \neq (za)b$ in general.

For example, grouping a and b together in this example gives

$$(\lambda xy.yxy)(ab) = \lambda y.y(ab)y.$$

Combinators

Combinators

Feeding one combinator to another

Combinators

Feeding one combinator to another

Assume that $T = \lambda xy.x$, $F = \lambda xy.y$, and $I = \lambda x.x$.

Combinators

Feeding one combinator to another

Assume that $T = \lambda xy.x$, $F = \lambda xy.y$, and $I = \lambda x.x$.

The combinators T and F take two inputs and return the first or second one.

Combinators

Feeding one combinator to another

Assume that $T = \lambda xy.x$, $F = \lambda xy.y$, and $I = \lambda x.x$.

The combinators T and F take two inputs and return the first or second one.

Since I is the identity, $IT = T$ and $IF = F$.

Combinators

Feeding one combinator to another

Assume that $T = \lambda xy.x$, $F = \lambda xy.y$, and $I = \lambda x.x$.

The combinators T and F take two inputs and return the first or second one.

Since I is the identity, $IT = T$ and $IF = F$.

Now, we have

$$TI \quad = \quad (\lambda xy.x)(\lambda x.x)$$

Combinators

Feeding one combinator to another

Assume that $T = \lambda xy.x$, $F = \lambda xy.y$, and $I = \lambda x.x$.

The combinators T and F take two inputs and return the first or second one.

Since I is the identity, $IT = T$ and $IF = F$.

Now, we have

$$\begin{aligned} TI &= (\lambda xy.x)(\lambda x.x) \\ &= (\lambda xy.x)(\lambda z.z) \end{aligned}$$

Combinators

Feeding one combinator to another

Assume that $T = \lambda xy.x$, $F = \lambda xy.y$, and $I = \lambda x.x$.

The combinators T and F take two inputs and return the first or second one.

Since I is the identity, $IT = T$ and $IF = F$.

Now, we have

$$\begin{aligned} TI &= (\lambda xy.x)(\lambda x.x) \\ &= (\lambda xy.x)(\lambda z.z) \\ &= \lambda y.(\lambda z.z) \end{aligned}$$

Combinators

Feeding one combinator to another

Assume that $T = \lambda xy.x$, $F = \lambda xy.y$, and $I = \lambda x.x$.

The combinators T and F take two inputs and return the first or second one.

Since I is the identity, $IT = T$ and $IF = F$.

Now, we have

$$\begin{aligned} TI &= (\lambda xy.x)(\lambda x.x) \\ &= (\lambda xy.x)(\lambda z.z) \\ &= \lambda y.(\lambda z.z) \\ &= \lambda yz.z \end{aligned}$$

Combinators

Feeding one combinator to another

Assume that $T = \lambda xy.x$, $F = \lambda xy.y$, and $I = \lambda x.x$.

The combinators T and F take two inputs and return the first or second one.

Since I is the identity, $IT = T$ and $IF = F$.

Now, we have

$$\begin{aligned} TI &= (\lambda xy.x)(\lambda x.x) \\ &= (\lambda xy.x)(\lambda z.z) \\ &= \lambda y.(\lambda z.z) \\ &= \lambda yz.z \\ &= F. \end{aligned}$$

Combinators

Combinators

Feeding one combinator to another

Combinators

Feeding one combinator to another

Some products, such as TI, reach a final state, or normal form, in which no more reductions are possible.

Combinators

Feeding one combinator to another

Some products, such as *TI*, reach a final state, or normal form, in which no more reductions are possible.

Either there are no λs left, or there are no strings to the right of the first combinator to plug into it.

Combinators

Feeding one combinator to another

Some products, such as *TI*, reach a final state, or normal form, in which no more reductions are possible.

Either there are no λs left, or there are no strings to the right of the first combinator to plug into it.

However, this is not true for all expressions.

Combinators

Feeding one combinator to another

Some products, such as TI, reach a final state, or normal form, in which no more reductions are possible.

Either there are no λs left, or there are no strings to the right of the first combinator to plug into it.

However, this is not true for all expressions.

Consider $Q = \lambda x.xx$, which takes x as input and returns xx.

Combinators

Feeding one combinator to another

Some products, such as TI, reach a final state, or normal form, in which no more reductions are possible.

Either there are no λs left, or there are no strings to the right of the first combinator to plug into it.

However, this is not true for all expressions.

Consider $Q = \lambda x.xx$, which takes x as input and returns xx.

If we feed Q to itself, we get

Combinators

Feeding one combinator to another

Some products, such as TI, reach a final state, or normal form, in which no more reductions are possible.

Either there are no λs left, or there are no strings to the right of the first combinator to plug into it.

However, this is not true for all expressions.

Consider $Q = \lambda x.xx$, which takes x as input and returns xx.

If we feed Q to itself, we get

$$QQ = (\lambda x.xx)(\lambda x.xx)$$

Combinators

Feeding one combinator to another

Some products, such as TI, reach a final state, or normal form, in which no more reductions are possible.

Either there are no λs left, or there are no strings to the right of the first combinator to plug into it.

However, this is not true for all expressions.

Consider $Q = \lambda x.xx$, which takes x as input and returns xx.

If we feed Q to itself, we get

$$QQ = (\lambda x.xx)(\lambda x.xx) \quad = \quad (\lambda x.xx)(\lambda x.xx)$$

Combinators

Feeding one combinator to another

Some products, such as TI, reach a final state, or normal form, in which no more reductions are possible.

Either there are no λs left, or there are no strings to the right of the first combinator to plug into it.

However, this is not true for all expressions.

Consider $Q = \lambda x.xx$, which takes x as input and returns xx.

If we feed Q to itself, we get

$$\begin{aligned} QQ = (\lambda x.xx)(\lambda x.xx) &= (\lambda x.xx)(\lambda x.xx) \\ &= (\lambda x.xx)(\lambda x.xx) \end{aligned}$$

Combinators

Feeding one combinator to another

Some products, such as TI, reach a final state, or normal form, in which no more reductions are possible.

Either there are no λs left, or there are no strings to the right of the first combinator to plug into it.

However, this is not true for all expressions.

Consider $Q = \lambda x.xx$, which takes x as input and returns xx.

If we feed Q to itself, we get

$$\begin{aligned} QQ = (\lambda x.xx)(\lambda x.xx) &= (\lambda x.xx)(\lambda x.xx) \\ &= (\lambda x.xx)(\lambda x.xx) \\ &= \ldots \end{aligned}$$

Combinators

Feeding one combinator to another

Some products, such as TI, reach a final state, or normal form, in which no more reductions are possible.

Either there are no λs left, or there are no strings to the right of the first combinator to plug into it.

However, this is not true for all expressions.

Consider $Q = \lambda x.xx$, which takes x as input and returns xx.

If we feed Q to itself, we get

$$\begin{aligned} QQ = (\lambda x.xx)(\lambda x.xx) &= (\lambda x.xx)(\lambda x.xx) \\ &= (\lambda x.xx)(\lambda x.xx) \\ &= \ldots \end{aligned}$$

The chain of the reductions never halts.

Combinators

Combinators

Feeding one combinator to another

Combinators

Feeding one combinator to another

Even worse is the following expression, which grows without bounds as we try to reduce it:

Combinators

Feeding one combinator to another

Even worse is the following expression, which grows without bounds as we try to reduce it:

$$(\lambda x.xx)(\lambda y.yyy) \quad = \quad (\lambda y.yyy)(\lambda y.yyy)$$

Combinators

Feeding one combinator to another

Even worse is the following expression, which grows without bounds as we try to reduce it:

$$\begin{aligned}(\lambda x.xx)(\lambda y.yyy) &= (\lambda y.yyy)(\lambda y.yyy) \\ &= (\lambda y.yyy)(\lambda y.yyy)(\lambda y.yyy)\end{aligned}$$

Combinators

Feeding one combinator to another

Even worse is the following expression, which grows without bounds as we try to reduce it:

$$\begin{aligned}(\lambda x.xx)(\lambda y.yyy) &= (\lambda y.yyy)(\lambda y.yyy) \\ &= (\lambda y.yyy)(\lambda y.yyy)(\lambda y.yyy) \\ &= (\lambda y.yyy)(\lambda y.yyy)(\lambda y.yyy)(\lambda y.yyy)\end{aligned}$$

Combinators

Feeding one combinator to another

Even worse is the following expression, which grows without bounds as we try to reduce it:

$$\begin{aligned}(\lambda x.xx)(\lambda y.yyy) &= (\lambda y.yyy)(\lambda y.yyy) \\ &= (\lambda y.yyy)(\lambda y.yyy)(\lambda y.yyy) \\ &= (\lambda y.yyy)(\lambda y.yyy)(\lambda y.yyy)(\lambda y.yyy) \\ &= \ldots\end{aligned}$$

Combinators

Feeding one combinator to another

Even worse is the following expression, which grows without bounds as we try to reduce it:

$$\begin{aligned}(\lambda x.xx)(\lambda y.yyy) &= (\lambda y.yyy)(\lambda y.yyy) \\ &= (\lambda y.yyy)(\lambda y.yyy)(\lambda y.yyy) \\ &= (\lambda y.yyy)(\lambda y.yyy)(\lambda y.yyy)(\lambda y.yyy) \\ &= \ldots\end{aligned}$$

These are examples of infinite loops in the λ-calculus.

Combinators

Feeding one combinator to another

Even worse is the following expression, which grows without bounds as we try to reduce it:

$$\begin{aligned}(\lambda x.xx)(\lambda y.yyy) &= (\lambda y.yyy)(\lambda y.yyy) \\ &= (\lambda y.yyy)(\lambda y.yyy)(\lambda y.yyy) \\ &= (\lambda y.yyy)(\lambda y.yyy)(\lambda y.yyy)(\lambda y.yyy) \\ &= \ldots\end{aligned}$$

These are examples of infinite loops in the λ-calculus.

Like partial recursive functions, λ-expressions may never return an answer, and the functions they compute may be undefined for some values of their input.

Combinators

Combinators

Feeding one combinator to another: Boolean expressions

Combinators

Feeding one combinator to another: Boolean expressions

Consider the combinators T and F defined above.

Combinators

Feeding one combinator to another: Boolean expressions

Consider the combinators T and F defined above.

We will interpret them as *true* and *false*, respectively.

Combinators

Feeding one combinator to another: Boolean expressions

Consider the combinators T and F defined above.

We will interpret them as *true* and *false*, respectively.

