The World as Information

By Bruce Long
May 2025

Abstract
Here we introduce the idea that the world can be seen as information. Some mathematics of an “infon” are discussed then we show how to make it practical via the “Proteus” notation to model information, language and reality.

Part 1

The Philosophy: A better way to conceptualize the World

Introduction

The history of philosophy is full of attempts at theories to describe reality. These range from theories that can't even be simulated to those that simulate as worlds much like the game Minecraft. The *Process Philosophy* of Alfred Whitehead was almost good enough to make fully mathematical. His "almost successful" attempt to do so contributed to the formation of 20th Century mathematics. And when we use the equations of modern mathematics to represent the World, we typically measure things like matter and motions of matter. So the corresponding philosophy is called *materialism* or even *materialistic reductionism*.

Now, something exciting, or at least interesting is happening. New concepts from information theory and computer science are coming into focus. These are concepts that would be nearly impossible for the pre-computer philosophers to access. Yet today it is obvious to every kid that information structures inside a computer can be made into complex 3D worlds.

A question arises, In materialism we store information in the position or other states of matter. The material is considered real and the information stored there is considered a construct. Could there be benefits to trying it the other way round? That is, could everything be information and when an information structure observes another it can appear to be matter in motion? This philosophy is called *information realism*.

In this series of articles we look at how modeling information structure can tell us a lot about the world, including some things that materialism cannot.

In this first article we briefly discuss such a world view in philosophical terms. But philosophy without math is woo-woo. So the next few articles will develop a concrete way to talk about information structure. In this series I will try not to use math-speak but a little work through the details will be surprisingly useful.

I find that math without code is impractical. So we will end the series with some practical articles that introduce actual models and code that makes them work.

Is Everything Information?

The question “Is everything information?” Is a question in philosophy about something called an *ontology*. We can think of an ontology as a list of concepts that we consider to be “complete” in some way. For example, the website Yahoo has a hierarchical list of categories that they use to categorize every web-page. They call this their ontology. Similarly, the military has an ontology they maintain to describe military situations. In fact, everyone has an ontology in their head that they use to categorize their experiences.

Metaphysical Ontologies

A Metaphysical Ontology is an ontology where completeness means it can represent everything. And I really mean everything. Not just things like chickens and chairs, walking and beauty, but also things like numbers and math, consciousness and choice.

Naive Ontologies

When we are young, before we learn science, our ontology often consists of a long list of object types, action types etc. Like *bird* and *flying* and *flying fast*. Soon we learn that these concepts, or at least many of them are relative or that

the categories don't always apply. For example, the concept of an iPhone didn't exist in the past.

Old Ontologies

An ontology from the past held that everything was a *substance*. Think of a substance as being like cheese. If you cut cheese in half, both halves are still cheese. That is unlike a bike or a television where if you cut them in half, both sides are not bikes or televisions. In fact, the whole thing will no longer be a functioning bike or television. In the substance ontology things made out of substances are not fundamental and don't belong in the ontology. The philosopher Descartes theorized that there were two kinds of substances: material substances like cheese and the substance God is made of.

Ontological Materialism

Today, a common ontology is that everything is made of a list of subatomic particles such as electrons and photons. So cheese isn't a substance because as you repeatedly cut it in half you eventually get to atoms, not cheese. Interestingly, newer descriptions of subatomic particles involve how they transmit information to each other.

Another formulation given today's physics would hold that there are about 18 things in reality and they are all fields that span the whole universe. Subatomic particles arise as waves in these fields. Again, it's interesting that these waves are often represented as *quantum information*.

These physics based ontologies can be called material reductionism. They have the problem that they cannot account for such things as numbers, which philosophers have had to classify as “abstract entities”. Nor can material reductionism account for consciousness or meaning and value. Actually, it cannot really even account for macroscopic objects like cheese or chickens. Much less love or emotions.

There is actually a third ontology that physicists use. They use it to describe both particles and fields, *and* even bigger things like the masses and positions of projectiles. This is the ontology of *states*.

States and information

Everything in physics — *everything*, and indeed in other scientific fields, is represented in terms of states. Even space and time are considered states.

Now there is a tiny equation in computer science that relates states to information. In a non-general form we have:

```
2 states = 1 bit of information
```

Or:

```
256 states = 1 byte of information
```

Also, 1 byte = 8 bits. So the conversion from states to information isn't linear.

So states and information are the same thing but under a different unit of measure.

When our brain is using a naive or a simple material ontology we try to classify information in terms of objects. That is, objects exist and their position (or other state) can store information. But when we can wrap our head around it, it works better the other way around.

Some benefits of information realism

Parts of this series will be a little technical and you might find yourself wondering why you should wade through them. So I want to mention a few benefits that you can remind yourself of when you're in the middle of it.

1. No Abstract Entities Needed

In current philosophy, namely material reductionism, there is a problem accounting for things like numbers. If everything is made of matter (or energy), what are numbers made of? You can wade into that debate by googling "Abstract entities". But with information realism, it's not a mystery. Numbers are just a type of information. And as we will see, so are other kinds of abstract entity such as concepts and words.

Interestingly, this also solves another philosophy problem which is the divide between math or logic statements and the world. More on this in a future article.

2. Improved Logic

Many have noticed, in the last few decades, that traditional logic isn't that effective for real world argumentation. Logic that takes statements that are either true or false and deduces other statements that are true or false is

extremely unwieldy. As easily as such logic can be used to both prove and disprove the existence of a god, it can be used to bolster any side of political arguments.

With the theory of information realism we can instead take any set of information pieces and deduce other information. For example, from the information in a particular photograph, perhaps one could deduce that Bob was in Hawaii with his family over the holidays. Imagine trying to use traditional logic to explain how to correctly interpret photographs. It's not worth doing.

On the other hand, specifying how information flows from a scene, via photons, into a camera lens and into a jpg file, along with some shape information about our world (such as that encoded in 3d video games), and a few other things such as who took the picture, and so on, such an inference about Bob would be computationally trivial.

Part of the reason information based logic works better is because in 20th Century logic, all the relevant 'if' statements have to be spelled out. And there can be an unbounded number of them. But if you know the information structure of a situation, a possibly unbounded number of 'if' statements can be

deduced. So it doesn't take a huge number of rules to do complex things because a small structural specification can generate the huge number of rules.

The Levels-of-being Problem

Problems that involve inferring what will happen when we put a bunch of parts together can easily be solved by thinking about them. If I describe how to construct a bicycle from some wheels, tube's, a chain, etc. Almost everyone could figure out that it's a bike and that it would work as a bike. Or not. If I told you that this bike does not have a chain or anything else connecting the pedals to a wheel you would instantly know that it won't work. It's not even a hard calculation. But 20th century math and logic cannot make such inferences. Material reductionism, the idea that we reduce things to their component parts and so on until we reach atoms or subatomic particles doesn't work.

But our brains must be using information reductionism internally because with information reductionism it's pretty easy.

Suppose I have a bike on a rack and you are nearby but can only see the rear wheel. If it starts to spin you know something about the pedals even though

you cannot see them. In other words, information about the pedal state of motion flows through the gears, into the chain and so on into the rear wheel. You receive the information from light that reflects off the wheel and thus about the chain and about the pedals and perhaps even that I am standing there turning the pedals.

Now suppose I had told that same story but instead of talking about pedals and chains I referred to the materials they are made of. For example, I am moving some plastic wrapped over metal (the pedals) while cause atoms in a metal lattice to send that information from atom to atoms through the lattice and so on through the metal links in the chain, through the aluminum and rubber of the wheel and so on.

Notice that from the information perspective it doesn't change the picture. The same information is flowing in the same structure and can be used to make the same inferences. A lower level-of-being is just a more detailed example of how the information was structured but the overall flow is the same. In fact, as long as the information from seeing the wheels is linked to the pedals correctly, the description does not even need to refer to the chain.

Perhaps the bike doesn't even use a chain. Maybe it uses gears or something else. It’s still a bike because the information flow has the “bike” structure.

With what is to come we will be able to define and calculate this kind of thing with software.

What it will look like

Before embarking on this journey I want to give a brief description of what we are constructing so that you can recognize what we are making and won't wonder why the heck are we talking about this.

Think about how programming languages represent classes. A class in a programming language can represent and simulate essentially anything. Physical objects, numbers, a 3d game world. Anything. There are 3 aspects of this we are interested in. 1) how they represent the state of the system at an instant, 2) how they represent the way that state changes over time, and 3) how they facilitate interfacing the objects of the class to other objects.

The first aspect: classes represent the state of an object by listing its sub-parts as member variables. Those sub-parts have sub-parts as well with the hierarchy terminating in numbers like an int or in characters or even bits.

A good example of storing the state of something at an instant is the JavaScript spin-off JSON. JSON is a notation that can represent the state at an instant of small systems.

There are a few things we will need to fix with JSON. One is that we need to be able to represent large lists of states using expressions. In JSON if you wanted to refer to the states of a trillion atoms you would have to list each one. We will need to be able to say something like There are a trillion atoms that are roughly organized like <some expression>. It obviously should not try to allocate all trillion of them.

JSON can store the state of an object at an instant but it cannot represent how the state will change over time.

The second aspect: representing how states change over time. Typical programming languages explicitly spell out how the object will change using member functions that encapsulate state updates. The functions have a list of steps the computer should take to calculate the next state.

The problem here is that state evolution is given imperatively. We need to modify this so that state changes are represented declaratively in such a way that we can traverse changes backwards and forwards to make inferences.

In addition to representing state evolution declaratively, we need to be able to represent objects that have many things going on at the same time. For example, a bike may be being pulled down by gravity, held upright by balancing movements and pushed forward by the rear wheel's interaction with the ground. All the while, its parts are holding themselves together and their atoms are doing atomic things. So the objects need to be more like 3d video games objects.

