Pure Data Foundation of Mathematics

Saul Youssef

Boston University

April, 2023

types

ee

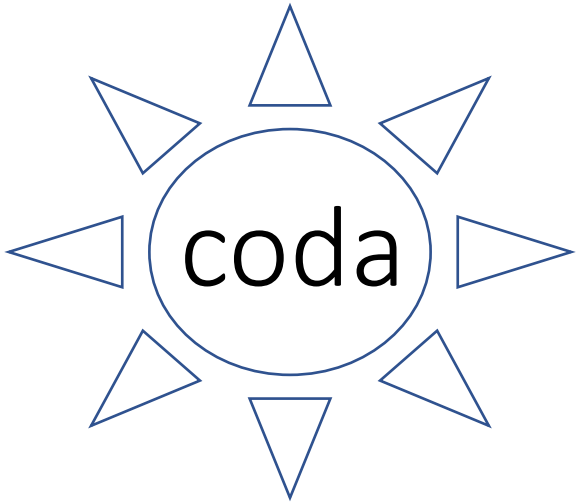
egg

With Margo Seltzer and David Parkes

Aldor

By Stephen Watt & collaborators

coda classic



types

ee

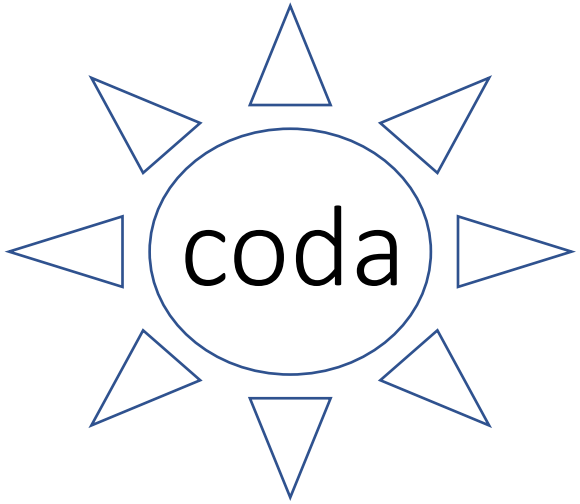
egg

With Margo Seltzer and David Parkes

Aldor

By Stephen Watt & collaborators

coda classic



- Formal system which can be a foundation of mathematics in general, analogous to ZFC or HOTT.
- A computing system where all computations are proofs and vice versa.
- Analogue of categories and types appears naturally.
- Applications are mathematical exploration, proof assistance, “mathematical machine learning.”
- This is Model theory-like in the sense that we can address issues like the consistency of Mathematics.

Mathematics is defined by formal systems

Predicate logic with Zermelo-Fraenkel Set Theory

true, false, proposition, predicate

$\neg \wedge \vee \Rightarrow \forall \exists$

ϵ

10 axioms in ZFC

Any system of reasoning must have terms which are understood before the first definition.

Dependent type theories: Martin-Lof, CIC, HOTT,…

type, term

$\Gamma \vdash : \times \lambda \rightarrow$

a) Formation rules, b) introduction rules, c) elimination rules, d) computation rules

Predicate logic with Zermelo-Fraenkel Set Theory

true, false, proposition, predicate

$\neg \wedge \vee \Rightarrow \forall \exists$

ϵ

10 axioms in ZFC

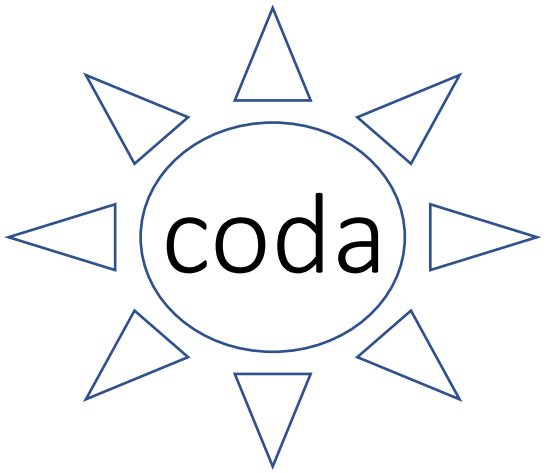
Any system of reasoning must have terms which are understood before the first definition.

Dependent type theories: Martin-Lof, CIC, HOTT,…

type, term

$\Gamma \vdash : \times \lambda \rightarrow$

a) Formation rules, b) introduction rules, c) elimination rules, d) computation rules



finite sequence

:

Axiom of definition

a) A data is a finite sequence of codas.
b) A coda is a pair of data.

We will use a small natural algebra of data:

A B denotes the concatenation of data A with data B.
A : B denotes the data consisting of a single coda made from A and B.

( ) - the empty sequence of codas.
(:) - the data consisting of one coda made from the pair () and ().
(:) ((:):(:(:))) ((:):(:)) - a sequence of three codas.

This is *pure data*. Pure data is "finite sequences of nothing."

- A definition is a partial function from coda to data.
- A valid context is a definition that has been accepted as valid.

Given a context δ, equality of data is defined by

$$\left.\begin{array}{l} A \;\; B \;=\; \delta\,(A) \;\; B \;=\; A \;\; \delta\,(B) \\ A : B \;=\; \delta\,(A) : B \;=\; A : \delta\,(B) \end{array}\right\}$$

These are the "deduction rules" in the sense of a formal system.

The Axiom of Definition:

a) The empty definition is a valid context.

b) Given a valid context δ, if definition δ' has a domain which is disjoint from the domain of δ, then $\delta \cup \delta'$ is also a valid context.