Show that we can represent the well-known Boolean functions as follows:

Combinators

Feeding one combinator to another: Boolean expressions

Consider the combinators T and F defined above.

We will interpret them as *true* and *false*, respectively.

Show that we can represent the well-known Boolean functions as follows:

$$not = \lambda pxy.pyx,$$

Combinators

Feeding one combinator to another: Boolean expressions

Consider the combinators T and F defined above.

We will interpret them as *true* and *false*, respectively.

Show that we can represent the well-known Boolean functions as follows:

$$not = \lambda pxy.pyx,$$
$$and = \lambda pq.pqp,$$

Combinators

Feeding one combinator to another: Boolean expressions

Consider the combinators T and F defined above.

We will interpret them as *true* and *false*, respectively.

Show that we can represent the well-known Boolean functions as follows:

$$not = \lambda pxy.pyx,$$
$$and = \lambda pq.pqp,$$
$$or = \lambda pq.ppq.$$

Combinators

Feeding one combinator to another: Boolean expressions

Consider the combinators T and F defined above.

We will interpret them as *true* and *false*, respectively.

Show that we can represent the well-known Boolean functions as follows:

$$not = \lambda pxy.pyx,$$
$$and = \lambda pq.pqp,$$
$$or = \lambda pq.ppq.$$

One can also show that

$$andTF = (andT)F = IF = F.$$

Combinators

Combinators

How about arithmetic?

Combinators

How about arithmetic?

One can represent the natural numbers $0, 1, 2, 3, \ldots$ in λ-calculus.

Combinators

How about arithmetic?

One can represent the natural numbers $0, 1, 2, 3, \ldots$ in λ-calculus.

We will use the Church numerals, defined as follows:

$$\bar{0} \quad = \quad \lambda fx.x,$$

Combinators

How about arithmetic?

One can represent the natural numbers $0, 1, 2, 3, \ldots$ in λ-calculus.

We will use the Church numerals, defined as follows:

$$\begin{aligned} \bar{0} &= \lambda fx.x, \\ \bar{1} &= \lambda fx.fx, \end{aligned}$$

Combinators

How about arithmetic?

One can represent the natural numbers $0, 1, 2, 3, \ldots$ in λ-calculus.

We will use the Church numerals, defined as follows:

$$\begin{aligned} \bar{0} &= \lambda fx.x, \\ \bar{1} &= \lambda fx.fx, \\ \bar{2} &= \lambda fx.f(fx), \end{aligned}$$

Combinators

How about arithmetic?

One can represent the natural numbers $0, 1, 2, 3, \ldots$ in λ-calculus.

We will use the Church numerals, defined as follows:

$$\begin{aligned} \bar{0} &= \lambda fx.x, \\ \bar{1} &= \lambda fx.fx, \\ \bar{2} &= \lambda fx.f(fx), \\ \bar{3} &= \lambda fx.f(f(fx)). \end{aligned}$$

Combinators

How about arithmetic?

One can represent the natural numbers $0, 1, 2, 3, \ldots$ in λ-calculus.

We will use the Church numerals, defined as follows:

$$\begin{aligned} \bar{0} &= \lambda fx.x, \\ \bar{1} &= \lambda fx.fx, \\ \bar{2} &= \lambda fx.f(fx), \\ \bar{3} &= \lambda fx.f(f(fx)). \end{aligned}$$

Given a function f, the Church numeral $\bar{n}$ returns the function f^n, that is, f iterated n times:

Combinators

How about arithmetic?

One can represent the natural numbers $0, 1, 2, 3, \ldots$ in λ-calculus.

We will use the Church numerals, defined as follows:

$$\begin{aligned} \bar{0} &= \lambda fx.x, \\ \bar{1} &= \lambda fx.fx, \\ \bar{2} &= \lambda fx.f(fx), \\ \bar{3} &= \lambda fx.f(f(fx)). \end{aligned}$$

Given a function f, the Church numeral $\bar{n}$ returns the function f^n, that is, f iterated n times:

$$\bar{n}f = \lambda x.f^n x.$$

Combinators

Combinators

How about arithmetic?

Combinators

How about arithmetic?

One can define combinators that carry out arithmetic operations on these numerals.

Combinators

How about arithmetic?

One can define combinators that carry out arithmetic operations on these numerals.

$$succ = \lambda mfx.f(mfx)$$

Combinators

How about arithmetic?

One can define combinators that carry out arithmetic operations on these numerals.

$$succ = \lambda mfx.f(mfx)$$

If $\bar{m}$ is a numeral, $succ\bar{m}$ applies f one more time than $\bar{m}$ would.

Combinators

How about arithmetic?

One can define combinators that carry out arithmetic operations on these numerals.

$$succ = \lambda mfx.f(mfx)$$

If $\bar{m}$ is a numeral, $succ\bar{m}$ applies f one more time than $\bar{m}$ would. Thus $succ$ acts as the successor function.

Combinators

How about arithmetic?

One can define combinators that carry out arithmetic operations on these numerals.

$$succ = \lambda mfx.f(mfx)$$

If $\bar{m}$ is a numeral, $succ\bar{m}$ applies f one more time than $\bar{m}$ would. Thus $succ$ acts as the successor function.

Let's check this, reducing one step at a time:

$$succ\bar{n} \quad = \quad (\lambda mfx.f(mfx))(\lambda gy.g^n y)$$

Combinators

How about arithmetic?

One can define combinators that carry out arithmetic operations on these numerals.

$$succ = \lambda mfx.f(mfx)$$

If $\bar{m}$ is a numeral, $succ\bar{m}$ applies f one more time than $\bar{m}$ would. Thus $succ$ acts as the successor function.

Let's check this, reducing one step at a time:

$$\begin{aligned} succ\bar{n} &= (\lambda mfx.f(mfx))(\lambda gy.g^n y) \\ &= \lambda fx.f((\lambda gy.g^n y)fx) \end{aligned}$$

Combinators

How about arithmetic?

One can define combinators that carry out arithmetic operations on these numerals.

$$succ = \lambda mfx.f(mfx)$$

If $\bar{m}$ is a numeral, $succ\bar{m}$ applies f one more time than $\bar{m}$ would. Thus $succ$ acts as the successor function.

Let's check this, reducing one step at a time:

$$\begin{aligned} succ\bar{n} &= (\lambda mfx.f(mfx))(\lambda gy.g^n y) \\ &= \lambda fx.f((\lambda gy.g^n y)fx) \\ &= \lambda fx.f((\lambda y.f^n y)x) \end{aligned}$$

Combinators

How about arithmetic?

One can define combinators that carry out arithmetic operations on these numerals.

$$succ = \lambda mfx.f(mfx)$$

If $\bar{m}$ is a numeral, $succ\bar{m}$ applies f one more time than $\bar{m}$ would. Thus $succ$ acts as the successor function.

Let's check this, reducing one step at a time:

$$\begin{aligned} succ\bar{n} &= (\lambda mfx.f(mfx))(\lambda gy.g^n y) \\ &= \lambda fx.f((\lambda gy.g^n y)fx) \\ &= \lambda fx.f((\lambda y.f^n y)x) \\ &= \lambda fx.f(f^n x) \end{aligned}$$

Combinators

How about arithmetic?

One can define combinators that carry out arithmetic operations on these numerals.

$$succ = \lambda mfx.f(mfx)$$

If $\bar{m}$ is a numeral, $succ\bar{m}$ applies f one more time than $\bar{m}$ would. Thus $succ$ acts as the successor function.

Let's check this, reducing one step at a time:

$$\begin{aligned} succ\bar{n} &= (\lambda mfx.f(mfx))(\lambda gy.g^n y) \\ &= \lambda fx.f((\lambda gy.g^n y)fx) \\ &= \lambda fx.f((\lambda y.f^n y)x) \\ &= \lambda fx.f(f^n x) \\ &= \lambda fx.f^{n+1}x \end{aligned}$$

Combinators

How about arithmetic?

One can define combinators that carry out arithmetic operations on these numerals.

$$succ = \lambda mfx.f(mfx)$$

If $\bar{m}$ is a numeral, $succ\bar{m}$ applies f one more time than $\bar{m}$ would. Thus $succ$ acts as the successor function.

Let's check this, reducing one step at a time:

$$\begin{aligned} succ\bar{n} &= (\lambda mfx.f(mfx))(\lambda gy.g^n y) \\ &= \lambda fx.f((\lambda gy.g^n y)fx) \\ &= \lambda fx.f((\lambda y.f^n y)x) \\ &= \lambda fx.f(f^n x) \\ &= \lambda fx.f^{n+1}x \\ &= \overline{n+1}. \end{aligned}$$

Combinators

Combinators

How about arithmetic? The addition

Combinators

How about arithmetic? The addition

Combinators allow us to write inductive definitions of functions very compactly.

Combinators

How about arithmetic? The addition

Combinators allow us to write inductive definitions of functions very compactly.

For instance, suppose we want a combinator *add* that adds two numerals together.

Combinators

How about arithmetic? The addition

Combinators allow us to write inductive definitions of functions very compactly.

For instance, suppose we want a combinator *add* that adds two numerals together. One way to do this is:

$$add = \lambda nmfx.nf(mfx).$$

Combinators

How about arithmetic? The addition

Combinators allow us to write inductive definitions of functions very compactly.

For instance, suppose we want a combinator *add* that adds two numerals together. One way to do this is:

$$add = \lambda nmfx.nf(mfx).$$

Since $f^m(f^n(x)) = f^{m+n}(x)$, we have

Combinators

How about arithmetic? The addition

Combinators allow us to write inductive definitions of functions very compactly.

For instance, suppose we want a combinator *add* that adds two numerals together. One way to do this is:

$$add = \lambda nmfx.nf(mfx).$$

Since $f^m(f^n(x)) = f^{m+n}(x)$, we have

$$add\bar{m}\bar{n} = \overline{m+n}.$$

Combinators

How about arithmetic? The addition

Combinators allow us to write inductive definitions of functions very compactly.

For instance, suppose we want a combinator *add* that adds two numerals together. One way to do this is:

$$add = \lambda nmfx.nf(mfx).$$

Since $f^m(f^n(x)) = f^{m+n}(x)$, we have

$$add\bar{m}\bar{n} = \overline{m+n}.$$

Alternatively, we can describe adding n as iterating the successor function n times.

Combinators

How about arithmetic? The addition

Combinators allow us to write inductive definitions of functions very compactly.