The third aspect: programming language classes use public member functions to specify how they can interface to other objects.

Some goals for a language to describe reality

It might be easy to hack JSON to make a language that met the above descriptions. But how would we know that the set of inferences that could be made from such models is complete? In fact it certainly would not be. For example, it wouldn't be able to refactor models to obtain different perspectives. For example, in that language everything is an object. But in reality, not everything is an object.

We want the features of what we described but it should be smaller and more mathematical (at least at the low level) to ensure we get the theory right.

What we want here is a description of very low level state systems that can be combined through operators that are similar to multiplication and addition. And through such combinations, we can build up to descriptions of anything.

By having the way state systems connect and interact be described like arithmetic we can do algebraic like manipulations to make inferences about what will happen, or what did happen given evidence.

Part 2

The Math: The low-level components of information structure

Welcome to Part 2 of our journey to know how to represent the World in all its complexity by describing the information structure of state systems. To do this we will talk about a low-level component of structured information that can be combined with other ones using operations related to multiplication and addition. We will be able to combine these or take them apart in order to construct models of simple or complex systems. In advanced mathematics there is a similar concept called Group Theory. The name Group is only vaguely related to the regular meaning of the word. There are groups that represent every possible structure that doesn't lose information. For example, numbers can be defined as a group. There is also a group that mirrors the structure of a Rubix Cube. Groups cannot do what we need here. For one thing, groups can describe the possible states of something and how they might change, but they do not represent what state a system is in. It is like saying what a Rubix Cube is but not being able to tell what state a particular cube is in. Also, the Rubix Cube group says nothing about a 6 sided cube where, when solved, each side is a different color. But perhaps the most important issue is that it is very awkward to combine a bunch of groups to

make something really complex. It is too hard to represent something like an automobile with group theory. And forget representing something like democracy or a real legal system. In future parts of this series we will describe a rubix cube and how it changes, including that it can be taken apart. It will have colors and be a cube. In fact, there will be enough detail that the software can draw it.

This is my favorite of the articles as it provides the foundation for all that is to come. However, it is also the most abstract. So while this does provide the basis for the rest of the series, none of the other articles require an understanding of this one. The advantage of understanding this article is that most of what is to come will not seem like it is pulled out of a hat. In the other articles of the series I give little justification for why something is done a certain way — I just present it. If you are comfortable with that then this article can be skipped.

Chapter 1: Terminology, notation and method

This chapter gives names and notation to things anyone who has programmed a computer likely knows intuitively. So, for example, I'm not going to formally define information or mention Claude Shannon.

Infons

Our foundational components are actually just pieces of information. Indeed, we will be talking at length about pieces of information. Without a word for a piece of information it will be cumbersome. So I follow the mathematician Keith Devlin in using the term *infon* to mean a "piece of information". The term does not imply any magic or that information is a type of particle. It's just a shorthand for "piece of information"

Infons have a size and a value. For example we might have an infon of 256 states and it is in state 200. The size is 256 and the value is 200. Let us notate infons like this:

```
*256 +200
```

So in general we write:

```
*<size> +<value>
```

If we do not know the value or size we can write an underscore ('_'). So

```
*_ +20
```

is an infon with at least 21 states and it is in state 20. (Assuming we label the states numerically starting at 0)

Likewise,

```
*256 +_
```

represents an infon with 256 states but the state is not known here. Note that '_' does not mean 'void' or 'null' the way a database or programming language might use those terms. It means that there *is* a value, it just isn't given here. We can say it is unknown.

This notation using * and + will be useful shortly.

Infons are immutable. So for example, one's age cannot be a simple infon because age can change as time progresses. But one's age *on a specific date* is a simple infon.

Identity and inference

If we have two infons, let's call them A and B, they are possibly the same piece of information. In programming, A and B could be from two pointers of the same type pointing to the same address in memory. Or it could be that one is a copy of the other. Or perhaps they were both copied from some third infon. However it happened, we say that A and B are identical and we can write that like A = B. To be pedantic, let's have it mean that A and B are the same size

and not scrambled in some way. It could be that only parts of A are identical to parts of B or that A is a sub-infon of B. In those cases A != B.

It is important to note that for infons, equality does not imply identity. For example, if I am asked for my shoe size and I give a number, and I'm asked how many miles I biked today, and I give a number, it is possible that the two numbers I gave were both 9. But though they are equal they are not the same piece of information.

For the purpose of making inferences, if two infons are identical, you can substitute one for the other and there won't be a logical difference. Substitution of identicals is the primary way of making inferences here.

Incidentally, one could use infons to define *equals*, if, for example, there is a need to be compatible with regular math with numbers.

Methodology: The "stays the same size" rule

In academic papers we do not usually mention how and why we used a certain technique. But this is not really academic and I believe that knowing some context about why it is done a certain way will help with understanding. When I was in college I was trying to create a computer notation that could be used

to store knowledge in a way that software could use it. I wanted it to be complete. After I realized that what I was modeling was information I started making progress. Early attempts were similar to the JSON + expressions + time system that I discussed in the previous article. But nothing worked for everything. Some attempts could represent objects but not abstract entities or the class system couldn't handle time changes or it could not express mathematical concepts. After dozens of attempts some quick calculations showed that there were millions of possibilities and that trying them one at a time would not work. After several months I came up with a methodology that I hoped would direct me to the correct solution. Shockingly, it worked. It produced a theory that solved all the problems I was having. I use the method to this day to solve theory problems. Here is how it works:

First, assume that we have a system with some number of states. I usually imagine a system with 12 states. We suppose that this system is isolated, nothing external to it affects it. Now we can ask a question we have. Perhaps we ask how it can be divided into sub-states or if combining two parts must be associative or commutative. The rule is that nothing can cause there to be more than 12 states, because that violates our assumption that we have 12 states.

Let's look at an example. Consider that we isolate 12 (or some number) states in some system. These are states that are not affected in the instant by external things. Now suppose we do the same with a different system — we isolate 12 states. If these two 12 state systems are different from each other then that means there can be two different kinds of isolated 12 state system. Thus we would be able to talk about the *kind* of 12 states that come from the first system vs. the kind that comes from the second one. But if there are two different kinds then we could use what kind to store a new bit of information: 0 if type A and 1 if type B. So we would be able to store 24 states in a 12 state system. So we conclude by "stays the same size" that for completely isolated systems, all 12 state systems work the same way. Obviously we can generalize that for any whole number n, all isolated n state systems work the same way.

You may be surprised that the "stays the same size" method works. But it is actually very restrictive. As far as I can tell (and it's just a conjecture), there is only 1 way that a system following the rule can work.

Identity as a foundational relation

If you think about programming, we get Turing completeness with just conditionals (if's), repetitions (loops), assignment (A = B), sequences of actions

and functions. But the only one of these that really does something is assignment. The others all establish *patterns* of assignments but the assignments are what really does something. A cute way of thinking about this is that the assembly instruction MOV (which does assignment) can be Turing complete. See MOV is Turing complete.

So we can think of asserting that two infons are identical as a declarative way of representing assignment.

We have already given that two infons have a certain relationship to each other. Namely, that they are either identical or not identical. As we shall see, Identities among an infon's sub-parts can be used to fully describe what state it is in. If there is another relation that could be used to describe infons, it cannot carry more information than that stored in the Identities; otherwise we could use that relation to store *more* information in the system than it holds. For example, soon we will see that Identities among an infon's sub-parts can tell you what state it is in. If we could also say that two of the sub-parts have some other relationship to each other — a relationship that doesn't reduce to Identities — then we could use that to store more information. So for isolated

infons, we discuss only relations that are Identity or map to patterns of identity.

SUMMARY of chapter 1

Here are the main point of this section:

1. Infons are immutable pieces of information. They have a size that can be measured as the number of states. And they have a value which is what state they are in.
2. A notation for infons is such that *256+123 means an infon with 256 states that is in a state labeled 123.
3. Infons can have sub-infons
4. The properties of an infon, for example what state it is in, can be fully spelled out in terms of Identity relations or patterns of identity relations among sub-parts.
5. To validly process information, for example, to make inferences, substitution of identicals or patterns of substitutions is the foundational operation.

Chapter 2: Sub-infons

In this chapter we go over some standard theory of how infons can divide into sub-infons. We also add some notation that will be useful and lastly we discuss a problem that motivates the next chapter.

Sub-infons

Obviously, infons can have sub-infons. But let us walk through it because we are going to push on the concepts a little so it will be good to have it fresh in your head.

An example of dividing an infon into sub-infons is dividing a 32 bit word of computer memory into 4 bytes. Or dividing a byte into 8 bits.

The size of an infon can be measured in states. So the size must be an integer greater than 0. So there are not many combinations of how infons can be divided. The rule is:

- Multiplying the sizes of the sub-parts must result in the size of the whole.

This is a review so I'm just going to give an example instead of a wall of text.

How would we get the sub-infons of a 12 state system?

First we get the factors of 12: 1, 2, 3, 4, 6, 12.

Then for each factor, we can pair it with an infon that will make the total size multiply to 12. Here are the possible combinations for dividing a 12 state infon:

- 1 state x 12 states
- 2 states x 6 states
- 3 states x 4 states
- 4 states x 3 states
- 6 states x 2 states
- 12 states x 1 state

NOTE: Obviously 6 and 4 state infons can be further divided but we aren't concerned with that here.

We need to set the values of the sub-parts correctly so that they line up with the parent infon. Continuing with examples instead of a wall of math text, what would the values be when we divide it into 3 x 4? What about 4 x 3?