There is only one axiom.

The 0 and 1 bit, bit sequences, byte sequences, functions, variables, categories, morphisms, types, theorems, language expressions,… are represented by pure data.

domain	coda	display	meaning
()	(:)	◎	Atom
(:)	(:):	**0**	The 0 bit
(:)	(:):(:)	**I**	The 1 bit
0	(**0**:..*sequence of bits*..)	**00I00II0**	Sequence of 8 bits
I	(**I**:..*sequence of bytes*..)	hello	Sequence of bytes

In a context δ, if δ(c) = c for coda c, then c is called an **atom**. In the above, the "hydrogen atom" (:), individual bits, bit sequences and byte sequences are all atoms, unchanged by any future definition. If data contains one or more atoms in it's sequence, it is **atomic data**.

```
pure : Hello World
```

```
(((:):(:)):(((:):(:):((:):(:))((:):(:)):((:):(:))((:):(:))((:):(:)):((:):(:))((:):(:))((:):(:)):(((:):(:):((:):(:))((:):(:)):((:):(:))((:):(:))((:):(:)):((:):(:))((:):(:))((:):(:)):):)
((:):)((:):(:))((:):(:)):((:):(:)))(((:):(:):((:):(:))((:):(:)):((:):(:))((:):(:))((:):(:)):((:):(:))((:):(:))((:):(:)):(((:):(:):
((:):)((:):(:))((:):(:)):((:):(:))((:):(:)):((:):(:))((:):(:))((:):(:)):(((:):(:):((:):(:))((:):(:)):((:):(:))((:):(:))((:):(:)):((:):(:)):
(:))((:):(:))((:):(:)))) (((:):(:)):(((:):(:):((:):(:))((:):(:)):((:):(:))((:):(:))((:):(:)):((:):(:))((:):(:))((:):(:)):(((:):(:)))
(((:):(:):((:):(:)):((:):(:))((:):(:)):((:):(:))((:):(:))((:):(:))((:):(:))((:):(:)):(((:):(:):((:):(:))((:):(:)):((:):(:))((:):(:)):((:):(:)):
(:))((:):(:)):((:):(:)):((:):(:))((:):)):(((:):(:):((:):(:))((:):(:)):((:):(:))((:):(:))((:):(:)):((:):(:))((:):(:))((:):)):(((:):(:):
((:):)((:):(:))((:):(:))((:):(:)):((:):(:))((:):(:))((:):(:)):((:):(:))((:):(:)):)))
```

Typical definitions:

- Doing nothing to “input”
  - (pass A:B) -> B
- Null data for any A,B
  - (null A : B) -> ()
- Reversing the order of data
  - (rev : A B) → (rev : B) (rev : A)
  - (rev : a) → a if a is an atom
  - (rev : ()) → ()
- Typical combinatorics
  - (ap A : B C) → (ap A:B) (ap A : C)
  - (ap A : b) → A:b if b is an atom
  - (ap A : ()) → ()
- The natural numbers
  - (nat : n) → n (nat: n+1)

coda	schematic result
`ap` A:b1 b2 b3	(A:b1) (A:b2) (A:b3)
`app` a1 a2 a3:B	(a1:B) (a2:B) (a3:B)
`ap2` a a1 a2 a3:B	(a a1:b1) (a a2:b2) (a a3:b3)
`aps` A:b1 b2 b3	(A b1 : (A b2 : (A b3 : b4)))
`apby` 2 A:b1 b2 b3 b4...	(A:b1 b2) (A:b2 b3)...
`apif` A:b1 b2...	(b1..if (A:b1)) (b2..if (A:b2))...

Data is always a **finite** sequence

```
#
#   Trying again, we have a modified partial evaluation of (nat:0), the natural numbers.
#
nat : 0
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 (nat:19)
```

```
#
#   You can use step to how this gets evaluated in sequence.
#
step 10 : nat : 0
```

```
[ 0] 0 (nat:1)
[ 1] 0 1 (nat:2)
[ 2] 0 1 2 (nat:3)
[ 3] 0 1 2 3 (nat:4)
[ 4] 0 1 2 3 4 (nat:5)
[ 5] 0 1 2 3 4 5 (nat:6)
[ 6] 0 1 2 3 4 5 6 (nat:7)
[ 7] 0 1 2 3 4 5 6 7 (nat:8)
[ 8] 0 1 2 3 4 5 6 7 8 (nat:9)
[ 9] 0 1 2 3 4 5 6 7 8 9 (nat:10)
[10] 0 1 2 3 4 5 6 7 8 9 10 (nat:11)
```