For instance, suppose we want a combinator *add* that adds two numerals together. One way to do this is:

$$add = \lambda nmfx.nf(mfx).$$

Since $f^m(f^n(x)) = f^{m+n}(x)$, we have

$$add\bar{m}\bar{n} = \overline{m+n}.$$

Alternatively, we can describe adding n as iterating the successor function n times. We could have written:

$$add' = \lambda n.nsucc.$$

Combinators

How about arithmetic? The addition

Combinators allow us to write inductive definitions of functions very compactly.

For instance, suppose we want a combinator *add* that adds two numerals together. One way to do this is:

$$add = \lambda nmfx.nf(mfx).$$

Since $f^m(f^n(x)) = f^{m+n}(x)$, we have

$$add\bar{m}\bar{n} = \overline{m+n}.$$

Alternatively, we can describe adding n as iterating the successor function n times. We could have written:

$$add' = \lambda n.nsucc.$$

One can show that *add* and add' are equivalent, when applied to Church numerals.

Combinators

Combinators

How about arithmetic? More arithmetic combinators

Combinators

How about arithmetic? More arithmetic combinators

For multiplication, we use the fact that if we iterate f^n m times, we get $(f^n)^m = f^{n \cdot m}$.

Combinators

How about arithmetic? More arithmetic combinators

For multiplication, we use the fact that if we iterate f^n m times, we get $(f^n)^m = f^{n \cdot m}$. Thus if

$$mult = \lambda nmf.n(mf),$$

Combinators

How about arithmetic? More arithmetic combinators

For multiplication, we use the fact that if we iterate f^n m times, we get $(f^n)^m = f^{n \cdot m}$. Thus if

$$mult = \lambda nmf.n(mf),$$

we have

$$mult\bar{m}\bar{n} = \overline{m \cdot n}.$$

Combinators

How about arithmetic? More arithmetic combinators

For multiplication, we use the fact that if we iterate f^n m times, we get $(f^n)^m = f^{n \cdot m}$. Thus if

$$mult = \lambda nmf.n(mf),$$

we have

$$mult\bar{m}\bar{n} = \overline{m \cdot n}.$$

One can show that if

$$exp = \lambda nm.mn,$$

Combinators

How about arithmetic? More arithmetic combinators

For multiplication, we use the fact that if we iterate f^n m times, we get $(f^n)^m = f^{n \cdot m}$. Thus if

$$mult = \lambda nmf.n(mf),$$

we have

$$mult\bar{m}\bar{n} = \overline{m \cdot n}.$$

One can show that if

$$exp = \lambda nm.mn,$$

then

$$exp\bar{n}\bar{m} = \overline{m^n}.$$

The power of λ

The power of λ

λ-definable functions

The power of λ

λ-definable functions

A function $f : N \longrightarrow N$ is λ-definable,

The power of λ

λ-definable functions

A function $f : N \longrightarrow N$ is λ-definable, if there is a combinator Φ, such that

$$\Phi\bar{n} = \overline{f(n)},$$

The power of λ

λ-definable functions

A function $f : N \longrightarrow N$ is λ-definable, if there is a combinator Φ, such that

$$\Phi\bar{n} = \overline{f(n)},$$

for any Church numeral $\bar{n}$, as long as $f(n)$ is defined.

The power of λ

λ-definable functions

A function $f : N \longrightarrow N$ is λ-definable, if there is a combinator Φ, such that

$$\Phi\bar{n} = \overline{f(n)},$$

for any Church numeral $\bar{n}$, as long as $f(n)$ is defined.

Can we characterize the class of λ-definable functions?

The power of λ

λ-definable functions

A function $f : N \longrightarrow N$ is λ-definable, if there is a combinator Φ, such that

$$\Phi\bar{n} = \overline{f(n)},$$

for any Church numeral $\bar{n}$, as long as $f(n)$ is defined.

Can we characterize the class of λ-definable functions?

How expressive is the λ-calculus compared to the recursive functions?

The power of λ

λ-definable functions

A function $f : N \longrightarrow N$ is λ-definable, if there is a combinator Φ, such that

$$\Phi\bar{n} = \overline{f(n)},$$

for any Church numeral $\bar{n}$, as long as $f(n)$ is defined.

Can we characterize the class of λ-definable functions?

How expressive is the λ-calculus compared to the recursive functions?

Are all partial recursive functions λ-definable, or vice versa?

The power of λ

λ-definable functions

A function $f : N \longrightarrow N$ is λ-definable, if there is a combinator Φ, such that

$$\Phi\bar{n} = \overline{f(n)},$$

for any Church numeral $\bar{n}$, as long as $f(n)$ is defined.

Can we characterize the class of λ-definable functions?

How expressive is the λ-calculus compared to the recursive functions?

Are all partial recursive functions λ-definable, or vice versa?

How one can prove that the two classes coincide?

The power of λ

The power of λ

λ-definable functions

The power of λ

λ-definable functions

Recall that the partial recursive functions are defined inductively from zero and the successor function using composition, primitive recursion and μ-recursion.

The power of λ

λ-definable functions

Recall that the partial recursive functions are defined inductively from zero and the successor function using composition, primitive recursion and μ-recursion.

We already have zero and the successor.

The power of λ

λ-definable functions

Recall that the partial recursive functions are defined inductively from zero and the successor function using composition, primitive recursion and μ-recursion.

We already have zero and the successor.

We can compose two functions with the following combinator,

$$comp = \lambda fgx.f(gx),$$

The power of λ

λ-definable functions

Recall that the partial recursive functions are defined inductively from zero and the successor function using composition, primitive recursion and μ-recursion.

We already have zero and the successor.

We can compose two functions with the following combinator,

$$comp = \lambda fgx.f(gx),$$

which takes two functions f, g as input and returns $f \circ g$.

The power of λ

λ-definable functions

Recall that the partial recursive functions are defined inductively from zero and the successor function using composition, primitive recursion and μ-recursion.

We already have zero and the successor.

We can compose two functions with the following combinator,

$$comp = \lambda fgx.f(gx),$$

which takes two functions f, g as input and returns $f \circ g$.

Similarly, one can handle primitive recursion by treating it as a *for* loop.

The power of λ

The power of λ

Checking conditions

The power of λ

Checking conditions

One can show that if

$$iszero = \lambda n.n(\lambda x.F)T,$$

The power of λ

Checking conditions

One can show that if

$$iszero = \lambda n.n(\lambda x.F)T,$$

where F and T are defined above,

The power of λ

Checking conditions

One can show that if

$$iszero = \lambda n.n(\lambda x.F)T,$$

where F and T are defined above, then

$$iszero\bar{n}xy = \begin{cases} x, & \text{if } n = 0 \\ y, & \text{if } n > 0. \end{cases}$$

The power of λ

The power of λ

Modeling the μ-recursion

The power of λ

Modeling the μ-recursion

Recall that if $f(x, y)$ is a partial recursive function,

The power of λ

Modeling the μ-recursion

Recall that if $f(x, y)$ is a partial recursive function, we can define a function $h(x)$ as the smallest y such that $f(x, y) = 0$:

The power of λ

Modeling the μ-recursion

Recall that if $f(x, y)$ is a partial recursive function, we can define a function $h(x)$ as the smallest y such that $f(x, y) = 0$:

$$h(x) = \mu_y f(x, y)$$

The power of λ

Modeling the μ-recursion

Recall that if $f(x, y)$ is a partial recursive function, we can define a function $h(x)$ as the smallest y such that $f(x, y) = 0$:

$$h(x) = \mu_y f(x, y) = \min\{y : f(x, y) = 0\},$$

The power of λ

Modeling the μ-recursion

Recall that if $f(x, y)$ is a partial recursive function, we can define a function $h(x)$ as the smallest y such that $f(x, y) = 0$:

$$h(x) = \mu_y f(x, y) = \min\{y : f(x, y) = 0\},$$

assuming that $f(x, y)$ is defined for all $y \leq h(x)$.

The power of λ

Modeling the μ-recursion

Recall that if $f(x, y)$ is a partial recursive function, we can define a function $h(x)$ as the smallest y such that $f(x, y) = 0$:

$$h(x) = \mu_y f(x, y) = \min\{y : f(x, y) = 0\},$$

assuming that $f(x, y)$ is defined for all $y \leq h(x)$.

First let Φ be a function which takes a pair (x, y) as input.

The power of λ

Modeling the μ-recursion

Recall that if $f(x, y)$ is a partial recursive function, we can define a function $h(x)$ as the smallest y such that $f(x, y) = 0$:

$$h(x) = \mu_y f(x, y) = \min\{y : f(x, y) = 0\},$$

assuming that $f(x, y)$ is defined for all $y \leq h(x)$.

First let Φ be a function which takes a pair (x, y) as input. It explores larger and larger values of y until it finds one such that $f(x, y) = 0$, which is its return:

The power of λ

Modeling the μ-recursion

Recall that if $f(x, y)$ is a partial recursive function, we can define a function $h(x)$ as the smallest y such that $f(x, y) = 0$:

$$h(x) = \mu_y f(x, y) = \min\{y : f(x, y) = 0\},$$

assuming that $f(x, y)$ is defined for all $y \leq h(x)$.

First let Φ be a function which takes a pair (x, y) as input. It explores larger and larger values of y until it finds one such that $f(x, y) = 0$, which is its return:

$$\Phi(x, y) = \begin{cases} y, & \text{if } f(x, y) = 0 \\ \Phi(x, y + 1), & \text{otherwise.} \end{cases}$$

The power of λ

Modeling the μ-recursion

Recall that if $f(x, y)$ is a partial recursive function, we can define a function $h(x)$ as the smallest y such that $f(x, y) = 0$:

$$h(x) = \mu_y f(x, y) = \min\{y : f(x, y) = 0\},$$

assuming that $f(x, y)$ is defined for all $y \leq h(x)$.

First let Φ be a function which takes a pair (x, y) as input. It explores larger and larger values of y until it finds one such that $f(x, y) = 0$, which is its return:

$$\Phi(x, y) = \begin{cases} y, & \text{if } f(x, y) = 0 \\ \Phi(x, y + 1), & \text{otherwise.} \end{cases}$$

Then we can write $h(x) = \Phi(x, 0)$.