Values for 3 x 4

Think of this as an odometer with a 3-state wheel and a 4 state wheel. Or like a number where each digit is in a different base: base 3 for one, base 4 for the other.

So to divide *__12__ + __5__ into 3 and 4 state infons, the 3 state needs to be *__3__+__1__ and the 4 state infon needs to be *__4__+__1.__

If we instead divided *__12__ + __5__ into 4 x 3, we would have *__4__+__1__ and *__3__+__2.__

This illustrates that what order the sub-parts are in matters. That's cool because it means we have a list-like structure. Or, as will be described in future articles, in a temporal context this list can describe a sequence of events. Let us add this to our notation.

List Notation: specifying sub-infons

We looked at how *12+5 could be divided into *4+1 and *3+2.

Let us write that like this

```
{ *4+1 *3+2 }
```

We can easily calculate what the combined infon would be. Assume we start with 0. Then do the arithmetic operations from right to left:

```
To get the value of the whole:

0 * 4 = 0
0 + 1 = 1
1 * 3 = 3
3 + 2 = 5

So 5 is the value. The size is  4 * 3 = 12
So { *4+1 *3+2 } == *12 +5
```

Concatenating infons

If we want to combine the parts into a whole we use ( ) instead of { }

So (`*4+1 *3+2) evaluates to *12+5`

We can put as many components in the brackets or parentheses as we need.

If we only concatenate one item it is just itself. That means we can also use ( ) in the traditional way to control the order of evaluation.

SUMMARY of chapter 2

The main points of this chapter are:

1. Infons can be divided into sub-infons as long as the product of the sub-infon sizes is the same as the size of the parent infon. The values of the sub-infons can be determined with division and modulus.
2. The ordering of the sub-infons matters. Different arrangements imply different values for the sub-infons.
3. We can use that fact to define lists or sequences by listing the sub-parts in { }.
4. We can find the value of the whole infon by applying the * and + operations from right to left across the list.
5. Where { } are used to access the sub-parts of an infon, the inverse of that, concatenation, can be notated by giving the parts to be combined in ( ).
6. With no extra work, that means ( ) can also be used, in the traditional way, to control the order of evaluation.

Chapter 3: The Problem of missing structure

If infons did not have identity then we would be done. But since they do have identity there are valid questions that we cannot yet answer. For example, consider *12+5 again. Let us divide it in two different ways, like this:

```
*12+5 == {*4+1 *3+2}
```

```
 *12+5 == {*6+2 *2+1}
```

Now if we take, for example, the left most items from both lists how do they overlap?

We can know they overlap because since the whole system is in state 5, then, given the *4+1 component what possibilities are there for the *6+2 component? In fact, given any three of the four components on the left side, we would be able to fully reconstruct the missing one. So how do we characterize the overlap?

Some information from *4+1 is identical to some information in *6+2. This is important because the algorithm for processing infons involves substitution of

identicals. If there are some identicals that we do not know then there are inferences we cannot make. In particular some Diophantine inferences cannot be made as easily.

We need a different way of breaking infons into parts if we want to untangle the overlapping infon.

Namely, we need to be able to reference individual states.

A single state can be written:

```
*1+0
```

Single states do not carry information.

```
1 state = 0 bits information
```

A non-programming example of a single state is when Henry Ford stated that customers can have any color car that want as long as it is black. In programming, the NULL type is a single state type. It could also be an enum with only one item.

So could we write *12+5 like this?

```
{*1+0 *1+0 *1+0 *1+0 *1+0 *1+0 *1+0 *1+0 *1+0 *1+0 *1+0 *1+0 }
```

The problem is, how would we know that it is in state 5? All the states will look the same. If you calculate it out, the result is always 0. We know that the items in the list have an ordering. But with all the states being the same value (though not the same identity), how could we tell what order the items are in?

The solution is to compare it to another infon that has the same states though perhaps in a different order. This "reference infon" or "observing infon" is held to be "in order". The way that its individual states map to the non-reference infon defines its value.

Most of the time we do not have to think about reference infons. And doing so may seem complex or odd but it actually makes sense. If an isolated, fundamental infon could store a state absolutely then it would be an object not an infon. Information does not have any absolute interpretation. Instead, for isolated infons, their state is relative to a reference copy.

OK, let's go through how it works.

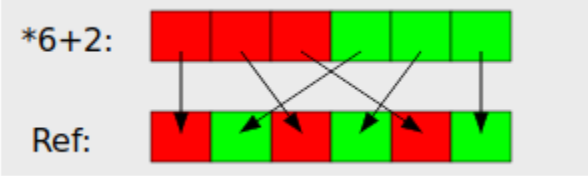
Chapter 4: The state of isolated infons

This chapter is the result of decades of work trying to reconcile information theory, number theory and some mathematics called group theory. At some

point I became unable to easily access academic journals and anyhow it can take me weeks to fully understand an article written in math-speak. So it is quite possible that someone has done this before me. If so, I humbly acknowledge that. For me, the goal of having a better understanding of reality is more important than getting acknowledgement. Also, I have written this in full-on math-speak which more formally reduces all this to the "stays the same size" rule. But the audience for that has very little overlap with who I'd like to share it with, which is curious people who want to make the world better — like myself. I have also written it in code. Feel free to request it.

Infons and reference infons

As mentioned above, in order to describe an infon in terms of individual states we must compare it to a reference infon that we assume is "in order" and has the same (identical) states as the infon we want to describe, though it may not be in the same order. An example will help. Let's go with 6-state infons for now. If we draw the states as squares all in a row then an infon compared to a reference infon will look something like this:

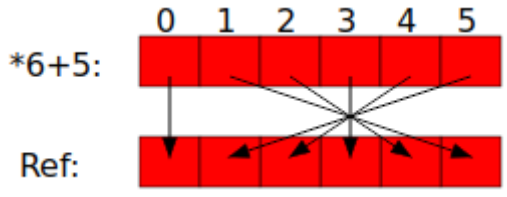


The arrows mean “is identical to”.

The next step is to determine how an infon’s states can be mapped to the reference infon in a way that only allows 6 (or n) states.

Suppose any mapping at all is allowed. Then our 6 state infon could store 6 factorial states or 720, violating “stays the same size”. The fact is that if we want to pass that rule and also maintain the structure described above, where infons can divide into sub-infons, there is only one way to do it. In group theory such mappings are called endomorphisms. The pattern of arrows for making endomorphisms is simple but explaining it in words requires a wall of text. Instead let us look at some examples.

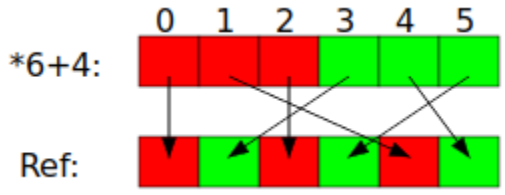
Let us look at all 6 states of a 6 state infon. We will start with *6+5 because it is pretty simple.



Here we have labeled the states from 0 to 5. To get the pattern of arrows, we start iterating the top infon from left to right, that is from 0 to 5. But for the

reference infon we iterate counting by 5 steps. (or n steps for state n). So we start, as always, by setting state 0 of the top one to state 0 of the reference. When we get to the end of the reference states we wrap around. In other words we iterate modulus 6 where 6 is the number of states. So we map 1 of the top to 5 on the reference. Then 2 goes to 4, 3 to 3, 4 to 2 and 5 to 1. Each time we counted 5 steps. We end up getting all 6 states mapped this way.

Now we look at *6+4. Here we will not get all the states mapped so we will have to do an extra step.

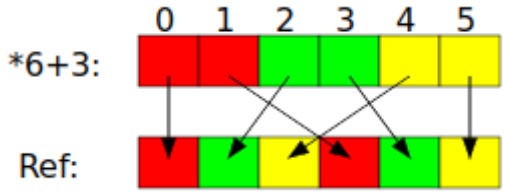


We start, as always, mapping 0 to 0. Then we move forward by 1 on the top infon and 4 on the reference. So state 1 points to state 4 of the reference. Counting 4 more steps, we map state 2 of the top to state 2 of the reference. When we count 4 more steps we find that we are back to where we started at state 0. When this happens, we go forward by one step of the reference. Since we moved forward to avoid a loop I changed the color of the squares to green so that we can see the pattern. Continuing then, we map 3 to 1, 4 to 5 and 5 to

3. Now we have mapped all the states. The 3 red squares are a "coset" and the green ones are another coset.

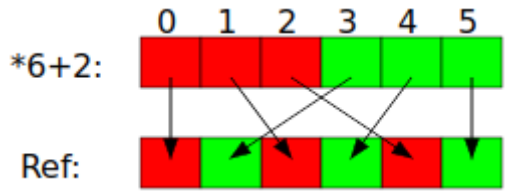
So the new rule is that when we return to a state we have already mapped, if there are still more states to map, then we move forward by one cell on the reference. This makes a new coset. All the cosets will be the same size.

Now we apply this to *6+3:

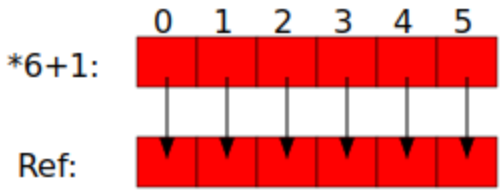


Working though the pattern, we see that there are 3 cosets in *6+3.

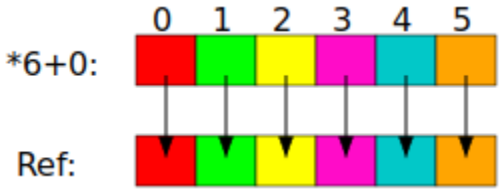
*6+2 is similar to *6+4. You can tell what state an infon is in by looking at how many steps are taken in the reference when going from 0 to 1 on the top.