..but this is not a meaningful limitation

```
#
#    What if we ask for "the natural numbers in reverse order?"...
#
rev : nat : 0
```

```
(rev:(nat:17)) (rev:16) 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

Logic

Within coda, all items of mathematics or computing are merely different kinds of data as we have defined. This means, roughly speaking, that any mathematical question will be of the form

Is data A equal to data B?

Since the answer to this question A=B is also data, the answer to this question should be deducible from the concrete data (= A : B) suggesting that "logic" should be the coarsest classification of data in general.

- Data is true if it is empty.
- Data is false if it is atomic.
- Data is undecided otherwise.

Notice that logic is directly visible in the duality between () and (:) in pure data. Since “everything is made of (:)”, a definition

$$(:) \rightarrow (\ )$$

would cause all data to collapse to the empty data. The alternative

$$(:) \rightarrow (:)$$

Makes (:) into an atom which can never be modified by a future definition.

This is not as unfamiliar as it might appear. Any of the usual binary Boolean operators has a corresponding definition in coda. Each of the 16 usual binary Boolean operations has a corresponding definition in Coda:

A	B	XOR(A,B)
T	T	F
T	F	T
F	T	T
F	F	F

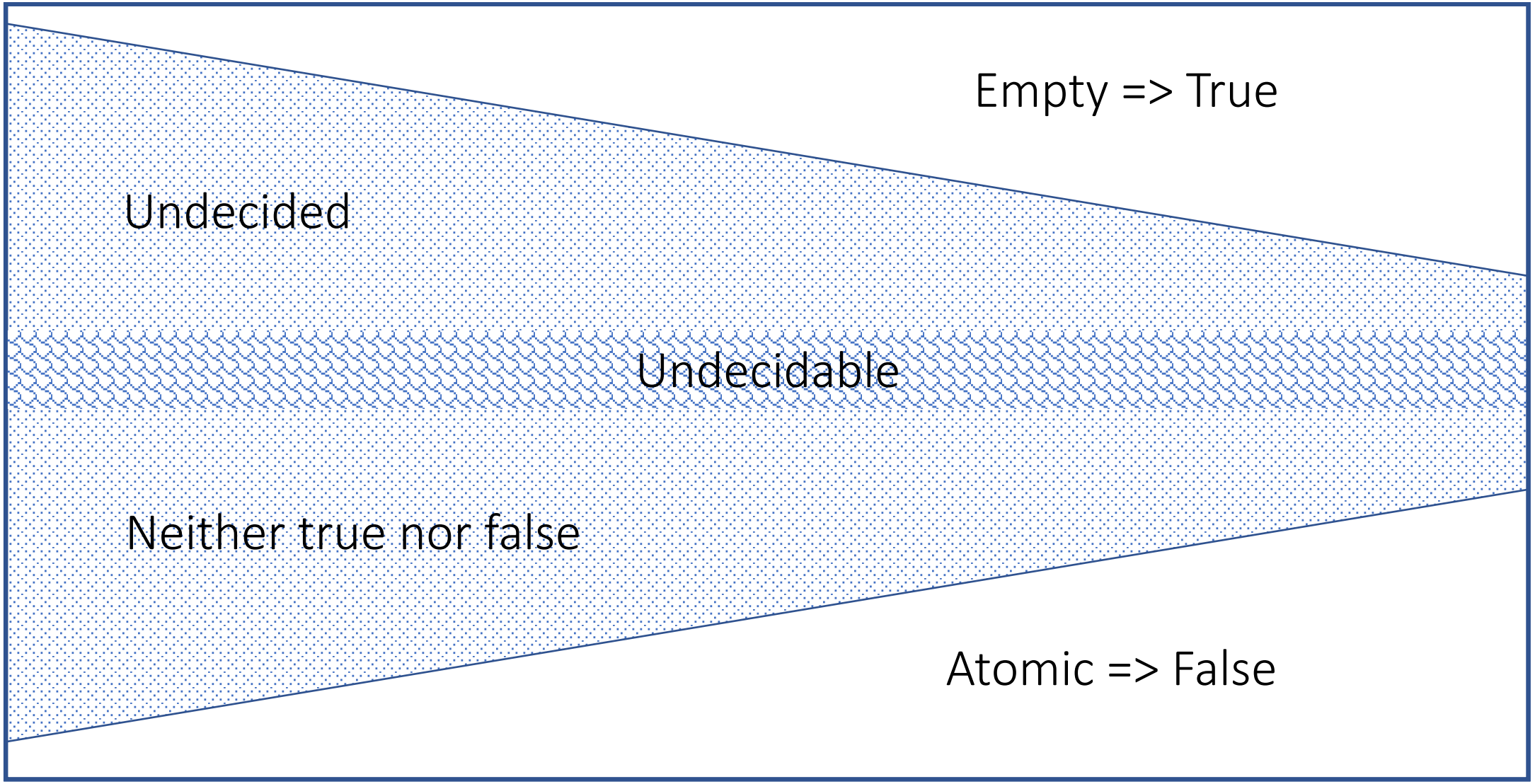
(XOR A : B) → XOR(A,B) with (:) in place of F and () in place of T, provided that A and B are either atomic or empty.

Since any added definitions must avoid the domain of any existing definition, being an atom is permanent.

- True data is “always true” independent of future definitions.
- False data is “always false” independent of future definitions.
- Undecided data may become true or false with future definitions

Undecided data like (foo:bar) or (?:X) are “variables” in the sense that they may receive values in future definitions. If data A=B is undecided, the undecided value presumably provides insight into why A is not equal to B. It may, for instance, inspire a new definition.

Some undecided data remains undecided no matter what future definitions are chosen. Such data are called undecidable. This is the Godel phenomenon, which we will see later.



Values, variables, functions, propositions, theorems, definitions, “spaces”, “morphisms”… live in the space of pure data

Language

As with predicate calculus with ZFC and dependent type theory, coda has a language which specifies data valued expressions. Unlike ZFC and dependent type theory, however, the whole language is merely one more definition just as we have defined it. It is a partial function from coda to data acting on codas of the form ({…some sequence of bytes…} A : B) where A and B can be any data.

- $(\{x \quad y\}\ A\ : B) \rightarrow (\{x\}\ A : B) \quad (\{y\}\ A : B\ )$
- $(\{x : y\}\ A : B) \rightarrow (\{x\}\ A : B) : (\{y\}\ A : B)$
- $(\{A\}\ A : B) \rightarrow A$
- $(\{B\}\ A : B) \rightarrow B$
- …plus a few more like these complete the language

There are several points to notice…

Language expressions mix freely with any other data