The power of λ

The power of λ

Modeling the μ-recursion

The power of λ

Modeling the μ-recursion

It can be checked that

$$h = \lambda x.\Phi x\bar{0}, \text{ where } \Phi = YH,$$

The power of λ

Modeling the μ-recursion

It can be checked that

$$h = \lambda x.\Phi x\bar{0}, \text{ where } \Phi = YH,$$

and

$$Y = \lambda R.(\lambda x.R(xx))(\lambda x.R(xx)),$$

The power of λ

Modeling the μ-recursion

It can be checked that

$$h = \lambda x.\Phi x\bar{0}, \text{ where } \Phi = YH,$$

and

$$Y = \lambda R.(\lambda x.R(xx))(\lambda x.R(xx)), H = \lambda \Phi xy.iszero(fxy)y(\Phi x(succy)).$$

The power of λ

Modeling the μ-recursion

It can be checked that

$$h = \lambda x.\Phi x\bar{0}, \text{ where } \Phi = YH,$$

and

$$Y = \lambda R.(\lambda x.R(xx))(\lambda x.R(xx)), H = \lambda \Phi xy.iszero(fxy)y(\Phi x(succy)).$$

This shows that all partial recursive functions are λ-definable.

The power of λ

Modeling the μ-recursion

It can be checked that

$$h = \lambda x.\Phi x\bar{0}, \text{ where } \Phi = YH,$$

and

$$Y = \lambda R.(\lambda x.R(xx))(\lambda x.R(xx)), H = \lambda \Phi xy.iszero(fxy)y(\Phi x(succy)).$$

This shows that all partial recursive functions are λ-definable.

One can also show that the converse is also true.

The power of λ

Modeling the μ-recursion

It can be checked that

$$h = \lambda x.\Phi x\bar{0}, \text{ where } \Phi = YH,$$

and

$$Y = \lambda R.(\lambda x.R(xx))(\lambda x.R(xx)), H = \lambda \Phi xy.iszero(fxy)y(\Phi x(succy)).$$

This shows that all partial recursive functions are λ-definable.

One can also show that the converse is also true.

Thus the class of partial recursive functions coincides with that of λ-definable functions.

Turing machines

Turing machines

Definition of a Turing machine

Turing machines

Definition of a Turing machine

A Turing machine has a "head", which can be in finite number of internal states,

Turing machines

Definition of a Turing machine

A Turing machine has a "head", which can be in finite number of internal states, and a "tape", where each square contains a symbol drawn from a finite alphabet.

Turing machines

Definition of a Turing machine

A Turing machine has a "head", which can be in finite number of internal states, and a "tape", where each square contains a symbol drawn from a finite alphabet.

In each step of computation, it observes the symbol at its current location on the tape.

Turing machines

Definition of a Turing machine

A Turing machine has a "head", which can be in finite number of internal states, and a "tape", where each square contains a symbol drawn from a finite alphabet.

In each step of computation, it observes the symbol at its current location on the tape.

Based on this symbol and its internal state, it then

Turing machines

Definition of a Turing machine

A Turing machine has a "head", which can be in finite number of internal states, and a "tape", where each square contains a symbol drawn from a finite alphabet.

In each step of computation, it observes the symbol at its current location on the tape.

Based on this symbol and its internal state, it then

- updates the symbol at its current location on the tape,

Turing machines

Definition of a Turing machine

A Turing machine has a "head", which can be in finite number of internal states, and a "tape", where each square contains a symbol drawn from a finite alphabet.

In each step of computation, it observes the symbol at its current location on the tape.

Based on this symbol and its internal state, it then

- updates the symbol at its current location on the tape,
- updates its internal state, and

Turing machines

Definition of a Turing machine

A Turing machine has a "head", which can be in finite number of internal states, and a "tape", where each square contains a symbol drawn from a finite alphabet.

In each step of computation, it observes the symbol at its current location on the tape.

Based on this symbol and its internal state, it then

- updates the symbol at its current location on the tape,
- updates its internal state, and
- moves one step left or right on the tape.

Turing machines

Turing machines

Definition of a Turing machine

Turing machines

Definition of a Turing machine

If A is the set of symbols that can appear on the tape,

Turing machines

Definition of a Turing machine

If A is the set of symbols that can appear on the tape, and S is the set of internal states,

Turing machines

Definition of a Turing machine

If A is the set of symbols that can appear on the tape, and S is the set of internal states, we can write the machine's behavior as a transition function:

$$F : A \times S \to A \times S \times \{\pm 1\}.$$

Turing machines

Definition of a Turing machine

If A is the set of symbols that can appear on the tape, and S is the set of internal states, we can write the machine's behavior as a transition function:

$$F : A \times S \to A \times S \times \{\pm 1\}.$$

For instance, $F(a, s) = (a', s', +1)$ would mean that if the machine sees the symbol a on the tape when it is in state s,

Turing machines

Definition of a Turing machine

If A is the set of symbols that can appear on the tape, and S is the set of internal states, we can write the machine's behavior as a transition function:

$$F : A \times S \to A \times S \times \{\pm 1\}.$$

For instance, $F(a, s) = (a', s', +1)$ would mean that if the machine sees the symbol a on the tape when it is in state s, then it changes the tape symbol to a',

Turing machines

Definition of a Turing machine

If A is the set of symbols that can appear on the tape, and S is the set of internal states, we can write the machine's behavior as a transition function:

$$F : A \times S \to A \times S \times \{\pm 1\}.$$

For instance, $F(a, s) = (a', s', +1)$ would mean that if the machine sees the symbol a on the tape when it is in state s, then it changes the tape symbol to a', changes its state to s',

Turing machines

Definition of a Turing machine

If A is the set of symbols that can appear on the tape, and S is the set of internal states, we can write the machine's behavior as a transition function:

$$F : A \times S \to A \times S \times \{\pm 1\}.$$

For instance, $F(a, s) = (a', s', +1)$ would mean that if the machine sees the symbol a on the tape when it is in state s, then it changes the tape symbol to a', changes its state to s', and moves to the right.

Turing machines

Definition of a Turing machine

If A is the set of symbols that can appear on the tape, and S is the set of internal states, we can write the machine's behavior as a transition function:

$$F : A \times S \to A \times S \times \{\pm 1\}.$$

For instance, $F(a, s) = (a', s', +1)$ would mean that if the machine sees the symbol a on the tape when it is in state s, then it changes the tape symbol to a', changes its state to s', and moves to the right.

We start the machine by writing the input string x on the tape, with a special blank symbol Λ written on the rest of the tape stretching left and right to infinity.

Turing machines

Definition of a Turing machine

If A is the set of symbols that can appear on the tape, and S is the set of internal states, we can write the machine's behavior as a transition function:

$$F : A \times S \to A \times S \times \{\pm 1\}.$$

For instance, $F(a, s) = (a', s', +1)$ would mean that if the machine sees the symbol a on the tape when it is in state s, then it changes the tape symbol to a', changes its state to s', and moves to the right.

We start the machine by writing the input string x on the tape, with a special blank symbol Λ written on the rest of the tape stretching left and right to infinity.

We place the head at one end of the input, and start it in some specific initial state.

Turing machines

Definition of a Turing machine

If A is the set of symbols that can appear on the tape, and S is the set of internal states, we can write the machine's behavior as a transition function:

$$F : A \times S \to A \times S \times \{\pm 1\}.$$

For instance, $F(a, s) = (a', s', +1)$ would mean that if the machine sees the symbol a on the tape when it is in state s, then it changes the tape symbol to a', changes its state to s', and moves to the right.

We start the machine by writing the input string x on the tape, with a special blank symbol Λ written on the rest of the tape stretching left and right to infinity.

We place the head at one end of the input, and start it in some specific initial state.

The machine uses the tape as its workspace, reading and writing symbols representing its intermediate results until the output is all that remains.

Turing machines

Definition of a Turing machine

If A is the set of symbols that can appear on the tape, and S is the set of internal states, we can write the machine's behavior as a transition function:

$$F : A \times S \to A \times S \times \{\pm 1\}.$$

For instance, $F(a, s) = (a', s', +1)$ would mean that if the machine sees the symbol a on the tape when it is in state s, then it changes the tape symbol to a', changes its state to s', and moves to the right.

We start the machine by writing the input string x on the tape, with a special blank symbol Λ written on the rest of the tape stretching left and right to infinity.

We place the head at one end of the input, and start it in some specific initial state.

The machine uses the tape as its workspace, reading and writing symbols representing its intermediate results until the output is all that remains.

At that point, it enters a special *HALT* state to announce that it's done.

Turing machines

Turing machines

Definition of a Turing machine

Turing machines

Definition of a Turing machine

We can think of a Turing machine's transition function as its "source code".

Turing machines

Definition of a Turing machine

We can think of a Turing machine's transition function as its "source code".

We can write out a table of its values as a finite string of symbols.

Turing machines

Definition of a Turing machine

We can think of a Turing machine's transition function as its "source code".

We can write out a table of its values as a finite string of symbols.

Example: The Successor Function

Turing machines

Definition of a Turing machine

We can think of a Turing machine's transition function as its "source code".

We can write out a table of its values as a finite string of symbols.

Example: The Successor Function

Let $A = \{\Lambda, 1\}$, and let n be represented with $(n + 1)$ 1s.

Turing machines

Definition of a Turing machine

We can think of a Turing machine's transition function as its "source code".

We can write out a table of its values as a finite string of symbols.

Example: The Successor Function

Let $A = \{\Lambda, 1\}$, and let n be represented with $(n + 1)$ 1s. Design a Turing machine that computes the successor function, that is, $S(x) = x + 1$.

Turing machines

Definition of a Turing machine

We can think of a Turing machine's transition function as its "source code".

We can write out a table of its values as a finite string of symbols.

Example: The Successor Function

Let $A = \{\Lambda, 1\}$, and let n be represented with $(n + 1)$ 1s. Design a Turing machine that computes the successor function, that is, $S(x) = x + 1$. Repeat the same with diagrams.

Turing machines

Definition of a Turing machine

We can think of a Turing machine's transition function as its "source code".

We can write out a table of its values as a finite string of symbols.

Example: The Successor Function

Let $A = \{\Lambda, 1\}$, and let n be represented with $(n + 1)$ 1s. Design a Turing machine that computes the successor function, that is, $S(x) = x + 1$. Repeat the same with diagrams.

Example: The parity function

Turing machines

Definition of a Turing machine

We can think of a Turing machine's transition function as its "source code".

We can write out a table of its values as a finite string of symbols.

Example: The Successor Function

Let $A = \{\Lambda, 1\}$, and let n be represented with $(n + 1)$ 1s. Design a Turing machine that computes the successor function, that is, $S(x) = x + 1$. Repeat the same with diagrams.

Example: The parity function

Let $A = \{\Lambda, 0, 1\}$.

Turing machines

Definition of a Turing machine

We can think of a Turing machine's transition function as its "source code".

We can write out a table of its values as a finite string of symbols.