The last two, *6+1 and *6+0 look similar but there is an important difference.



For *6+1 we have a single coset where each state maps to the corresponding state of the reference. For *6+0 each state is it's own coset. That is because when we "move forward by zero steps" we land on the starting state and thus have to start a new coset and move up 1 cell in the reference.

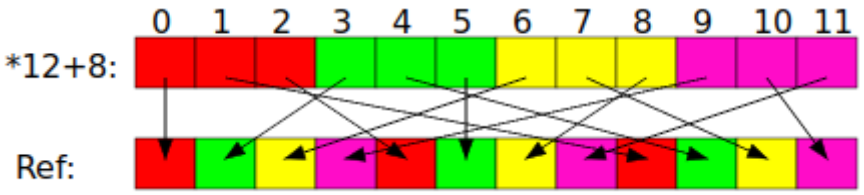


Cosets in an infon all contain the same information — they are identical infons.

Now you can make the identities for any isolated infon. Next we need to find their sub-parts among the states. We will stick to lists with only two items but the process can be repeated to get lists of any size.

For this let us use 12 state infons so that we have more examples to study.

Here is a 12 state infon in state 8:



We want to divide it up. We can divide it into chunks of size 2 and 6, 3 and 4, 4 and 3, 6 and 2. We can also divide it into 1 and 12 or 12 and 1 but we ignore those for now. Here are the non-trivial possible ways we can divide it:

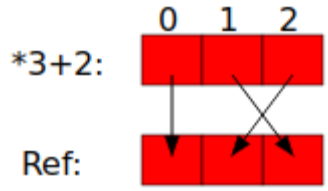
```
1. *12+8 = {*2+1 *6+2}
2. *12+8 = {*3+2 *4+0}
3. *12+8 = {*4+2 *3+2}
4. *12+8 = {*6+4 *2+0}
```

*12+8 = {*3+2 *4+0}

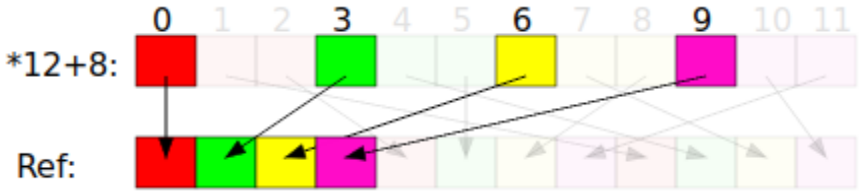
Let us divide it into a 3-infon and a 4-infon as in choice #2.

First we find the 3-infon component. For this we simply take the first 3 states and discard the others.

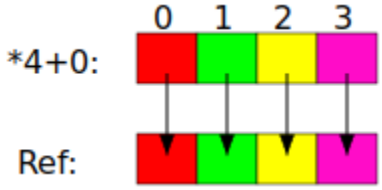
Re-aligning and relabeling the selected states we get *3+2:



Now we find the 4-infon component by selecting every 3rd item. Every third because we want 4 states and 12 / 4 = 3.



Re-aligning and relabeling the selected states we get *4+0:

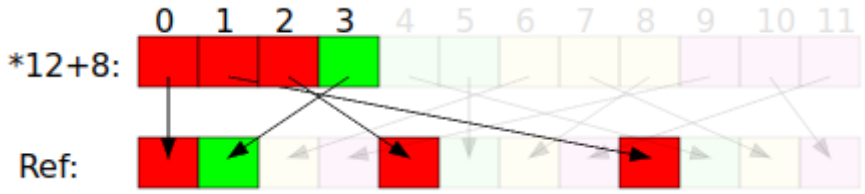


There we go: *12+8 = {*3+2 *4+0}

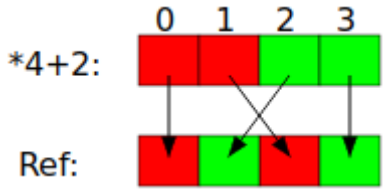
*12+8 = {*4+2 *3+2}

Let us try it with the 4-infon and the 3-infon swapped. This will illustrate the full rules for doing this because of an anomaly.

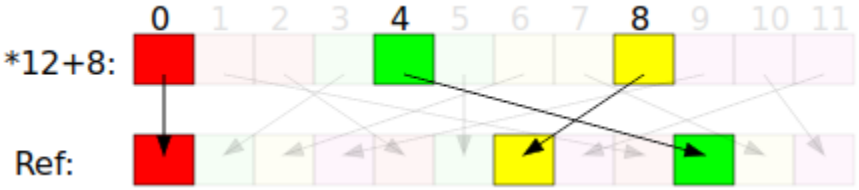
First we get the 4-infon by selecting the first 4 states:



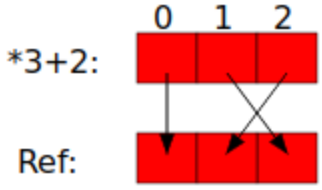
Notice that the last item is from a new coset in a way that breaks the pattern. This cannot happen with the quotients but it can with the divisor. All we do is move the last state to the end. So we have *4+2:



Lastly we get the 3-infon by selecting every 4th state (12/3):



Which reorganizes to *3+2:



Generality

This method of decomposing infons works for any isolated infon. It doesn't matter if the size or value of any part or of the whole is prime or not.

The identities are the important part

By looking at the decompositions of a 12 state infon that we just went over, notice that there were two different components that were *3+2. But they are not identical because they are composed of different states.

The primary algorithm for doing anything with infons is substitution of identicals. So it is important to know all the identicals implied in an infon. In this chapter we went over the main identity assertions that define an infon's state. There are others as well. Namely, the various cosets hold the same information.

The next articles

In the articles to come, we develop this into a complete system for representing reality. We will be able to represent extremely complex systems and concepts including natural languages. Remember, if you did not completely understand this article you will still be able to understand what is to come.

Part 3

The Basic Syntax for Information Structure

Welcome to part 3 of this series on modeling the World in terms of information structure. The theory in part 2 gives a basis for representing information structure but there isn't a way to define structures and combine them into larger structures. Nor are there any practical modifications. For example, in the real world we need the concept of a string. In this part we develop a notation, "Proteus", for representing any possible infon system. As a bonus, our notation will be computer readable and even streamable so that computers can interactively apply the models in useful ways.

An important fact to consider when reading the following is that infons are immutable. They do not change. The "next state" of a system is a new piece of information. Also, comments are like // to end of line OR /* … */

Chapter 1: Low level infons

There are three different types of low level infon. The idea is that these three are enough to allow the development of any other type. They are numeric

infons, lists and a special list where every item has 256 states, that is, a string of bytes. If you program this is obviously familiar.

Before we look at these, there is an important point; it is important to be able to assert that there is a piece of information that we do not know. For numbers we use an underscore "_" to represent "unknown". Note that "_" is not NULL. NULL is sometimes used in programming to mean the value is not available. Here it means that there is a value, but it is not given here for whatever reason.

Bare infons are when a value is given but not a size. Here are examples of bare numeric infons and string infons that illustrate the possibilities:

- 123 // a bare number infon
- 0xffff // a number in hex
- 0B00011101 // a number in binary
- _ // an unknown number
- "A bare string infon"
- 'Another \'string\' with escapes\n'
- $ // an unknown string

NOTES:

- Numbers and strings can be arbitrarily long.
- Strings can contain Unicode but they are measured in bytes.

- There are also here-doc strings and string syntax for streaming large binary strings.
- Later, after we can map states to them, we will add rational forms and decimals.

There is a special syntax for an infon where even the type is not known. It is mostly used internally when processing infons but it can also be used explicitly. It's very simple. An infon of unknown type and unknown value is written

```
?
```

Infons with the size given

Suppose we want to declare an infon of 12 states that is in state 3. We use the * and + symbols like this:

```
*12 +3   // A 12 state infon in state 3
```

Here is a 12 state infon in an unknown state:

```
*12 +_
```

And here is an infon in state 3 but the size is unknown:

```
*_ +3
```

List Infons

We can make a list by using curly braces like so:

```
{ *4+2 *3+1 }
```

That list represents a compound infon of 12 states. But it is divided into a 4 state component and a three state component. To find the state of the whole, we do the arithmetic from left to right. We start with zero:

```
0 * 4 + 2 * 3 + 1
```

Thus the whole undivided system is:

```
*12 +7
```

Lists can be arbitrarily long. And the list items can be any infon. Note that the state arithmetic only works with numeric infons.

```
{ *255+22 "abc"  {2 4 3} }
```

Here is an unknown list:

```
{...}
```

And a partially unknown list:

```
{ 1 2 ... 6 7 ... 9 }
```

An unknown list with 200 elements:

```
*200 + {...}
```

Sizes

While the state arithmetic does not work with strings and lists, the size notation is still useful. Thus:

```
*4 + "hello"  = "hell"
```

Note that without the *4, the size of “hello” would default to 5. This applies to any list, not just strings.

Chapter 2: Asserting identity

We can assert that two infons are identical, that is, they contain the same information, with “=”:

```
*12+_ = *12+5
```

That is a trivial example but, just as with the equal sign in regular math, identities are the primary way of making inferences with information models like Proteus models.

Here is a slightly better example:

```
{ 1 2 ... 6 7} = {_ _ 3 4 5 _ _}
```

In this case we can use identity to infer the entire list.

To be precise, identity means that the size of the identical infons is the same, as well as the contents and the ordering of the contents.

Identity also usually implies that the types are the same. If we need to assert that the underlying information is the same but that the interpretation may be different we can use two equal signs "==".

```
*256 +65 == "A"
```

Because ASCII "A" is 65.