```
#
#   You can use "step" to show how this get's evaluated in more detail.
#   Each step below is made by application definitions.  You can see
#   that Coda freely mixes data and source code as it computes.
#
step : {first A : B} 2 : a b c d e
```

```
[0] (({{first A : B} 2 }:):({ a b c d e}:))
[1] (({{first A : B} 2}:):({a b c d e}:))
[2] (({{first A : B}}:) ({2}:):({a}:) ({b c d e}:))
[3] (({first A } 2:a ({b}:) ({c d e}:)):({ B} 2:a ({b}:) ({c d e}:)))
[4] (({first A} 2:a b ({c}:) ({d e}:)):({B} 2:a b ({c}:) ({d e}:)))
[5] (({first} 2:a b c ({d}:) ({e}:)) ({A} 2:a b c ({d}:) ({e}:)):a b c ({d}:) ({e}:))
[6] a (first 1:b c d e)
[7] a b (first 0:c d e)
[8] a b
```

```
#
#   You can use "def" to make your own definitions.  This one defines a new
#   command "mydef" which gets the first n elements from the reversed input.
#
def mydef : {first A : rev : B}
mydef 2 : a b c d e
```

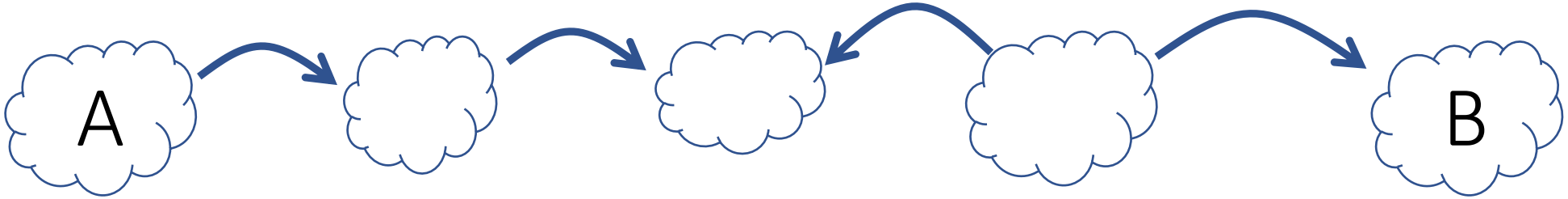
```
e d
```

Note…

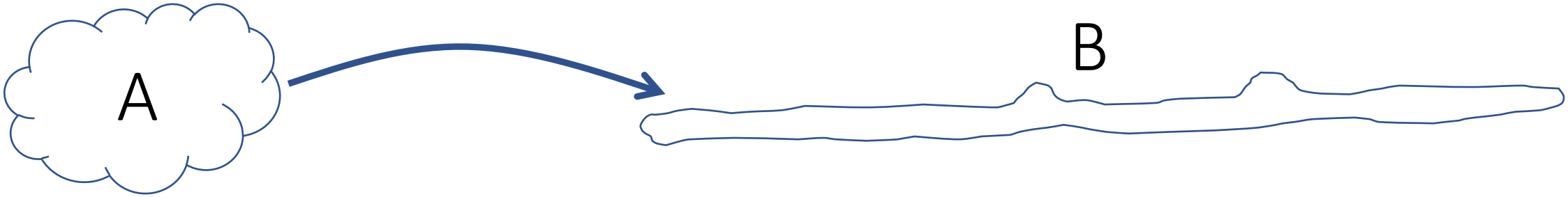
- Because the whole language is merely one more definition like any other, we do not need to define a syntax with additional axioms or rules.
- In predicate logic with ZFC or dependent type theory, one needs to define a syntax to distinguish meaningful sentences from nonsense sentences like "$\exists\exists xyy\forall\forall\forall$." This is not actually necessary in coda because every byte sequence is valid coda source code. There is no such thing as a syntax error.
- The mapping $s \to \{s\}$: is an onto mapping from finite byte sequences s to data. From pure data, it is also clear that there is an invertible mapping from data to coda source code.

Proof and Computation

Any sequence $A=D_1=D_2=\ldots=D_n=B$ constitutes a proof that data A is equal to data B.



"Computation of A" just means proving that A=B where B is suitably "flattened" for terminal output



Since these steps only apply definitions, every computation is a proof and every proof is a computation.

Echoing Russell & Whitehead, here is the complete proof that 1 + 1 = 2