Example: The Successor Function

Let $A = \{\Lambda, 1\}$, and let n be represented with $(n + 1)$ 1s. Design a Turing machine that computes the successor function, that is, $S(x) = x + 1$. Repeat the same with diagrams.

Example: The parity function

Let $A = \{\Lambda, 0, 1\}$. Design a Turing machine that halts in the state S_{YES}, if in the input string the number of 1s is even, and halts in the state S_{NO}-otherwise.

Turing machines

Definition of a Turing machine

We can think of a Turing machine's transition function as its "source code".

We can write out a table of its values as a finite string of symbols.

Example: The Successor Function

Let $A = \{\Lambda, 1\}$, and let n be represented with $(n + 1)$ 1s. Design a Turing machine that computes the successor function, that is, $S(x) = x + 1$. Repeat the same with diagrams.

Example: The parity function

Let $A = \{\Lambda, 0, 1\}$. Design a Turing machine that halts in the state S_{YES}, if in the input string the number of 1s is even, and halts in the state S_{NO}-otherwise. Repeat the same with diagrams.

Turing machines

Turing machines

Turing-computable functions

Turing machines

Turing-computable functions

We say that a partial function $f : N \longrightarrow N$ is Turing-computable,

Turing machines

Turing-computable functions

We say that a partial function $f : N \longrightarrow N$ is Turing-computable, if there is a Turing-machine that, given input x, halts with $f(x)$ on its input tape if $f(x)$ is defined,

Turing machines

Turing-computable functions

We say that a partial function $f : N \longrightarrow N$ is Turing-computable, if there is a Turing-machine that, given input x, halts with $f(x)$ on its input tape if $f(x)$ is defined, and runs forever if $f(x)$ is undefined.

Turing machines

Turing-computable functions

We say that a partial function $f : N \longrightarrow N$ is Turing-computable, if there is a Turing-machine that, given input x, halts with $f(x)$ on its input tape if $f(x)$ is defined, and runs forever if $f(x)$ is undefined.

The Grand Unification

Turing machines

Turing-computable functions

We say that a partial function $f : N \longrightarrow N$ is Turing-computable, if there is a Turing-machine that, given input x, halts with $f(x)$ on its input tape if $f(x)$ is defined, and runs forever if $f(x)$ is undefined.

The Grand Unification

It can be shown that the class of Turing-computable functions coincides with that of partial recursive functions.

Church-Turing thesis

Church-Turing thesis

The thesis

Church-Turing thesis

The thesis

Any two definitions of computability will result to the same class of computable functions.

A counter machine

A counter machine

Restricting the memory and the window

A counter machine

Restricting the memory and the window

If a machine has only a finite number of states, with no access to anything but its own thoughts, then it is doomed to halt or fall into a periodic loop after a finite number of states.

A counter machine

Restricting the memory and the window

If a machine has only a finite number of states, with no access to anything but its own thoughts, then it is doomed to halt or fall into a periodic loop after a finite number of states.

However, if we give it access to a memory with an infinite number of possible states, then it is capable of universal computation.

A counter machine

Restricting the memory and the window

If a machine has only a finite number of states, with no access to anything but its own thoughts, then it is doomed to halt or fall into a periodic loop after a finite number of states.

However, if we give it access to a memory with an infinite number of possible states, then it is capable of universal computation.

Turing showed that this is the case even if this memory has a simple structure,

A counter machine

Restricting the memory and the window

If a machine has only a finite number of states, with no access to anything but its own thoughts, then it is doomed to halt or fall into a periodic loop after a finite number of states.

However, if we give it access to a memory with an infinite number of possible states, then it is capable of universal computation.

Turing showed that this is the case even if this memory has a simple structure, and the machine can only observe and modify it through a very narrow window

A counter machine

Restricting the memory and the window

If a machine has only a finite number of states, with no access to anything but its own thoughts, then it is doomed to halt or fall into a periodic loop after a finite number of states.

However, if we give it access to a memory with an infinite number of possible states, then it is capable of universal computation.

Turing showed that this is the case even if this memory has a simple structure, and the machine can only observe and modify it through a very narrow window - namely the symbol and its current location on the tape.

A counter machine

Restricting the memory and the window

If a machine has only a finite number of states, with no access to anything but its own thoughts, then it is doomed to halt or fall into a periodic loop after a finite number of states.

However, if we give it access to a memory with an infinite number of possible states, then it is capable of universal computation.

Turing showed that this is the case even if this memory has a simple structure, and the machine can only observe and modify it through a very narrow window - namely the symbol and its current location on the tape.

Can we make the memory even simpler, and this window even narrower?

A counter machine

A counter machine

Restricting the memory and the window

A counter machine

Restricting the memory and the window

A counter machine has a finite number of internal states, and access to a finite number of integer counters.

A counter machine

Restricting the memory and the window

A counter machine has a finite number of internal states, and access to a finite number of integer counters.

The only thing it can do to these counters is increment, or decrement them, and the only question it can ask about them is whether they are zero.

A counter machine

Restricting the memory and the window

A counter machine has a finite number of internal states, and access to a finite number of integer counters.

The only thing it can do to these counters is increment, or decrement them, and the only question it can ask about them is whether they are zero.

The machine receives its input through the initial state of one of these counters, and like the Turing machine, it enters a *HALT* state when it is done.

A counter machine

A counter machine

Restricting the memory and the window

A counter machine

Restricting the memory and the window

We can think of a counter machine as a flowchart or a program in a miniature programming language.

A counter machine

Restricting the memory and the window

We can think of a counter machine as a flowchart or a program in a miniature programming language.

Each internal state corresponds to a node in the flowchart or a line of the program,

A counter machine

Restricting the memory and the window

We can think of a counter machine as a flowchart or a program in a miniature programming language.

Each internal state corresponds to a node in the flowchart or a line of the program, and the only allowed instructions are *inc* (increment),

A counter machine

Restricting the memory and the window

We can think of a counter machine as a flowchart or a program in a miniature programming language.

Each internal state corresponds to a node in the flowchart or a line of the program, and the only allowed instructions are *inc* (increment), *dec* (decrement),

A counter machine

Restricting the memory and the window

We can think of a counter machine as a flowchart or a program in a miniature programming language.

Each internal state corresponds to a node in the flowchart or a line of the program, and the only allowed instructions are *inc* (increment), *dec* (decrement), and conditionals like *if* $x = 0$.

A counter machine

Restricting the memory and the window

We can think of a counter machine as a flowchart or a program in a miniature programming language.

Each internal state corresponds to a node in the flowchart or a line of the program, and the only allowed instructions are *inc* (increment), *dec* (decrement), and conditionals like *if* $x = 0$.

One can show that there is a way of simulating a Turing machine arithmetically.

A counter machine

Restricting the memory and the window

We can think of a counter machine as a flowchart or a program in a miniature programming language.

Each internal state corresponds to a node in the flowchart or a line of the program, and the only allowed instructions are *inc* (increment), *dec* (decrement), and conditionals like *if* $x = 0$.

One can show that there is a way of simulating a Turing machine arithmetically.

Specifically, we can transform any Turing machine into a counter machine with just three counters.

A counter machine

Restricting the memory and the window

We can think of a counter machine as a flowchart or a program in a miniature programming language.

Each internal state corresponds to a node in the flowchart or a line of the program, and the only allowed instructions are *inc* (increment), *dec* (decrement), and conditionals like *if* $x = 0$.

One can show that there is a way of simulating a Turing machine arithmetically.

Specifically, we can transform any Turing machine into a counter machine with just three counters.

Thus the Halting Problem for three-counter machines is undecidable.

A counter machine

Restricting the memory and the window

We can think of a counter machine as a flowchart or a program in a miniature programming language.

Each internal state corresponds to a node in the flowchart or a line of the program, and the only allowed instructions are *inc* (increment), *dec* (decrement), and conditionals like *if* $x = 0$.

One can show that there is a way of simulating a Turing machine arithmetically.

Specifically, we can transform any Turing machine into a counter machine with just three counters.

Thus the Halting Problem for three-counter machines is undecidable.

Moreover, three-counter machines can compute any partial recursive function f, by starting with x in one counter and ending with $f(x)$ in that counter when they halt.

Multiplicative counter machines

Multiplicative counter machines

Reducing the number of counters

Multiplicative counter machines

Reducing the number of counters

We can reduce the number of counters even further.

Multiplicative counter machines

Reducing the number of counters

We can reduce the number of counters even further.

Consider multiplicative counter machines, where each step can multiply or divide a counter by a constant,

Multiplicative counter machines

Reducing the number of counters

We can reduce the number of counters even further.

Consider multiplicative counter machines, where each step can multiply or divide a counter by a constant, or check to see whether a counter is a multiple of some constant.

Multiplicative counter machines

Reducing the number of counters

We can reduce the number of counters even further.

Consider multiplicative counter machines, where each step can multiply or divide a counter by a constant, or check to see whether a counter is a multiple of some constant.

Now even a single counter suffices.

Multiplicative counter machines

Reducing the number of counters

We can reduce the number of counters even further.

Consider multiplicative counter machines, where each step can multiply or divide a counter by a constant, or check to see whether a counter is a multiple of some constant.

Now even a single counter suffices. This is because we can encode three additive counters as a single multiplicative one:

Multiplicative counter machines

Reducing the number of counters

We can reduce the number of counters even further.

Consider multiplicative counter machines, where each step can multiply or divide a counter by a constant, or check to see whether a counter is a multiple of some constant.

Now even a single counter suffices. This is because we can encode three additive counters as a single multiplicative one:

$$w = 2^x \cdot 3^y \cdot 5^z.$$

Multiplicative counter machines

Reducing the number of counters

We can reduce the number of counters even further.

Consider multiplicative counter machines, where each step can multiply or divide a counter by a constant, or check to see whether a counter is a multiple of some constant.

Now even a single counter suffices. This is because we can encode three additive counters as a single multiplicative one:

$$w = 2^x \cdot 3^y \cdot 5^z.$$

We can increment or decrement y, say by multiplying or dividing w by 3.

Multiplicative counter machines

Reducing the number of counters

We can reduce the number of counters even further.

Consider multiplicative counter machines, where each step can multiply or divide a counter by a constant, or check to see whether a counter is a multiple of some constant.

Now even a single counter suffices. This is because we can encode three additive counters as a single multiplicative one:

$$w = 2^x \cdot 3^y \cdot 5^z.$$

We can increment or decrement y, say by multiplying or dividing w by 3. Similarly, we can test whether $y \neq 0$ by asking whether w is a multiple of 3.