Lastly, if we add a : to the end of the equal symbol, it means that the length of the left side may be shorter than that of the right side. This is useful for parsing strings or lists.

```
"hell" =: "hello"
```

Chapter 3: Concatenating Infons

If we want to concatenate the items in a list we can enclose it in parentheses. In the case of numeric infons in a list this has the effect of calculating the size and state number of the whole system. For example:

```
(*4+2 *3+1) = *12+7
```

With strings and lists, the same operation is concatenation.

NOTES:

- If there are different types of item in the ( ), the type of the first item is the type of the result.
- If there is only 1 item in the ( ), it is just itself, but this means that parentheses can also be used in the traditional way to specify the order of operations.

Chapter 4: Sub-lists vs element lists

There is another list-item operation that will be useful. If we precede a list inside another list with "&", it means that the list is not a sub-list but a list of elements on the same level. An example will illustrate this.

Notice that the list

```
{  1 2 {5 4 3 } }
```

has 3 items in it. The list at the end counts as a single item. But if we add a & before it, the 5, 4 and 3 become members of the parent list. Thus the following list has 7 items:

```
{ 1 2 &{ 5 4 3 } } = { 1 2 5 4 3 }
```

This will be very useful later when we want to describe things that happen for a while then stop and something else happens.

Chapter 5: ListSpecs: Declarative Repetition

What if we want to say something about all the items in a list. For example, what if we want to say that all the elements in a list are 3 character strings. That is, each member of the list is like `*3+$`. (Recall that $ is an unknown string.)

We can put the template at the front of the list followed by a |. Here declares a list of 3 character strings:

```
{ *3+$ | ... }
```

We can actually do a little parsing with this!

```
{ *3+$ | ... } == "cathatbatdog"
```

This would result in the list:

```
{ "cat" "hat" "bat" "dog" }
```

Using what we have learned so far, how would we turn it back into the original string?

If we enclosed it in ( ) to concatenate it we would get the same list back because the list is a single item. However if we mark it with &, we get the desired effect. See if you can work through this:

```
( &{ *3+$ | ... } == "cathatbatdog" )
```

Chapter 6: Partial unknown numbers

A fully unknown numeric infon is represented by '_'. But often we know *something* about an infon. Using what we have so far we can express partial knowledge by using '_' in an expression. Here are some examples:

A 10 state system in a state from 0 to 4:

```
*10 +(*5+_)
```

A 10 state system in a state from 2 to 6:

```
*10 +(*5+_+2)
```

A 10 state system in a state from 2 to 9:

```
*10 +(2+_)
```

A system with 5 or more states:

```
*(5+_)+_
```

A system with at least 2 but no more than 6 states:

```
*(5+_+2)+_
```

Much more complex partial knowledge is possible. As an exercise, how would we represent that the value is even?

Syntax for temporal sequences

We will go over this in great detail in future articles. For completeness, note that if a list has a 'T' immediately after the opening brace it is a *temporal list* and some more rules apply. A temporal list might look like this:

```
{T
  state1
  state2
  &{someAction_to_repeat | ...}
  ...
  futureState
  ...
}
```

Chapter 7: Summary and Preview

In this part we went over the basics of representing infons.

- There are three types: numeric, string and list
- They can be written in the normal ways
- The types are merely interpretations
- If not even the type is known, write '?'
- A string is a list like: `{ *256+_ | ... }`
- Identity can be asserted with = or ==. The latter means that the type may change. Adding a : to the = or == mean the size may be smaller.
- A list in ( ) has its elements concatenate into an infon with the type of the first item.
- ( ) can be used to specify the order of evaluation.

- Partial knowledge of a number can be expressed with '_' in an expression.
- In a list, a sub-list marked by '&' becomes a list of elements in the parent.
- A listSpec is an infon template that declares something about all the elements in a list. It is at the beginning of the list and is followed by a bar '|'.
- ListSpecs provide the declarative equivalent of repetitions or loops.

Preview of the next part

The next part will see defining the remaining parts of the Proteus Language. Examples will start to illustrate just how Proteus works.

We will start by defining one more list syntax that can be used in many ways including asserting conditional situations as well as acting as a function.

At that point we will have quite a bit: We'll have a declarative syntax for the equivalent of: variables, assignment, repetitions, and conditionals or "ifs". All we need is abstractions to be complete.

To round it off we will develop a simple but very flexible way of abstracting infons. Later, we will be able to abstract infons that represent the equivalent of

nouns, verbs, adjectives, and so on. Indeed we can abstract over any other such construct if, for example, we find a language that doesn't work that way.

Lastly we will use what has been learned to describe a file structure that could be used both to parse and to construct a file format or streaming protocol.

Part 4

Proteus: Conditionals, Abstraction and Examples

In this article we continue looking at the notation, Proteus, for describing information structures and thus complex reality. We went over syntax for declaratively representing what in imperative programming would be variables, assignment, and repetition. Here we complete the language by adding declarative conditionals and abstraction. Of course we have the added constraint that it must be defined by the identities it implies so that the reasoning algorithm will work.

Chapter 1: Bracket lists

We have so far defined 2 kinds of lists. Regular lists in curly braces, and lists to be concatenated which are given in parentheses. The last kind will let us extract items from other lists. They are written with square brackets like this:

```
[ 1 2 3 4 5 ]
```

The key is that they evaluate to their last item. Thus the above list evaluates to 5. There are two major uses of this. Each will require some new syntax. First, if the last item is an expression that uses the other members of the list,

particularly the first item, we can use this like a function. But it is a cool kind of function because the internals are accessible if we wish. I know that it isn't “pure” to do that, but real-world processes do it all the time. We might want to model them. Think of real world processes as a bunch of information streams that split, join, and intersect. This construct captures the intersections. As we shall see, it is very useful.

Let us look at some syntax for supporting this function-like mode. We need two features.

1. We need to be able to reference other parts of the list so that we can map to them.
2. We need to be able to set the first item in the list and then read the last item. Or, the other way around for inverting calculations.

Referencing list internals

Inside a list there are several references you can use as an infon.

- %, and %self // These reference the parent list; the one in [ ].
- %args // Reference to the first item. It may be a single infon or a list with multiple items.

External references

We also have some references to external infons:

- %world, %W // Reference to a global model of instances important to the situation.
- %user, %U // Reference to a model of relevant aspects of the current user. Can be used to dereference the equivalent of 'I' or 'my' etc.
- %ctx // Reference to a context object useful in conversations.

The use of these will be illustrated in future articles.

Function like usage

There are two different ways of calling functions which give two ways of chaining functions. We can put the argument on the left, like this:

```
23 :> [ _ %args +1 ]
```

Or on the right:

```
[] <: 23
```

They are the same but chaining functions together is easier to see one way depending on the purpose.

We can also invert a function. Inverted, we set the last item and return the first.

```
24 !> [ ]
```

or

```
[ ] <! 24
```

Notice that I haven't spent much time on functions. That is because they are not that useful in Proteus. They are a vestigial feature from when I was thinking about information in more mathematically traditional patterns.

Scanning usage

Preface

In this section we present a tool for declaratively scanning through a string or list. You might be tempted to wonder why create this novel way to search or parse. Remember we are developing a declarative way to represent any information structure. If those structures don’t allow for searching then they aren’t that useful. So what is being illustrated here is not about search, but about demonstrating how to describe information structures defined only by identity that do the things that Turing complete systems can but in a declarative way that lets us use the structures like knowledge. (Though I am not enumerating the identities here — this is already more technical than is usual for substack.) When we completely specify structures in terms of

identities we can traverse the identities like a graph (a network) to infer infons from other infons.

Furthermore, it may seem that some of the syntax can be complex for a simple search. Again, the syntax lets us describe reality. But the observation is not false. And in our reference implementation there are optimizations and syntactic sugar that increase coolness. However, the optimizations and syntactic sugar cannot cover all the possibilities so the full set of tools is available for more complex descriptions.

Assigning to the whole bracket list

We can use bracket lists to scan or search a list or string. You might ask, How? Don't bracket lists just return their last item? Let's look at some examples.

Suppose we have a list like this:

```
{ 1 2 3 4 5 }
```

and we want to get the third item. Let us set an instance of a bracket list identical to the list we want to search.

Let us use this bracket list:

```
[ _ _ _ ]
```

And we assert an identity:

```
[ _ _ _ ] =: { 1 2 3 4 5 }
```

Remember that '=:' means identical but the size may be smaller. This makes sense because the list on the left has size 3 while on the right the size is 5.

So, if the two lists are identical, except for the size, and the bracket list returns its last item, then the bracket list must be:

```
[ 1 2 3 ]
```

so it will return 3. Notice that in this example the items in the list are required to be numbers because _ is a number. If we didn't care about the type, the list could be:

```
[ ? ? ? ]
```

How can we get any value without hard coding 'the third item'? Recall that we can set the size of an infon.

```
*2 + [...] =: { 1 2 3 4 5 }
```

Here we have an index; the '2'. And since counting starts at zero, this returns the 3rd item.

To do more complex searches we need one more bit of syntax. We need to be able to define an infon that is NOT like something.

To do that we just put a bang ('!') before it. Like this:

```
!123
```

That is an infon that is *not* the value 123.

With that, what if we want to search for the first string in a list? We need a bracket list that starts with zero or more non-strings, followed by a string.

```
[ &{  !$ | ...} $ ]
```

Let's break that down. Looking at that, see the '!$'? As we just learned, that means an infon that is not a string. Then notice that it is inside a list as the listSpec. That is, it is before the '|'. So that means every item in this list is not a string. Further, notice that there is a '&' before the list. Recall that means every item in the following list should be treated as a member of the parent list.