```
step : sum n : 1 1
[ 0] (({sum n }:):({ 1 1}:))
[ 1] (({sum n}:):({1 1}:))
[ 2] (({sum}:) ({n}:):({1}:) ({1}:))
[ 3] ({(Sum:A) A : B} n:1 1)
[ 4] (({(Sum:A) A } n:1 1):({ B} n:1 1))
[ 5] (({(Sum:A) A} n:1 1):({B} n:1 1))
[ 6] (({(Sum:A)} n:1 1) ({A} n:1 1):1 1)
[ 7] (({Sum:A} n:1 1) n:1 1)
[ 8] ((({Sum} n:1 1):({A} n:1 1)) n:1 1)
[ 9] (({ap put A } n:1 1):({ aps int_add : pre 0 : get A : make A : B} n:1 1))
[10] (({ap put A} n:1 1):({aps int_add : pre 0 : get A : make A : B} n:1 1))
[11] (({ap} n:1 1) ({put A} n:1 1):(({aps int_add } n:1 1):({ pre 0 : get A : make A : B} n:1 1)))
[12] (ap ({put} n:1 1) ({A} n:1 1):(({aps int_add} n:1 1):({pre 0 : get A : make A : B} n:1 1))) (ap ({put} n:1 1) ({A} n:1 1):)
[13] (ap put n:(({aps} n:1 1) ({int_add} n:1 1):(({pre 0 } n:1 1):({ get A : make A : B} n:1 1)))) (ap put n:)
[14] (ap put n:(aps int_add:(({pre 0} n:1 1):({get A : make A : B} n:1 1)))) (ap put n:)
[15] (ap put n:(aps int_add:(({pre} n:1 1) ({0} n:1 1):(({get A } n:1 1):({ make A : B} n:1 1))))) (ap put n:)
[16] (ap put n:(aps int_add:0 (({get A} n:1 1):({make A : B} n:1 1)))) (ap put n:)
[17] (ap put n:(aps int_add:0 (({get} n:1 1) ({A} n:1 1):(({make A } n:1 1):({ B} n:1 1))))) (ap put n:)
[18] (ap put n:(aps int_add:0 ({ap get1 A : B} n:(({make A} n:1 1):({B} n:1 1))))) (ap put n:)
[19] (ap put n:(aps int_add:0 (({ap get1 A } n:(({make} n:1 1) ({A} n:1 1):1 1)):({ B} n:(({make} n:1 1) ({A} n:1 1):1 1))))) (ap put n:)
[20] (ap put n:(aps int_add:0 (({ap get1 A} n:({(Make:A) A : B} n:1 1)):({B} n:({(Make:A) A : B} n:1 1))))) (ap put n:)
[21] (ap put n:(aps int_add:0 (({ap} n:((({(Make:A) A } n:1 1):({ B} n:1 1))) ({get1 A} n:((({(Make:A) A } n:1 1):({ B} n:1 1))):(({(Make:A) A } n:1 1):({ B} n:1 1))))) (ap put n:)
[22] (ap put n:(aps int_add:0 (ap ({get1} n:((({(Make:A) A} n:1 1):({B} n:1 1))) ({A} n:((({(Make:A) A} n:1 1):({B} n:1 1))):((({(Make:A) A} n:1 1):({B} n:1 1))) (ap ({get1} n:((({(Make:A) A} n:1 1):({B} n:1 1))) ({A} n:((({(Make:A) A} n:1 1):({B} n:1 1))):))) (ap put n:)
[23] (ap put n:(aps int_add:0 (ap get1 n:(({(Make:A)} n:1 1) ({A} n:1 1):1 1)) (ap get1 n:))) (ap put n:)
[24] (ap put n:(aps int_add:0 (ap get1 n:(({Make:A} n:1 1) n:1 1)) (ap get1 n:))) (ap put n:)
[25] (ap put n:(aps int_add:0 (ap get1 n:((({Make} n:1 1):({A} n:1 1)) n:1 1)) (ap get1 n:))) (ap put n:)
[26] (ap put n:(aps int_add:0 (ap get1 n:(({ap put A: ap nat1 : B) (has A:B) (ap app (Extra:A):B) } n:1 1)) (ap get1 n:))) (ap put n:)
[27] (ap put n:(aps int_add:0 (ap get1 n:(({ap put A: ap nat1 : B) (has A:B) (ap app (Extra:A):B)} n:1 1)) (ap get1 n:))) (ap put n:)
[28] (ap put n:(aps int_add:0 (ap get1 n:(({ap put A: ap nat1 : B)} n:1 1)) (ap get1 n:(({has A:B) (ap app (Extra:A):B)} n:1 1)))) (ap put n:)
[29] (ap put n:(aps int_add:0 (ap get1 n:({ap put A: ap nat1 : B} n:1 1)) (ap get1 n:) (ap get1 n:(({has A:B)} n:1 1)) (ap get1 n:(({ap app (Extra:A):B)} n:1 1)))) (ap put n:)
[30] (ap put n:(aps int_add:0 (ap get1 n:(({ap put A} n:1 1):({ ap nat1 : B} n:1 1))) (ap get1 n:) (ap get1 n:({has A:B} n:1 1)) (ap get1 n:) (ap get1 n:({ap app (Extra:A):B} n:1 1)) (ap get1 n:))) (ap put n:)
[31] (ap put n:(aps int_add:0 (ap get1 n:(({ap} n:1 1) ({put A} n:1 1):({ap nat1 : B} n:1 1))) (ap get1 n:) (ap get1 n:(({has A} n:1 1):({B} n:1 1))) (ap get1 n:) (ap get1 n:(({ap app (Extra:A)} n:1 1):({B} n:1 1))) (ap get1 n:))) (ap put n:)
[32] (ap put n:(aps int_add:0 (ap get1 n:(ap ({put} n:1 1) ({A} n:1 1):(({ap nat1 } n:1 1):({ B} n:1 1)))) (ap get1 n:(ap ({put} n:1 1) ({A} n:1 1):)) (ap get1 n:(({has} n:1 1) ({A} n:1 1):1 1)) (ap get1 n:) (ap get1 n:(({ap} n:1 1) ({app (Extra:A)} n:1 1):1 1)) (ap get1 n:))) (ap put n:)
[33] (ap put n:(aps int_add:0 (ap get1 n:(ap put n:(({ap nat1} n:1 1):({B} n:1 1)))) (ap get1 n:(ap put n:)) (ap get1 n:({ap has1 A : B} n:1 1)) (ap get1 n:) (ap get1 n:(ap ({app} n:1 1) (({Extra:A)} n:1 1):1)) (ap get1 n:(ap ({app} n:1 1) (({Extra:A)} n:1 1):1)))) (ap put n:)
[34] (ap put n:(aps int_add:0 (ap get1 n:(ap put n:(({ap} n:1 1) ({nat1} n:1 1):1 1))) (ap get1 n:(ap put n:)) (ap get1 n:(({ap has1 A } n:1 1):({ B} n:1 1))) (ap get1 n:) (ap get1 n:(app ({Extra:A} n:1 1):1)) (ap get1 n:) (ap get1 n:(app ({Extra:A} n:1 1):1)) (ap get1 n:))) (ap put n:)
[35] (ap put n:(aps int_add:0 (ap get1 n:(ap put n:(ap nat1:1))) (ap get1 n:(ap put n:(ap nat1:1))) (ap get1 n:(({ap has1 A} n:1 1):({B} n:1 1))) (ap get1 n:) (ap get1 n:(app (({Extra} n:1 1):({A} n:1 1)):1)) (ap get1 n:) (ap get1 n:(app (({Extra} n:1 1):({A} n:1 1)):1)) (ap get1 n:))) (ap put n:)
[36] (ap put n:(aps int_add:0 (get1 n:(n:1)) (get1 n:(n:1)) (ap get1 n:(({ap} n:1 1) ({has1 A} n:1 1):1 1)) (ap get1 n:))) (ap put n:)
[37] (ap put n:(aps int_add:0 (if (= n:n):1) (if (= n:n):1) (ap get1 n:(ap ({has1} n:1 1) ({A} n:1 1):1)) (ap get1 n:(ap ({has1} n:1 1) ({A} n:1 1):1)))) (ap put n:)
[38] (ap put n:(int_add 0:(aps int_add:1 1 (ap get1 n:(if (= n:IIIOIOIIIOIOIIIIO):1)) (ap get1 n:) (ap get1 n:(if (= n:IIIOIOIIIOIOIIIIO):1)) (ap get1 n:)))) (ap put n:)