Multiplicative counter machines

Multiplicative counter machines

Reducing the number of counters

Multiplicative counter machines

Reducing the number of counters

But a one-counter multiplicative machine, in turn, can be simulated by a two-counter additive one.

Multiplicative counter machines

Reducing the number of counters

But a one-counter multiplicative machine, in turn, can be simulated by a two-counter additive one.

If we have an additional counter u, we can multiply or divide w by 2, 3 or 5, or check to see whether w is a multiple of any of these using loops.

Multiplicative counter machines

Reducing the number of counters

But a one-counter multiplicative machine, in turn, can be simulated by a two-counter additive one.

If we have an additional counter u, we can multiply or divide w by 2, 3 or 5, or check to see whether w is a multiple of any of these using loops.

For instance, in order to divide w by 3, we decrement w three times for each time we increment u.

Multiplicative counter machines

Reducing the number of counters

But a one-counter multiplicative machine, in turn, can be simulated by a two-counter additive one.

If we have an additional counter u, we can multiply or divide w by 2, 3 or 5, or check to see whether w is a multiple of any of these using loops.

For instance, in order to divide w by 3, we decrement w three times for each time we increment u. If w becomes zero partway though a group of three decrements, we know that w is not a multiple of 3.

Multiplicative counter machines

Reducing the number of counters

But a one-counter multiplicative machine, in turn, can be simulated by a two-counter additive one.

If we have an additional counter u, we can multiply or divide w by 2, 3 or 5, or check to see whether w is a multiple of any of these using loops.

For instance, in order to divide w by 3, we decrement w three times for each time we increment u. If w becomes zero partway though a group of three decrements, we know that w is not a multiple of 3.

Thus even two counters suffice for universal computation.

Multiplicative counter machines

Reducing the number of counters

But a one-counter multiplicative machine, in turn, can be simulated by a two-counter additive one.

If we have an additional counter u, we can multiply or divide w by 2, 3 or 5, or check to see whether w is a multiple of any of these using loops.

For instance, in order to divide w by 3, we decrement w three times for each time we increment u. If w becomes zero partway though a group of three decrements, we know that w is not a multiple of 3.

Thus even two counters suffice for universal computation. Specifically, for any partial recursive function f,

Multiplicative counter machines

Reducing the number of counters

But a one-counter multiplicative machine, in turn, can be simulated by a two-counter additive one.

If we have an additional counter u, we can multiply or divide w by 2, 3 or 5, or check to see whether w is a multiple of any of these using loops.

For instance, in order to divide w by 3, we decrement w three times for each time we increment u. If w becomes zero partway though a group of three decrements, we know that w is not a multiple of 3.

Thus even two counters suffice for universal computation. Specifically, for any partial recursive function f, there is a two-counter-machine which, given the initial value $w = 2^x$,

Multiplicative counter machines

Reducing the number of counters

But a one-counter multiplicative machine, in turn, can be simulated by a two-counter additive one.

If we have an additional counter u, we can multiply or divide w by 2, 3 or 5, or check to see whether w is a multiple of any of these using loops.

For instance, in order to divide w by 3, we decrement w three times for each time we increment u. If w becomes zero partway though a group of three decrements, we know that w is not a multiple of 3.

Thus even two counters suffice for universal computation. Specifically, for any partial recursive function f, there is a two-counter-machine which, given the initial value $w = 2^x$, ends with $w = 3^{f(x)}$,

Multiplicative counter machines

Reducing the number of counters

But a one-counter multiplicative machine, in turn, can be simulated by a two-counter additive one.

If we have an additional counter u, we can multiply or divide w by 2, 3 or 5, or check to see whether w is a multiple of any of these using loops.

For instance, in order to divide w by 3, we decrement w three times for each time we increment u. If w becomes zero partway though a group of three decrements, we know that w is not a multiple of 3.

Thus even two counters suffice for universal computation. Specifically, for any partial recursive function f, there is a two-counter-machine which, given the initial value $w = 2^x$, ends with $w = 3^{f(x)}$, or runs forever if $f(x)$ is undefined.

Fantastic Fractions

Fantastic Fractions

Programs as fractions

Fantastic Fractions

Programs as fractions

Here is a program written in a rather odd programming language called *FRACTARAN*, consisting of a list of fractions:

Fantastic Fractions

Programs as fractions

Here is a program written in a rather odd programming language called *FRACTARAN*, consisting of a list of fractions:

$$\frac{17}{91}, \frac{78}{85}, \frac{19}{51}, \frac{23}{38}, \frac{29}{33}, \frac{77}{29}, \frac{95}{23}, \frac{77}{19}, \frac{1}{17}, \frac{11}{13}, \frac{13}{11}, \frac{15}{14}, \frac{15}{2}, \frac{55}{1}.$$

Fantastic Fractions

Programs as fractions

Here is a program written in a rather odd programming language called *FRACTARAN*, consisting of a list of fractions:

$$\frac{17}{91}, \frac{78}{85}, \frac{19}{51}, \frac{23}{38}, \frac{29}{33}, \frac{77}{29}, \frac{95}{23}, \frac{77}{19}, \frac{1}{17}, \frac{11}{13}, \frac{13}{11}, \frac{15}{14}, \frac{15}{2}, \frac{55}{1}.$$

The state of the computer consists of an integer n.

Fantastic Fractions

Programs as fractions

Here is a program written in a rather odd programming language called *FRACTARAN*, consisting of a list of fractions:

$$\frac{17}{91}, \frac{78}{85}, \frac{19}{51}, \frac{23}{38}, \frac{29}{33}, \frac{77}{29}, \frac{95}{23}, \frac{77}{19}, \frac{1}{17}, \frac{11}{13}, \frac{13}{11}, \frac{15}{14}, \frac{15}{2}, \frac{55}{1}.$$

The state of the computer consists of an integer n.

At each step, we run through the list until we find the first fraction $\frac{p}{q}$, such that $\frac{p}{q} \cdot n$ is an integer

Fantastic Fractions

Programs as fractions

Here is a program written in a rather odd programming language called *FRACTARAN*, consisting of a list of fractions:

$$\frac{17}{91}, \frac{78}{85}, \frac{19}{51}, \frac{23}{38}, \frac{29}{33}, \frac{77}{29}, \frac{95}{23}, \frac{77}{19}, \frac{1}{17}, \frac{11}{13}, \frac{13}{11}, \frac{15}{14}, \frac{15}{2}, \frac{55}{1}.$$

The state of the computer consists of an integer n.

At each step, we run through the list until we find the first fraction $\frac{p}{q}$, such that $\frac{p}{q} \cdot n$ is an integer (that is, the first one whose denominator q divides n).

Fantastic Fractions

Programs as fractions

Here is a program written in a rather odd programming language called *FRACTARAN*, consisting of a list of fractions:

$$\frac{17}{91}, \frac{78}{85}, \frac{19}{51}, \frac{23}{38}, \frac{29}{33}, \frac{77}{29}, \frac{95}{23}, \frac{77}{19}, \frac{1}{17}, \frac{11}{13}, \frac{13}{11}, \frac{15}{14}, \frac{15}{2}, \frac{55}{1}.$$

The state of the computer consists of an integer n.

At each step, we run through the list until we find the first fraction $\frac{p}{q}$, such that $\frac{p}{q} \cdot n$ is an integer (that is, the first one whose denominator q divides n).

Then we multiply n by $\frac{p}{q}$, and start again at the beginning of the list.

Fantastic Fractions

Programs as fractions

Here is a program written in a rather odd programming language called *FRACTARAN*, consisting of a list of fractions:

$$\frac{17}{91}, \frac{78}{85}, \frac{19}{51}, \frac{23}{38}, \frac{29}{33}, \frac{77}{29}, \frac{95}{23}, \frac{77}{19}, \frac{1}{17}, \frac{11}{13}, \frac{13}{11}, \frac{15}{14}, \frac{15}{2}, \frac{55}{1}.$$

The state of the computer consists of an integer n.

At each step, we run through the list until we find the first fraction $\frac{p}{q}$, such that $\frac{p}{q} \cdot n$ is an integer (that is, the first one whose denominator q divides n).

Then we multiply n by $\frac{p}{q}$, and start again at the beginning of the list. If there is no such $\frac{p}{q}$ in the list, the program halts.

Fantastic Fractions

Programs as fractions

Here is a program written in a rather odd programming language called *FRACTARAN*, consisting of a list of fractions:

$$\frac{17}{91}, \frac{78}{85}, \frac{19}{51}, \frac{23}{38}, \frac{29}{33}, \frac{77}{29}, \frac{95}{23}, \frac{77}{19}, \frac{1}{17}, \frac{11}{13}, \frac{13}{11}, \frac{15}{14}, \frac{15}{2}, \frac{55}{1}.$$

The state of the computer consists of an integer n.

At each step, we run through the list until we find the first fraction $\frac{p}{q}$, such that $\frac{p}{q} \cdot n$ is an integer (that is, the first one whose denominator q divides n).

Then we multiply n by $\frac{p}{q}$, and start again at the beginning of the list. If there is no such $\frac{p}{q}$ in the list, the program halts.

The program mentioned above never halts,

Fantastic Fractions

Programs as fractions

Here is a program written in a rather odd programming language called *FRACTARAN*, consisting of a list of fractions:

$$\frac{17}{91}, \frac{78}{85}, \frac{19}{51}, \frac{23}{38}, \frac{29}{33}, \frac{77}{29}, \frac{95}{23}, \frac{77}{19}, \frac{1}{17}, \frac{11}{13}, \frac{13}{11}, \frac{15}{14}, \frac{15}{2}, \frac{55}{1}.$$

The state of the computer consists of an integer n.

At each step, we run through the list until we find the first fraction $\frac{p}{q}$, such that $\frac{p}{q} \cdot n$ is an integer (that is, the first one whose denominator q divides n).

Then we multiply n by $\frac{p}{q}$, and start again at the beginning of the list. If there is no such $\frac{p}{q}$ in the list, the program halts.

The program mentioned above never halts, If we start it with $n = 2$, this program produces the prime numbers x, in increasing order, in the form 2^x.

Fantastic Fractions

Fantastic Fractions

Programs as fractions

Fantastic Fractions

Programs as fractions

The program mentioned above is kind of a multiplicative counter machine.