So the bracket list starts with zero or more non-strings. Then followed by a string: '$'. And that string is the last item in its list so will be returned. Great! We can use that to return the first string in a list:

```
[ &{ !$ | ...} $ ] =: { 2 3 4 "hi" 6 7 "there"}
```

returns "hi".

We are working toward a full parser. Let's look at a few more examples.

What if we want to fetch a list of *all* the strings in a list? Or all of some other match? We can combine things we have learned so far to do this. Let us fetch all numbers in a range. Here is the bracket list:

```
{[&{!*4+_|  ... } *4+_] | ...}
```

OK, look at the big picture this time. Notice that the whole thing is an unknown list. But it has a listSpec describing what each member of the list is like. What you will notice is that these listSpecs can stream over a list.

Now notice that the listSpec is a bracket list. It will return the last item. And same as before, we start it off with a list of zero or more of what we do not want. In this case we do not want any infons like this:

```
*4+_
```

So we do not want infons in the range 0 to 3, inclusive. (Recall the representation of numeric ranges in Part 3). Then we follow with the last item which IS an infon from 0 to 3.

Ok, one more small bit of syntax. We do not have an easy way to set that internal list identical to a list we want to search. We will use the symbol <~ to apply a list. So all together we have:

```
{[&{!*4+_|  ... } *4+_] | ...} <~ {1 2 0 3 444 555 666 777 2}
```

And we get the result:

```
{1 2 0 3 2}
```

Alternatives as conditionals

Recall the usage of bracket lists to select the nth item. For example, to fetch the 3rd item we can use a bracket list like:

```
*3+[...] = { 1 2 3 4 5 }
```

Here is a question; what if instead of *3, we put *_ . That is, we leave the size unknown. This, then will assert *one* of the items in the list. By asserting alternatives in this way the engine is free to find a difference or differences between the items and construct an "if" based on the differences or the differences in implications.

Here is a simple example. Suppose we are scanning a string to read a programming language and we have this definition:

```
@keyword = *_ + ['if' 'for' 'while' 'switch']
```

Any string this applies to should be one of the 4 strings listed. The engine could construct its own 'if statement' looking at the first letter, the entire word or even the last letter. Or it can just evaluate the information it has at run time.

By adding some information to the index (the _), the engine can keep track of what it has ruled out.

Syntactic sugar

Before going forward let us look at some ways to skip the intense syntax defined above.

Dot, caret, hash

At the end of an infon, usually a word infon, we can add a dot (‘.’) or a caret (‘^’) or a hash mark (‘#’). The dot literally applies the syntax above the searches for the first item. For example, {1 2 ‘hi’ 4 5}.$ is the same as the syntax described above for finding the first string in a list. So we could say something like:

```
My bike.front wheel.tube
```

A ‘#’ works like `*n+[...]={1 2 3 4 5}`

Thus:

```
{1 2 3 4 5}#2
```

Returns the 3rd item.

The ‘^’ access the parent item’s members.

```
Bob^friend

Bike^rider
```

These access an item in the context regarding Bob's relations to things.

There is also syntax for selecting a sub-list or substring.

Chapter 2: Words and Abstraction

Programming languages often have multiple ways to abstract code. Functions abstract a computation. Classes abstract groups of variables and functions. And Templates abstract groups of classes or functions.

Since infons are so flexible, we only need one method of abstraction. That is, we associate a word from a natural language with an infon.

We use the symbol '@' to signify we are defining a word.

In a toy language we could simply use it like this:

```
@wind = { infon-description of wind }
```

In actual use there are many things to account for:

1. Different languages
2. In real languages, words have many different meanings

3. Many words are made of multiple tokens. For example, “Rubix Cube” acts like a single word.
4. Some words are formal, slang, southern, etc.

In imperative programming, each program can have identifiers (programming words) that are unique to that program and are meaningful to the developers. But Proteus models are intended to be interoperable with other systems, people with different languages, and in situations with massive namespace collision. For example in the real world there are many many people named John. And it would be cool if domain names could be replaced by a description and context so that we do not need dns. In such an Internet there will be many many namespace collisions. But with Proteus modeling them it will not be a problem.

In addition, to be future proof, we should be able to define new Proteus-like languages in Proteus and have them work.

So here is the abstraction system that meets those requirements:

The simplest form is:

```
@word or phrase to define = infonic-description
```

This can be used to define synonyms or words in different languages by putting the old word on the right of the =.

Since there can be words with the same spelling, we can add an identifier to a word with an underscore: run_wn3. The wn here means this is the word identifier of the WordNet corpus which contains a numeric identifier for each word. If you look it up in WordNet, run_wn3 has the description:

```
footrace, foot race, run
-- (a race run on foot; "she broke the record for the half-mile run")
```

Lastly, after the word being defined and before the '=', we can have a list of tags enclosed in < >. Depending on the software, these may mark what language the word is in as well as tags for categories such as slang, academic, or southern. In a game context, the tags could identify what faction uses the words.

When a defined word is used, it is as if the infon associated with the word were entered at the point the word is applied. However, engines do not have to dereference the word. Sometimes, for example, in a syllogism, the meaning of the word is not needed.

Built-in features to support natural language

It would be possible to define a natural language in Proteus and then have the engine understand text in a natural language by transforming it into Proteus code for storing and processing. Furthermore, the results of processing Proteus code, e.g., for a query, statement or request, could, by use of the same definitions in reverse, be expressed in a natural language.

That is what we are going to do, but to make it easier and to save processing power we can write a language module for Proteus that automatically recognizes and transforms different word forms in the language as well as a few simple transforms such as changing spelled-out numbers into numeric infons when appropriate.

Let us look at an example. Suppose we define an object that has a state that can change over time:

```
@bike = {bike frame, seat, front wheel, rear wheel, chain, ...}
```

We can then use other forms of the word and the language model will transform them. Using “bikes” will yield:

```
{ bike | ... }
```

And typing the above may be rendered as just “bikes”.

When there is an ambiguity in stemming or pluralizing words, an alternation is produced. The plural of sheep is sheep so without more specification it will get rendered:

```
*_+[sheep  { sheep | ...}]
```

Organizing definitions

When modeling most things we need to be able to organize definitions, refer to them, etc. Translating universal statements like "Bikes have two wheels" may even involve modification of a model.

In addition to numbers, strings and lists, we add definitions as an information type. An unknown definition is represented as '@'.

If we define:

```
@concept = @
```

We can have a list of definitions:

```
concepts:{
    // Related definitions go here
}
```

This can let us organize knowledge better. We could define a *field of knowledge*. For example, psychology or biology. A field of knowledge would contain the institutions that host it, its important people and their publications and so on.

A paper's ideas can be digitized as concept lists or more complex models, and then perhaps simulated.

Objects as unordered lists

While it is possible to use the syntax defined thus far to express an unordered list, it is quite awkward. And so often in the real world we need lists where the order does not matter.

To represent such we simply use commas in our list.

```
@bike = { front wheel, rear wheel, seat, pedal assembly, ... }
```

You can see how such lists can be used to store the hierarchy of states. Of course we should also define *wheel*, *front*, *bike seat*, etc. But notice that as long as definitions eventually end in expressions of simple states then such lists are simply describing the states of a bike. And of course a real description of a bike would include identities among the states and a description of how the parts change over time. Again, future articles will detail this.

Before we go over the more complex examples, let us look at a simpler example that nevertheless pulls everything together.

Chapter 3: A simple parser

Let us illustrate all this by making a parser for a tiny part of a programming language. This is only to illustrate the tools presented. So, for example, the only allowed names for variables are a, b and c. And the only numbers are 0, 1 and 2. Being complete here would make this section quite large without adding anything new.

Our tiny language, tlang, will consist of a list of statements and each statement is either an *if statement*, a *loop* or an *assignment statement*.

tLang

First we define "**tLang**" as a list of statements:

```
@tLang      = {statement|...}
```

This defines instances of tLang to be an unknown list where every item in the list is a statement.

Statement

A **statement** is either an *if*, *while* or *assignment* statement followed by a semicolon:

```
@statement  = {*_+[loop conditional assignment] ";"}
```

Here we illustrate how a bracket list with an unknown size means ‘oneOf’.

That option is followed by “;” to require that statements end with a semicolon.

Now we need to define loop, conditional and assignment:

Assignment

```
@assignment = {identifier "=" number}
```

An **assignment** is an identifier followed by “=” followed by a number.

Conditional

```
@conditional= {"if" "(" condition ")" "{" statements "}"}
```

This says that syntax for a **conditional** statement is “if” followed by “(” followed by a **condition** then “)” and “{“.

Then comes a list of statements. Notice that we use the plural of statement.

We could also have directly made a list of statements. Lastly we close with a “}”.

Loop

```
@loop       = {"while" "(" condition ")" "{" statements "}"}
```

This is almost exactly like the conditional syntax. Just “if” is changed to “while”.

Notice that if's and loop statements both have a "condition". That is next to define.

Condition

```
@condition  = {identifier compareSymbol identifier}
```

That definition refers to compareSymbol.

CompareSymbol

```
@compareSymbol = *_+["<" "=" ">"]
```

This is like: OneOf "<", "=", ">"

Lastly we need definitions for **identifiers** and **numbers**

Identifier

```
@identifier = *_+["a" "b" "c"]
```

Number

```
@number     = *_+["0" "1" "2" "3"]
```

Trying it out

Here is the finished code. We can put it into a file called tLang.pr.

```
{
    @identifier = *_+["a" "b" "c"]
    @compareSymbol = *_+["<" "==" ">"]
    @number     = *_+["0" "1" "2" "3"]
    @condition  = {identifier compareSymbol identifier}
    @loop       = {"while" "(" condition ")" "{" statements "}"}
    @conditional= {"if" "(" condition ")" "{" statements "}"}
```

```
    @assignment = {identifier "=" number}
    @statement  = {*_+[loop conditional assignment] ";"}
    @tLang      = {statement|...}
}
```

If we load that file into the Proteus engine and then query

```
tLang == 'if(a>b){a=2;};'
```

The "normalized" result we get is:

```
tLang:{
   statement:{
       conditional:{
          'if'
          '('
          condition:{
             'a'
             '>'
             'b'
          }
          ')'
          '{'
          statements:{
                {
                   assignment:{
                      'a'
                      '='
                      '2'
                   }
                   ';'
                }
             }
             '}'
        }
        ';'
    }
 }
```

As you can see it correctly filled in the blanks and the result can now be queried using dot notation or bracket lists.