[39] (ap put n:(int_add 0:(int_add 1:(aps int_add:1)))) (ap put n:)
[40] (put n:2)
[41] (n:2)
```

This is actually proving a bit more. It is a proof that the data ({sum n: 1 1} :) is equal to the data (n:2).

Note…

1. Since each step is the application of definition, each computation is a valid proof and each proof is a computation.
2. This is different than the situation in dependent type theory with the Curry-Howard correspondence.
3. Since any definition can be applied at any point in the data, there is no unique way to “execute” data. There are different strategies for different purposes.
4. Although the the number of steps in the previous page may be unbounded, each step is the application of a definition, so each individual step cannot loop.

Spaces

The most natural way to identify a particular part of the enormous space of all pure data is to fix some data A and consider the data equal to

$$A : X$$

for any data X. To make the space "A" closed under sequences, we require

$$A : (A:X)\ (A:Y) = A : X\ \ Y$$

For any data X and Y. Any distributive data F can be thought of as a function from data X to data F:X. Suppose that we want F applied to data in A, F : (A:X) to end up in space "B". We can do that by requiring

$$F : A : X = B : F : X$$

For all data X. If so, we say that F is a **morphism** from space A to space B.

One ”space” of natural numbers: (n:0), (n:1), (n:2),…
Another: (m:0), (m:1), …

Familiar mathematical structure is nicest in equivalent sequential form, e.g.

sum n : 1 2 3 4 …rather than…. 1 + 2 + 3 + 4 with a binary operator
sort n : 4 3 99 1 …rather than…. Sorting with a binary operator

- type n
- sum n
- sort n
- prod n
- term n

Each of these is a “space”, each identifying the same collection of data: (n:0), (n:1), … but representing different mathematical structure.

- type n
- sum n
- sort n
- prod n
- term n

Each of these is a “space”, each identifying the same collection of data: (n:0), (n:1), … but representing different mathematical structure.

Consider distributive F between space (sum n) and space (sum m). We require

F : (sum n) : x_1 x_2 = (sum m) : F : x_1 x_2

this is $F(x_1 + x_2) = F(x_1) + F(x_2)$, so F preserves sums.

Consider distributive F between space (sort n) and space (sort m). We require

F : (sort n) : x_1 x_2 = (sort m) : F : x_1 x_2

this is equivalent to $x_1 \leq x_2 \Longrightarrow F(x_1) \leq F(x_2)$, so F preserves partial order.

Space	Action	Data	F*Space = Space*F
pass	A → A	All data	F distributive
null	A → ()	() only	F:X = ()
first	a A → a	Single atoms	F dependent only on the first atom
bool	A → () or (:)	True/false	F:X preserves bool:X
type n	A → (n:2) (n:1) (n:5)	Natural numbers	(type n) * F * (type n) for any F
sum n	(n:2) (n:1) (n:5) → (n:8)	Natural numbers	$F(n_1+n_2) = F(n_1)+F(n_2)$
prod n	(n:2) (n:1) (n:5) → (n:10)	Natural numbers	$F(n_1 \times n_2) = F(n_1) \times F(n_2)$
sort n	(n:2) (n:1) (n:5) → (n:1) (n:2) (n:5)	Natural numbers	$n_1 \leq n_2$ implies $F(n_1) \leq F(n_2)$
set n	(n:2) (n:1) (n:5) → equiv. classes	Equivalence classes	F preserves class
Sum n	$f_1\ f_2 \ldots f_n \rightarrow f_1 * f_2 * \ldots * f_n$	Linear n → n maps	F preserves composition
Sum	$f_1\ f_2 \ldots f_n \rightarrow f_1 * f_2 * \ldots * f_n$	Linear maps	F preserves composition
Space	$A \rightarrow S_1\ S_2\ S_3 \ldots S_n$	Spaces	Space*F*Space for any F
Mor	$f_1\ f_2 \ldots f_n \rightarrow f_1 * f_2 * \ldots * f_n$	Morphisms	Functorial

Compare “spaces” with categories in category theory

- Spaces have no dependence on sets or set theory.
- Unlike in set theory, any data F, G, H can be composed as $X \rightarrow F:G:H:X$ whether domains match with codomains or not.
- The categorical properties of morphisms is determined by the space. This is not true in standard category theory where, for instance, you can have a category of real vector spaces with ordinary, non-linear maps.
- Associativity happens a lot, due to concatenation of sequences being associative.
- A distributive morphism $X \rightarrow F:X$ is a sequence preserving function on the atoms of X. It is the coda version of an ordinary function.

There is an abstract theory of Spaces

Data is a space if

A : (A:X) (A:Y) = A : X Y

A space is idempotent if

A : A : X = A : X

A space is abelian if

A : X Y = A : Y X

A distributive space is a type

A : X Y = (A:X) (A:Y)

A morphism from space A to space B is distributive data F where

F : A : X = B : F : X

A space A has an anti-space B if

A : (A:X) (B:X) = A:

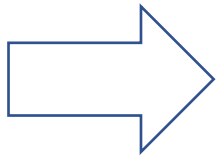
A : (B:X) (A:X) = A:

A pure group is a space with an anti-space.

Lemma: *A pure group G is a group*. Let set {(G:X)}, group multiplication (G:X) x (G:Y) → G:(G:X) (G:Y). Then (G:) is the identity and (G^{-1}:X) is the inverse of (G:X) where if G^{-1} is the promised antispace of G.

First steps in “Mathematical Machine Learning”

How can we distinguish some kind of mathematical objects…

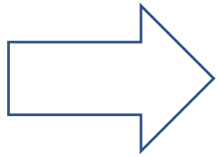