Fantastic Fractions

Programs as fractions

The program mentioned above is kind of a multiplicative counter machine. In this case, we can write the state in the form

$$n = 2^a \cdot 3^b \cdot 5^c \cdot 7^d \cdot 11^e \cdot 13^f \cdot 17^g \cdot 19^h \cdot 23^i \cdot 29^j$$

Fantastic Fractions

Programs as fractions

The program mentioned above is kind of a multiplicative counter machine. In this case, we can write the state in the form

$$n = 2^a \cdot 3^b \cdot 5^c \cdot 7^d \cdot 11^e \cdot 13^f \cdot 17^g \cdot 19^h \cdot 23^i \cdot 29^j$$

since these are the primes appearing in the fractions' denominators.

Fantastic Fractions

Programs as fractions

The program mentioned above is kind of a multiplicative counter machine. In this case, we can write the state in the form

$$n = 2^a \cdot 3^b \cdot 5^c \cdot 7^d \cdot 11^e \cdot 13^f \cdot 17^g \cdot 19^h \cdot 23^i \cdot 29^j$$

since these are the primes appearing in the fractions' denominators.

Thus we have 10 counters, which we can increment and decrement by multiplying n by fractions.

Fantastic Fractions

Programs as fractions

The program mentioned above is kind of a multiplicative counter machine. In this case, we can write the state in the form

$$n = 2^a \cdot 3^b \cdot 5^c \cdot 7^d \cdot 11^e \cdot 13^f \cdot 17^g \cdot 19^h \cdot 23^i \cdot 29^j$$

since these are the primes appearing in the fractions' denominators.

Thus we have 10 counters, which we can increment and decrement by multiplying n by fractions.

For instance, the first fraction in the program is $\frac{17}{91}$.

Fantastic Fractions

Programs as fractions

The program mentioned above is kind of a multiplicative counter machine. In this case, we can write the state in the form

$$n = 2^a \cdot 3^b \cdot 5^c \cdot 7^d \cdot 11^e \cdot 13^f \cdot 17^g \cdot 19^h \cdot 23^i \cdot 29^j$$

since these are the primes appearing in the fractions' denominators.

Thus we have 10 counters, which we can increment and decrement by multiplying n by fractions.

For instance, the first fraction in the program is $\frac{17}{91}$. So if both d and f are nonzero, we decrement them and increment g.

Fantastic Fractions

Fantastic Fractions

Programs as fractions

Fantastic Fractions

Programs as fractions

We already know that additive counters can simulate Turing machines.

Fantastic Fractions

Programs as fractions

We already know that additive counters can simulate Turing machines.

With a bit more work, one can covert such a machine into a *FRACTARAN* program.

Fantastic Fractions

Programs as fractions

We already know that additive counters can simulate Turing machines.

With a bit more work, one can covert such a machine into a *FRACTARAN* program. Thus *FRACTARAN* is computationally universal.

Fantastic Fractions

Programs as fractions

We already know that additive counters can simulate Turing machines.

With a bit more work, one can covert such a machine into a *FRACTARAN* program. Thus *FRACTARAN* is computationally universal.

For any computable function $f(x)$ there is a program that, if given 2^x as input, returns $3^{f(x)}$ as output whenever $f(x)$ is well-defined.

Fantastic Fractions and Collatz Conjecture

Fantastic Fractions and Collatz Conjecture

The Collatz Conjecture

Fantastic Fractions and Collatz Conjecture

The Collatz Conjecture

FRACTARAN is related to a deep problem in number theory.

Fantastic Fractions and Collatz Conjecture

The Collatz Conjecture

FRACTARAN is related to a deep problem in number theory.

Consider the function

$$g(x) = \begin{cases} \frac{x}{2}, & \text{if } x \text{ is even} \\ 3 \cdot x + 1, & \text{if } x \text{ is odd.} \end{cases}$$

Fantastic Fractions and Collatz Conjecture

The Collatz Conjecture

FRACTARAN is related to a deep problem in number theory.

Consider the function

$$g(x) = \begin{cases} \frac{x}{2}, & \text{if } x \text{ is even} \\ 3 \cdot x + 1, & \text{if } x \text{ is odd.} \end{cases}$$

The Collatz Conjecture states that for each n,

Fantastic Fractions and Collatz Conjecture

The Collatz Conjecture

FRACTARAN is related to a deep problem in number theory.

Consider the function

$$g(x) = \begin{cases} \frac{x}{2}, & \text{if } x \text{ is even} \\ 3 \cdot x + 1, & \text{if } x \text{ is odd.} \end{cases}$$

The Collatz Conjecture states that for each n, there is a k, such that $g^k(n) = 1$.

Fantastic Fractions and Collatz Conjecture

The Collatz Conjecture

FRACTARAN is related to a deep problem in number theory.

Consider the function

$$g(x) = \begin{cases} \frac{x}{2}, & \text{if } x \text{ is even} \\ 3 \cdot x + 1, & \text{if } x \text{ is odd.} \end{cases}$$

The Collatz Conjecture states that for each n, there is a k, such that $g^k(n) = 1$.

This conjecture has been verified for all n up to 10^{18}, and there are convincing heuristic arguments for it.

Fantastic Fractions and Collatz Conjecture

The Collatz Conjecture

FRACTARAN is related to a deep problem in number theory.

Consider the function

$$g(x) = \begin{cases} \frac{x}{2}, & \text{if } x \text{ is even} \\ 3 \cdot x + 1, & \text{if } x \text{ is odd.} \end{cases}$$

The Collatz Conjecture states that for each n, there is a k, such that $g^k(n) = 1$.

This conjecture has been verified for all n up to 10^{18}, and there are convincing heuristic arguments for it. However, there is still no proof, and resolving it seems quite difficult.

Fantastic Fractions and Collatz Conjecture

Fantastic Fractions and Collatz Conjecture

The Collatz Conjecture

Fantastic Fractions and Collatz Conjecture

The Collatz Conjecture

The $3 \cdot x + 1$ function belongs to a family of functions whose behavior depends on x mod q for some q:

Fantastic Fractions and Collatz Conjecture

The Collatz Conjecture

The $3 \cdot x + 1$ function belongs to a family of functions whose behavior depends on x mod q for some q:

$$g(x) = \frac{a_i}{q} \cdot (x - i) + b_i, \text{ where } x = i \mod q.$$

Fantastic Fractions and Collatz Conjecture

The Collatz Conjecture

The $3 \cdot x + 1$ function belongs to a family of functions whose behavior depends on x mod q for some q:

$$g(x) = \frac{a_i}{q} \cdot (x - i) + b_i, \text{ where } x = i \mod q.$$

Here a_i and b_i are integer coefficients for $0 \leq i < q$.

Fantastic Fractions and Collatz Conjecture

The Collatz Conjecture

The $3 \cdot x + 1$ function belongs to a family of functions whose behavior depends on x mod q for some q:

$$g(x) = \frac{a_i}{q} \cdot (x - i) + b_i, \text{ where } x = i \mod q.$$

Here a_i and b_i are integer coefficients for $0 \leq i < q$.

For any function of this form, we can ask whether a given initial x will reach some y after certain number of iterations:

Fantastic Fractions and Collatz Conjecture

The Collatz Conjecture

The $3 \cdot x + 1$ function belongs to a family of functions whose behavior depends on $x \mod q$ for some q:

$$g(x) = \frac{a_i}{q} \cdot (x - i) + b_i, \text{ where } x = i \mod q.$$

Here a_i and b_i are integer coefficients for $0 \leq i < q$.

For any function of this form, we can ask whether a given initial x will reach some y after certain number of iterations:

Problem

Collatz: Given x, y and q, and integer coefficients a_i, b_i for $0 \leq i < q$. Check whether there is an integer t such that $g^t(x) = y$?

Fantastic Fractions and Collatz Conjecture

The Collatz Conjecture

The $3 \cdot x + 1$ function belongs to a family of functions whose behavior depends on $x \mod q$ for some q:

$$g(x) = \frac{a_i}{q} \cdot (x - i) + b_i, \text{ where } x = i \mod q.$$

Here a_i and b_i are integer coefficients for $0 \leq i < q$.

For any function of this form, we can ask whether a given initial x will reach some y after certain number of iterations:

Problem

Collatz: Given x, y and q, and integer coefficients a_i, b_i for $0 \leq i < q$. Check whether there is an integer t such that $g^t(x) = y$?

Remark

It can be shown that HALTING $\leq$ Collatz.

Fantastic Fractions and Collatz Conjecture

The Collatz Conjecture

The $3 \cdot x + 1$ function belongs to a family of functions whose behavior depends on $x \mod q$ for some q:

$$g(x) = \frac{a_i}{q} \cdot (x - i) + b_i, \text{ where } x = i \mod q.$$

Here a_i and b_i are integer coefficients for $0 \leq i < q$.

For any function of this form, we can ask whether a given initial x will reach some y after certain number of iterations:

Problem

Collatz: Given x, y and q, and integer coefficients a_i, b_i for $0 \leq i < q$. Check whether there is an integer t such that $g^t(x) = y$?

Remark

It can be shown that HALTING $\leq$ Collatz. Thus Collatz is undecidable.

The Game of Tag

The Game of Tag

Tag Systems

The Game of Tag

Tag Systems

Emil Post showed that the axioms of the Principia can be reduced to a single initial string,

The Game of Tag

Tag Systems

Emil Post showed that the axioms of the Principia can be reduced to a single initial string, and that its rules of inference can be replaced with the rules of the following normal form:

The Game of Tag

Tag Systems

Emil Post showed that the axioms of the Principia can be reduced to a single initial string, and that its rules of inference can be replaced with the rules of the following normal form:

$$gs \Rightarrow sh,$$

The Game of Tag

Tag Systems

Emil Post showed that the axioms of the Principia can be reduced to a single initial string, and that its rules of inference can be replaced with the rules of the following normal form:

$$gs \Rightarrow sh,$$

where g and h are fixed strings and s is arbitrary.

The Game of Tag

Tag Systems

Emil Post showed that the axioms of the Principia can be reduced to a single initial string, and that its rules of inference can be replaced with the rules of the following normal form:

$$gs \Rightarrow sh,$$

where g and h are fixed strings and s is arbitrary.

In other words, for any string that begins with g, we can remove g from its head and append h to its tail.

The Game of Tag

Tag Systems

Emil Post showed that the axioms of the Principia can be reduced to a single initial string, and that its rules of inference can be replaced with the rules of the following normal form:

$$gs \Rightarrow sh,$$

where g and h are fixed strings and s is arbitrary.

In other words, for any string that begins with g, we can remove g from its head and append h to its tail.