Summary

In this part we developed the bracket list and looked at examples of using it to search, select and as the construct “oneOf”.

Next we looked at how to define words or phrases and use them to assert an instance of an infon.

We also added unordered lists — lists with commas to separate items — to the syntax. We looked at how this can let us more easily define objects with a state hierarchy.

Coming up

We are essentially done learning the basic language and syntax of Proteus. And we are on the way toward modeling more and more complex systems using Proteus.

In the next part we will develop some techniques that will be necessary for more complex models. For example, currently we only have tools for modeling

systems that have a given number of states. How do we model something that, for all practical purposes, is continuous?

Part 5

Modeling Continuous Phenomena

Now that we have a notation for representing the structure of the world it would be great to get right into modeling complex systems. However this would be like trying to write a useful app in a new language which does not yet have a standard library. We need to first develop some low-level models.

Continuous models

The first few models we will develop have to do with modeling phenomena that are continuous-like. In reality, a true continuum could store an infinite amount of information in a very small space. There is no such thing. As hard as one might try to imagine a true continuum, the planck scale will break it into individual states. For example, suppose we want to represent the position of an electron. We might allocate a 128 bit integer. Or a 128 bit floating point value. But never an infinity bit value. What we need is a way to map states to a system where we do not know how dense the states are, though we do know there are a finite number of states.

Unknown number of states

We have briefly considered the situation where we are making a model and we do not know how many states our system has. An example might be distance. If we want an infon that represents how far an object has moved, in, say meters. Let's say we reserve 1000 states so it can go up to 1000 meters. Obviously we need more than 1000. But, whatever number we choose to be the maximum, it could go further than that. Basically, we do not know the maximum.

This is easy to solve. Just leave the size of the infon as unknown:

```
*_ +_
```

This infon has an unknown size and unknown value.

Unknown step size

There is another problem we need to solve. Suppose we are modeling the rotation state of a bicycle wheel. We might try to model it as an unknown sized number like this:

```
*_+_
```

But this doesn't capture that the wheel eventually cycles back to its starting position. Whereas the infon *_+_ can keep growing and it never returns to its starting position.

What about representing that a wheel rotates through 360 degrees. Like this:

```
*360 +_
```

Now we have a rotation- because the state 360 is essentially the same as state 0. But in reality wheels have many more states than 360. Depending on the precision with which you can measure the wheel's position there can be quite a large number of states to model. It isn't infinite, of course. There can be only one infinite infon, namely the whole universe; it doesn't work for there to be a sub-infon with infinite size. So we must stick with “unbounded”. Anyhow, at some point the information content of a system is limited by the planck scale.

Here is how we can a represent a bounded number of states (say 360) where the size between steps is unknown:

```
(  *360 +_  *_+_ )
```

Recall that the parentheses mean that we combine the two infons inside them into a single infon. If we knew the size of the rightmost infon (the minor sub-infon) we could calculate the number of states of the whole system. It’s

that size `*360`. But we do not know it so we must leave the infon in the parenthetical form.

Now we *could* use that parenthetical form to represent cases where we need to model a for-all-practical-purposes continuous value. But the issue is that since we do not know the maximum value of the right hand side we do not know how far along a value is. For example, suppose the following represents the position of a wheel:

```
( *360 +45 *_+100 )
```

We know that the wheel is at an angle slightly greater than 45 degrees plus 100 somethings. But how much is 100? Is 100 a lot? We need a way to assert, for example, that the right side is 1/3rd the way through its values. Then, whether there are a trillion states or merely 30, we know that the wheel is at position 45 and 1/3 degrees.

We can get what we need by applying the inverse of the " * " operation. First let us assume that we know there are exactly 1000 states in between each degree of angle. In other words:

```
( *360+_   *1000+_ )
```

Evaluated, this would become

```
*360000 +_
```

And if we want to get the value in degrees we can say: `*360000 +_ / 1000`

But we need to look at " / " in more detail. Let us look at a simpler example:

```
*12 +7
```

Recall that one way to divide this is into a 4 state system and a 3 state system like this:

```
(*4+2  *3+1)
```

Therefore, if we divide that 12 state system by 3 states we get *4+2. We can generalize to any value regardless of how many states the system has. So, for values, ignoring the sizes, we get:

```
7/3 = 2.
```

We can try that with a 21 state system and get the same answer:

```
*21+7 = (*7+2  *3+1)
```

So `*21+7 / *3+1 = *7+2`.

Looking at the value, we see that `7/3 = 2`

Now in our value where 7 degrees is represented by:

```
( *360+7  *1000+_ )
```

The evaluated form would be

```
*360000+7000
```

7000 / 3 is 2333. Which is a thousand times closer to the value of 7/3 in a continuum.

Given that, we can solve our problem by having a notation that refers to the major value (the 360) and use state division for the minor value.

So if we want to represent that a wheel's angle is 2 and a half degrees we can use 5/2 which is 2 and a half. Alternatively, the engine will map "2.5" to 5/2.

We have mapped positive rational number notation to states in a way the engine can reason over using substitution of identicals.

Negative numbers mapped to states

What if we want to refer to the last state in a system? For example, in

```
*256 +_
```

The last state is 255. Let us also use the notation -1 for the last state. And -2 for the second to last and so on.

In the case of *_+_, the rule that there are at least enough states for whatever we do means that the negative references will never collide with the positive

ones. That is, any positive number we use will not be equal to any negative references. This will lend itself to using negative values in the typical ways.

Defining some words

We have specified that, for example, 123.456 or a fraction made by applying the inverse of * represents a value of the form:

```
( *_+_ *_+_ )
```

Where the part before the . is the state of the major sub-infon and the part after the . Specifies the fractional value of the minor infon.

Let us define a name for this type of infon. These are rational values so we can use define:

```
@rat = (*_+_ *_+_)
```

If we want to define a limit on the number of states of the major sub-infon we can use a notation like:

```
( *360+_ *_+_ )
```

Here we have states from 0 to 359 and some fractional part in between the degrees. This would be great to model the angle of a wheel.

Another type that we will use a lot is a value from 0 to 1. Let us call that a "unit".

```
@unit = (*2+_  *_+_ )
```

So now, if we want to represent a value that has a start and an end, and is continuous-like we can asset it:

```
unit:123.456
```

Summary

We have reviewed several methods for representing pseudo-continua by mapping states and representations to a finite but unknown number of states. We also mapped the decimal notation to the same. Using the inverses of ' * ' and ' + ' We defined names for two of the new models:

- *rat* for rational numbers
- *unit* for values from 0 to 1

Next

In the next part we begin to make models that will help us to use a more natural notation for making models. We will define simple words like: a, the,

some, any, and so on. And we will begin to use these to more easily make models.

Part 6
Modeling and Natural Language

Now we are getting to the more fun parts. In this part we will begin to see how information structures tend to correspond with natural language and how, with a little extra code we transform natural language constructs into Proteus models and vice-versa.

As with the previous articles I wish to avoid huge textual explanations as well as overly detailed examples. To that end, in what is ahead I will develop various models and then talk about how they illustrate a general modeling technique as well as point out their deficiencies and how they can be improved to cover more complex use-cases.

Chapter 1: A default library

Just as with languages like C++ with its standard library of useful classes and functions, there are a myriad of small models that are used so often we should just assume they are available. Let us build that as we go along, starting with models from previous parts:

```
concepts: {
   @rat  = (*_+_ *_+_)
   @unit = (*1+_ *_+_)
   @coord = rat
   @x = coord
   @y = coord
   @z = coord
   @angle = unit
   @a1 = angle
   @a2 = angle
   @a3 = angle
   @location = {x y z}
   @orientation = {a1, a2, a3}
   @color = *_+[red, blue, pink, ...]
   @object = {location, orientation, color}
}
```

The above is a very preliminary start. Location and orientation do not really capture spaces, the definition of color is almost useless; we will update it shortly. And the definition of objects doesn't account for velocity. Nor does it tell us how to identify a starting position. Nevertheless, this will get us started.

From now on, we assume these models can be used without further mention.

Chapter 1: A Toy Model

As we progress, we will go into more and more detail about modeling complex systems. For now let us start with a toy model that we can use as an example in this article. Let us model a bike. We will not model how the parts fit together

or their shape or what they are made of or how the bike changes over time or how it changes your spatial location if you ride it.

But we will model some of its state at an instant. Obviously we can assert the existence of states with the notation *n+m, or by using rat for rationals or unit or other such definitions. But we can also assert states by talking about a sub-part that has states. So for more complex systems we can simply list their parts to assert their states:

```
@Bike = {
   &object,
   Bike frame
   Steering system,
   Pedal assembly,
   Front wheel,
   Rear wheel,
   Bike chain,
   Bike seat,
   ...
}
```

It might seem that this definition is circular since it contains “bike” which is the object being defined. But recall that these definitions are descriptive. That is, Proteus can search for something based on its structure regardless of what it is named. We wouldn't have to define “front wheel”. We could define “front bike wheel” or just use a generic definition of “front” and “wheel” and refine our wheel description for a bike later. The reason “wheel” will work is that the

description “the bike's around-the-axel thing with gears” would be enough of a match to the description of the structure of a bike's rear wheel. (That is assuming definitions of gears, around, axel etc exist.) So the important part is not choosing the perfect words or phrases to define but getting the model that the words call up to a place that it works for what you need.