```
[5]: Odd = Generate.OddAtoms(10)
     for d in Odd: print(d)
```

```
⊙
⊙ ⊙ ⊙
⊙ ⊙ ⊙ ⊙ ⊙
⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙
⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙
⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙
⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙
⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙
⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙
⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙
```

```
[6]: Even = Generate.EvenAtoms(10)
     for d in Even: print(d)
```

…from some other kind of mathematical objects?



```
⊙ ⊙
⊙ ⊙ ⊙ ⊙
⊙ ⊙ ⊙ ⊙ ⊙ ⊙
⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙
⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙
⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙
⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙
⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙
⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙ ⊙
```

Search for a “classifying space” A…

```
[9]: S
```

```
[9]: eval: False, true: 0, false:0, undecided:0, codas:143, datas:11568
```

```
[10]: for i in range(20): print(S[i])
```

```
{B get1} counts
app rep
{B post} startswith
pass {tail B}
{apby B} null
equiv b
{B if} pre
{imply B} has1
pass {B 0}
endswith {B endswith}
A {last B}
{has1 B} collect
{B ap} nth1
rep {2 B}
has1 bin
{3 B} o
join split
o {if B}
{aps B} rep
{wrap B} ⊙
```

…such that A:X distinguishes even sequences of (:) from odd sequences of (:).

Three of these data successfully distinguish the two samples. One of these is…

aps not

Combinatorial operator that turns binary operations like (+ A : B) to a sequential sum like a+b+c+d…

Logical negation operation. not:X is atomic if X is empty and not:X is empty if X is atomic.

```
step : aps not : x y z w x
[0] (({aps not }:):({ x y z w x}:))
[1] (({aps not}:):({x y z w x}:))
[2] (({aps}:) ({not}:):({x}:) ({y z w x}:))
[3] (aps not:x ({y}:) ({z w x}:))
[4] (not x:(aps not:y ({z}:) ({w x}:)))
[5] (not x:(not y:(aps not:z ({w}:) ({x}:))))
[6] (not x:(not y:(not z:(aps not:w x))))
[7] (not x:(not y:(not z:(not w:(aps not:x)))))
[8] IOOIOIIIOOIIIO
⊙
```

Conclusions which hopefully generalize…

1. The solution is clever. aps not combines a combinatorial operator with a logical operator in a way that I did not think of beforehand.
2. The solution generalizes. The solution distinguishes the atomic parity of any data, not just sequences of (:).
3. A slightly modification: bool * aps not is a space, so we are "getting a mathematical structure for free."
4. The morphisms of the category bool * aps not are interesting. They are functions which preserve the atomic parity of data.

Is Mathematics Consistent?

1. We are claiming that empty data is “true” and atomic data is “false.”
2. No data can be both true and false.
3. Have we proved that coda is consistent? Does this contradict Godel’s 2d incompleteness theorem?

To see the answer, express “coda is consistent” in coda…

ap { XOR (coda:B) : (not:coda:B) } : allByteSequences:

Then the answer is…

To see the answer, express “coda is consistent” in coda…

ap { XOR (coda:B) : (not:coda:B) } : allByteSequences:

Then the answer is…

() () () () () () … (…undecided…)

Berry's Paradox

A paradox of G. G. Berry, reported by Bertrand Russell, is the following.

- Let N = The smallest positive integer not definable in less than twelve words.

The problem is this reasoning.

> Surely, for any positive integer N, N either is or isn't definable in less than twelve words. There is, thus, a smallest such integer. But if this integer exists, it **is** definable in less than twelve words. A contradiction.

To make a precise version of this in Coda, let's say that

- A byte string s **defines a positive number** n if `s:` in coda is equal to the character representation of the number n.

Then, this concrete Coda expression