Post was hoping to solve the Entscheidungsproblem by finding an algorithm to determine whether a given string can be produced from an initial string, representing system's axioms,

The Game of Tag

Tag Systems

Emil Post showed that the axioms of the Principia can be reduced to a single initial string, and that its rules of inference can be replaced with the rules of the following normal form:

$$gs \Rightarrow sh,$$

where g and h are fixed strings and s is arbitrary.

In other words, for any string that begins with g, we can remove g from its head and append h to its tail.

Post was hoping to solve the Entscheidungsproblem by finding an algorithm to determine whether a given string can be produced from an initial string, representing system's axioms, using a given set of rules $\{(g_i, h_i)\}$ in normal form.

The Game of Tag

Tag Systems

Emil Post showed that the axioms of the Principia can be reduced to a single initial string, and that its rules of inference can be replaced with the rules of the following normal form:

$$gs \Rightarrow sh,$$

where g and h are fixed strings and s is arbitrary.

In other words, for any string that begins with g, we can remove g from its head and append h to its tail.

Post was hoping to solve the Entscheidungsproblem by finding an algorithm to determine whether a given string can be produced from an initial string, representing system's axioms, using a given set of rules $\{(g_i, h_i)\}$ in normal form.

In general, the result of such a computation might be non-deterministic, since we might have a choice of which rule to apply to a given string.

The Game of Tag

Tag Systems

Emil Post showed that the axioms of the Principia can be reduced to a single initial string, and that its rules of inference can be replaced with the rules of the following normal form:

$$gs \Rightarrow sh,$$

where g and h are fixed strings and s is arbitrary.

In other words, for any string that begins with g, we can remove g from its head and append h to its tail.

Post was hoping to solve the Entscheidungsproblem by finding an algorithm to determine whether a given string can be produced from an initial string, representing system's axioms, using a given set of rules $\{(g_i, h_i)\}$ in normal form.

In general, the result of such a computation might be non-deterministic, since we might have a choice of which rule to apply to a given string.

For instance with the rules $\{(a, b), (ab, c)\}$, we could change abc to either bcb or cc.

The Game of Tag

The Game of Tag

Tag Systems

The Game of Tag

Tag Systems

As a start, Post considered the case where all the g_i have the same length ν for some constant ν,

The Game of Tag

Tag Systems

As a start, Post considered the case where all the g_i have the same length ν for some constant ν, and where each h_i is determined by g_i's first symbol a.

The Game of Tag

Tag Systems

As a start, Post considered the case where all the g_i have the same length ν for some constant ν, and where each h_i is determined by g_i's first symbol a.

Thus, at each step, if a string begins with the symbol a, we remove its first ν symbols including a,

The Game of Tag

Tag Systems

As a start, Post considered the case where all the g_i have the same length ν for some constant ν, and where each h_i is determined by g_i's first symbol a.

Thus, at each step, if a string begins with the symbol a, we remove its first ν symbols including a, and append the string $h(a)$.

The Game of Tag

Tag Systems

As a start, Post considered the case where all the g_i have the same length ν for some constant ν, and where each h_i is determined by g_i's first symbol a.

Thus, at each step, if a string begins with the symbol a, we remove its first ν symbols including a, and append the string $h(a)$.

This process is deterministic, so each initial string leads to a fixed sequence.

The Game of Tag

Tag Systems

As a start, Post considered the case where all the g_i have the same length ν for some constant ν, and where each h_i is determined by g_i's first symbol a.

Thus, at each step, if a string begins with the symbol a, we remove its first ν symbols including a, and append the string $h(a)$.

This process is deterministic, so each initial string leads to a fixed sequence. If this sequence reaches a string whose length is less than ν, it halts.

The Game of Tag

The Game of Tag

Tag Systems

The Game of Tag

Tag Systems

These are called tag systems.

The Game of Tag

Tag Systems

These are called tag systems.

Post initially expected them to be an easy special case.

The Game of Tag

Tag Systems

These are called tag systems.

Post initially expected them to be an easy special case.

Later, he was able to prove that *Collatz* can be reduced to this problem, hence it is undecidable.

The Game of Tag

Tag Systems

These are called tag systems.

Post initially expected them to be an easy special case.

Later, he was able to prove that *Collatz* can be reduced to this problem, hence it is undecidable.

This negative result can be applied in order to derive the undecidability of the following two problems:

The Game of Tag

Tag Systems

These are called tag systems.

Post initially expected them to be an easy special case.

Later, he was able to prove that *Collatz* can be reduced to this problem, hence it is undecidable.

This negative result can be applied in order to derive the undecidability of the following two problems:

Problem

Tag System Halting:

The Game of Tag

Tag Systems

These are called tag systems.

Post initially expected them to be an easy special case.

Later, he was able to prove that *Collatz* can be reduced to this problem, hence it is undecidable.

This negative result can be applied in order to derive the undecidability of the following two problems:

Problem

Tag System Halting: Given an integer ν,

The Game of Tag

Tag Systems

These are called tag systems.

Post initially expected them to be an easy special case.

Later, he was able to prove that *Collatz* can be reduced to this problem, hence it is undecidable.

This negative result can be applied in order to derive the undecidability of the following two problems:

Problem

Tag System Halting: Given an integer ν, a finite alphabet A,

The Game of Tag

Tag Systems

These are called tag systems.

Post initially expected them to be an easy special case.

Later, he was able to prove that *Collatz* can be reduced to this problem, hence it is undecidable.

This negative result can be applied in order to derive the undecidability of the following two problems:

Problem

Tag System Halting: Given an integer ν, a finite alphabet A, a string $h(a)$ for each $a \in A$,

The Game of Tag

Tag Systems

These are called tag systems.

Post initially expected them to be an easy special case.

Later, he was able to prove that *Collatz* can be reduced to this problem, hence it is undecidable.

This negative result can be applied in order to derive the undecidability of the following two problems:

Problem

Tag System Halting: Given an integer ν, a finite alphabet A, a string $h(a)$ for each $a \in A$, and an initial string x,

The Game of Tag

Tag Systems

These are called tag systems.

Post initially expected them to be an easy special case.

Later, he was able to prove that *Collatz* can be reduced to this problem, hence it is undecidable.

This negative result can be applied in order to derive the undecidability of the following two problems:

Problem

Tag System Halting: Given an integer ν, a finite alphabet A, a string $h(a)$ for each $a \in A$, and an initial string x, the goal is to check whether x leads to a string of length less than ν.

The Game of Tag

Tag Systems

These are called tag systems.

Post initially expected them to be an easy special case.

Later, he was able to prove that *Collatz* can be reduced to this problem, hence it is undecidable.

This negative result can be applied in order to derive the undecidability of the following two problems:

Problem

Tag System Halting: Given an integer ν, a finite alphabet A, a string $h(a)$ for each $a \in A$, and an initial string x, the goal is to check whether x leads to a string of length less than ν.

Problem

Normal Form Halting:

The Game of Tag

Tag Systems

These are called tag systems.

Post initially expected them to be an easy special case.

Later, he was able to prove that *Collatz* can be reduced to this problem, hence it is undecidable.

This negative result can be applied in order to derive the undecidability of the following two problems:

Problem

Tag System Halting: Given an integer ν, a finite alphabet A, a string $h(a)$ for each $a \in A$, and an initial string x, the goal is to check whether x leads to a string of length less than ν.

Problem

Normal Form Halting: Given a list of string pairs $\{(g, h)\}$ and an initial string x,

The Game of Tag

Tag Systems

These are called tag systems.

Post initially expected them to be an easy special case.

Later, he was able to prove that *Collatz* can be reduced to this problem, hence it is undecidable.

This negative result can be applied in order to derive the undecidability of the following two problems:

Problem

Tag System Halting: Given an integer ν, a finite alphabet A, a string $h(a)$ for each $a \in A$, and an initial string x, the goal is to check whether x leads to a string of length less than ν.

Problem

Normal Form Halting: Given a list of string pairs $\{(g, h)\}$ and an initial string x, the goal is to check whether starting from x and applying the rules $gs \Rightarrow sh$,

The Game of Tag

Tag Systems

These are called tag systems.

Post initially expected them to be an easy special case.

Later, he was able to prove that *Collatz* can be reduced to this problem, hence it is undecidable.

This negative result can be applied in order to derive the undecidability of the following two problems:

Problem

Tag System Halting: Given an integer ν, a finite alphabet A, a string $h(a)$ for each $a \in A$, and an initial string x, the goal is to check whether x leads to a string of length less than ν.

Problem

Normal Form Halting: Given a list of string pairs $\{(g, h)\}$ and an initial string x, the goal is to check whether starting from x and applying the rules $gs \Rightarrow sh$, it is possible to derive a string y to which no rule can be applied,

The Game of Tag

Tag Systems

These are called tag systems.

Post initially expected them to be an easy special case.

Later, he was able to prove that *Collatz* can be reduced to this problem, hence it is undecidable.

This negative result can be applied in order to derive the undecidability of the following two problems:

Problem

Tag System Halting: Given an integer ν, a finite alphabet A, a string $h(a)$ for each $a \in A$, and an initial string x, the goal is to check whether x leads to a string of length less than ν.

Problem

Normal Form Halting: Given a list of string pairs $\{(g, h)\}$ and an initial string x, the goal is to check whether starting from x and applying the rules $gs \Rightarrow sh$, it is possible to derive a string y to which no rule can be applied, that is, which does not begin with any g in the list.

The Correspondence Problem

The Correspondence Problem

Post's Correspondence Problem

The Correspondence Problem

Post's Correspondence Problem

Finally, Post was able to prove that the following problem is also undecidable:

The Correspondence Problem

Post's Correspondence Problem

Finally, Post was able to prove that the following problem is also undecidable:

Problem

Correspondence:

The Correspondence Problem

Post's Correspondence Problem

Finally, Post was able to prove that the following problem is also undecidable:

Problem

Correspondence: Given a list of pairs of strings $(s_1, t_1), ..., (s_k, t_k)$,

The Correspondence Problem

Post's Correspondence Problem

Finally, Post was able to prove that the following problem is also undecidable:

Problem

Correspondence: Given a list of pairs of strings $(s_1, t_1), ..., (s_k, t_k)$*, the goal is to check whether there is a finite sequence* $i_1, i_2, ..., i_l$ *with* $l \geq 1$*, such that* $s_{i_1} s_{i_2} ... s_{i_l} = t_{i_1} t_{i_2} ... t_{i_l}$.

References

References

Books

References

Books

Ch. Moore, S. Mertens, The Nature of Computation, Oxford University Press (2011).