```
aps int_max : ap {coda : B} : codes 52 : alphabet :
```

computes the maximum number definable in 52 or fewer characters. It helps to read the code from right to left.

1. `alphabet:` is a standard set of alphanumeric characters.
2. `codes 52 : alphabet :` is the finite, but very large, sequence of all strings in the input alphabet with 52 or fewer characters.
3. `ap {coda:B}` applies the coda compiler to each string, producing data as input to...
4. `aps int_max` computes the maximum integer input.

This is far to large a computation to do in practice, but the point, and the resolution of the paradox, is that the above expression actually is 51<52 characters long, so the full expression will appear again as input to `coda`, and the expression will, therefore, loop forever, even on an infinitely capable computer.

This is the resolution of the paradox. The question corresponds to data which is **undecidable**.

4]:
```
#
#    Just to demonstrate that the code above is real, reduce the alphabet to 1 2 3 and
#    and 52 to 2 and we get the correct maximum answer: 33.
#
aps int_max : ap {coda : B} : codes 2 : 1 2 3
```

```
33
```

Yablo's Paradox

Stephen Yablo (of MIT) has pointed out that paradoxes do not require self-referential statements. His example is as follows.

- Let Y_1 = true if all Y_i are false for i > 1
- Let Y_2 = true if all Y_i are false for i > 2
- Let Y_3 = true if all Y_i are false for i > 3
- ...

> Suppose that Y_n is true for some n. Then all sentences greater than n are false. But that means that all sentences greater than Y_{n+1} are also false, which means that Y_{n+1} is true. That is a contradition, so all the Y_n must be false. But if all the Y_n are false, Y_1, for instance, is true. Contradiction.

To compute this one in Coda, we can define

`def Yablo : {ap not : Yablo : skip 1 : nat : B}`

so that (Yablo : n) is true if all (Yablo:n+1) (Yablo:n+2)... are not true. Here (nat:k) are the natural numbers starting at k, skip 1 skips the first one and {ap not : Yablo : skip 1 : nat : B} : 5, for instance is true if all the Yablos greater than or equal to 5 are false.

```
def Yablo : {ap not : Yablo : skip 1 : nat : nat1 : B}
```

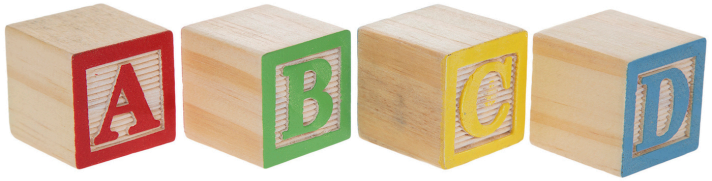
```
#
#   Of course, we cannot fully evaluate Yablo:1.  This is not a proof, but a partial evaluation
#   of Yablo:1 appears to be recursing forever, which would be as expected.
#
step 5 : Yablo : 1
```

```
[0] ({ap not : Yablo : skip 1 : nat : nat1 : B}:1)
[1] (({ap not }:1):({ Yablo : skip 1 : nat : nat1 : B}:1))
[2] (({ap}:1) ({not}:1):(({Yablo }:1):({ skip 1 : nat : nat1 : B}:1)))
[3] (ap not:(({ap not }:(({skip 1}:1):({nat : nat1 : B}:1))):({ Yablo : skip 1 : nat : nat1 : B}:(({skip 1}:1):({nat : nat1 : B}:1))))) (ap not:)
[4] (ap not:(({ap not}:(skip 1:(({nat}:1):({nat1 : B}:1)))):({Yablo : skip 1 : nat : nat1 : B}:(skip 1:(({nat}:1):({nat1 : B}:1)))))) (ap not:)
[5] (ap not:(({ap}:(skip 1:1 (nat:2))) ({not}:(skip 1:1 (nat:2))):(({Yablo }:(skip 1:1 (nat:2))):({ skip 1 : nat : nat1 : B}:(skip 1:1 (nat:2)))))) (ap not:)
```

Unlike the previous examples, it would take some more effort to prove that (Yablo:1) is undecidable, but this is clearly to be expected.

Summary

1. Axiomatic formal system aimed at mathematics in general.
    1. Pure data with natural algebra A B, A:B.
    2. Definitions added to a “context”.
    3. Only one axiom: The Axiom of Definition.
    4. Mathematical objects are different types of pure data.
    5. Context determines equality which determines mathematical meaning.
2. Internal Logic
    1. True/false/undecided ⇔ empty/atomic/neither data.
3. Internal language
    1. Language is a definition like any other.
    2. Language expressions freely mix during “evaluation”.
4. Proof and computation
    1. All proofs are computations. All computations are proofs. No Curry-Howard correspondence.
5. Abstract spaces
    1. Analogues of types in type theory and categories in category theory naturally appear as “spaces”.
    2. There appears to be an interesting abstract theory of spaces.
    3. First tries at “Mathematical machine learning” yields interesting results.
6. Global issues
    1. Direct computation of “paradoxes” gains confidence in coda logic and makes it clear that there is not going to be a simple criteria that avoids such ”paradoxes” in classical Mathematics.
    2. Interesting applications, e.g. “Mathematical machine learning”.



See Saul Youssef on github for software, jupyter notebooks etc.