General models

In the long run, a really great model of a bike would have a core that was just the essentials of what a bike is. Maybe: 2 inline wheels, a frame, and a way to propel it such that a person can use it to more rapidly change their location. The information flow is from the pedal system and the steering system to the rider’s location. Then other refinements could describe 10-speeds, mountain bikes, different brands of bike and so on. These refinements could include how pedals update the chain and in turn, the rear wheel against the ground to ultimately update the rider’s location state. By analyzing these more specific models the engine could determine if the mechanism would indeed update the rider's location. So there are different ways that being a bike can “supervene” on the states of the sub-parts. But they will all meet the basic core information flow. That is, whether the information flow is from pedals to the rear wheel then from the ground, through the frame and seat, into the rider's location, or,

through some hand-based propulsion mechanism or maybe an electric motor, doesn't matter. If the mechanism's information system matches the core "bike" information flow pattern then it can be found as matching the word "bike".

To complete the definition of a bike we would need to also model all the parts such as the pedals, steering system and so on. Further, we would model how the parts fit together when assembled, how the states of the parts interact and how it can be part of a super-system consisting of a bike and a rider.

Don't worry, we will go over how to do all of these things but with a simpler system than a bike. Spoiler, we will eventually model a rubix cube.

But first, let's look at different ways we can talk about what state something is in.

Chapter 2: Constraints on states

Now we have a way of describing an object by telling what states it has. For example, given the above we could use the following Proteus code. For simplicity, let's assume a bike is in the context %ctx:

```
%ctx.bike.rear wheel.orientation.a1 = 0.5
```

Presumably, this means that the wheel is half way around its rotation. But we do not talk like that. For one thing, what is the starting position? And it's linguistically odd to be so precise.

It would be cool to have a variety of models that, rather than assert that states exist, merely constrain the values of the states in useful ways. For example, rather than having to give an expression of someone's x,y,z location relative to our own, we could encapsulate that in a model and call it “far away” or “really far away” or “3 feet away” or “right next to”.

Let us come up with some ways of describing and constraining the “color” state given in the object model. We need to completely redo the color model given above.

Modeling color

When making a model it is important to think about the purpose it will be used for. That can inform which details to include and when you have enough detail. We want to define names for colors so we can use the Hue/Saturation/Brightness (HSB) model. Later we will also want to map colors to computer representations such as jpegs or streams from a camera. So we also need the RGB model and a mapping between HSV and RGB.

First we note that Hue is in the range 0 to 360 and the value wraps around. Saturation and brightness can be measured from 0 to 1. The states of a color are hue, saturation and brightness. Lastly, we define some color names by constraining the values for hue and sometimes saturation. We can use these colors in a definition of a rubix cube.

```
concepts:{
    @hue = (*360+_ *_+_),    // Hue: rational representation (0-359°)
    @saturation = unit,      // Saturation: unit value (0 to 1)
    @brightness = unit,      // Brightness: unit value (0 to 1)

    @HSB = { hue, saturation, brightness },
    @color = HSB,            // Color is defined as its HSB state

    // Named color definitions.
    // (Fields not given (saturation and brightness) are 'unknown')
    @red     =  color:{ hue = range:{start:355, end:10} },
    @brown   =  color:{ hue = range:{start:20, end:40},
                        saturation = range:{start:0.0, end:0.5} },
    @orange  =  color:{ hue = range:{start:20, end:40},
                        saturation = range:{start:0.5, end:1.0} },
    @yellow  =  color:{ hue = range:{start:50, end:60} },
    @green   =  color:{ hue = range:{start:80, end:140} },
    @cyan    =  color:{ hue = range:{start:170, end:200} },
    @blue    =  color:{ hue = range:{start:200, end:240} },
    @purple  =  color:{ hue = range:{start:240, end:280} },
    @magenta =  color:{ hue = range:{start:280, end:320} },
    @pink    =  color:{ hue = range:{start:330, end:355} },
|
}
```

This simple model of color can be made much more comprehensive in at least three ways. First, It basically only models typical human vision. Eagles and cats, for example, have different colors they can see. Similarly, space

telescopes have sensors for other ranges of light. A more comprehensive model might account for these.

Second, color can be mapped to frequencies of light and the way photons interact with objects and are reflected into eyes or sensors to be picked up by rods and cones or a camera sensor.

Lastly, There is an enormous amount of knowledge from color science that could be digitized in these models.

Mapping to RGB

Let's quickly look at how we can map RGB colors to HSB. One way to do this would be to define RGB as a color with the additional variables red-level, green-level and blue-level and map them to H S B. However, the RGB color model is just as valid as the HSB model so let us re-work the model of color to have both interpretations.

Something to notice in the following is that some intermediate values are defined to be used in the calculations. By keeping these defined in the same concept set, the scope rules of Proteus will not choose their definition if the topic is not converting color formats (and assuming there are other definitions that work better). Also, note the use of the *_+[...] syntax to choose one item

depending on the range of the hue value. This will be briefly discussed afterwards.

```
concepts:{
  @color = {
    HSB,

    chroma = saturation * brightness,
    X = chroma * (1 - abs((hue / 60 mod 2) - 1)),
    m = brightness - chroma,

    rgb1 = {r1, g1, b1} = *_+[
      { hue = range:{start:0,  end:60},
        result={r1 = chroma, g1 = X, b1 = 0 }
      },
      { hue = range:{start:60,  end:120},
        result= { r1 = X,  g1 = chroma, b1 = 0 }
      },
      { hue = range:{start:120, end:180},
        result= { r1 = 0,  g1 = chroma, b1 = X }
      },
      { hue = range:{start:180, end:240},
        result= { r1 = 0,  g1 = X,  b1 = chroma }
      },
      { hue = range:{start:240, end:300},
        result= { r1 = X,  g1 = 0,  b1 = chroma }
      },
      { hue = range:{start:300, end:360},
        result= { r1 = chroma, g1 = 0, b1 = X }
      }
    ].result,

    RGB = {
      red level   = (rgb1.r1 + m) * 255,
      green level = (rgb1.g1 + m) * 255,
      blue level  = (rgb1.b1 + m) * 255
    }
}
```

You can see here how RGB levels are calculated in terms of the HSB values. Notice that all the operations here are reversible and thus the engine should be able to calculate HSB from RGB.

Also notice that all the new values, such as chroma or even RGB, do not add new information to *colors* because they are calculated from HSB. Thus they are merely views into a color.

In the calculation there are different results based on the range of the hue. You can see that inside the square bracketed list (after '*_+' which gives the list an unknown size) there are 6 places where an assertion about the range of hue is given, followed by setting the result. After the [...] we see .result which selects the result.

This is how we encode conditionals. Recall that a list in square brackets returns the last item in the list. But if the size is unknown, that could be any one of the items. So if we put assertions in the items then it can evaluate which one it is by seeing if the assertion is true. If we also set some variable like *result* in each one, then it can choose the correct value for *result* to return. This way of doing conditionals might seem odd at first. But it lets the engine choose how it will determine a truth. Often a programmer will have to

construct an if-condition based on some logic but there could be other ways of making the condition that would be available in different situations. This method just describes possible situations. Determining the test for that situation can be done at run time with the available information. In fact, a Proteus engine could reverse the entire thing by seeing what the result must have been and thus know the range for hue.

Using adjective models

We have made a vague distinction between models that assert that there is a state — like bike or color — and those that constrain states, like red or large or far away. The models that constrain states are useful for describing the values of the states of objects. That is, what state is it in? The reason the distinction is vague is that many models that constrain states do, in fact, also have a state. For example, red, by the definition above, is a color and thus has states. It's just that the hue state is constrained to certain values. Such models can be used like objects. As in "What is your favorite color?" Nevertheless, the primary use of some models is to put constraints on an object's states and thus they perform like adjectives.

Here are some ways we can use adjective-like models:

“a red bike”

```
Bike.color=red
```

Recall that the “magic” infon %ctx stores the last few things referred to in a conversation. So, if we suppose that in a conversation there was a recent reference to a collection of bikes, we could refer to the red one with:

“The red bike”

```
%ctx.(bike.color=red)
```

Or: “The bike is red”

```
(%ctx.bike).color=red
```

The parentheses in these examples ensure we are referring to the *red bike* or the *bike's color* respectively.

We can use the techniques described in previous parts to make references or usages equivalent to:

- All the red bikes
- The bikes that are not red
- What color is the bike?
- Every third bike is red

And so on.

Summary

We have looked at how to model the state of an object at an instant and how to use models that constrain states to common use cases without having to give numeric expressions.

As an exercise, pick some adjectives from a dictionary and think about how to define them in terms of constraints on state.

Chapter 3: Function Words

Languages, in general, can have words in them that are sort of built in. C++ and similar languages have words like ‘for’ and ‘if’ and ‘class’ that are not user defined and are not functions or classes. I think it is cool that in *functional* languages these keywords *can* be user defined. Proteus is like that. In English, there are something like 200-300 function words depending on what you count and if you count different word forms. Words like ‘a’, ‘the’, ‘some’, ‘many’, and so on are considered function words. Let us define a few as examples:

We’ll start with the word ‘a’. We can define ‘an’ the same way.

The word ‘a’ has a number of uses but the primary one, which is its use as an indefinite article, is used to introduce something into the conversion context (as well as to assert or query something in connection with the associated noun). If one says:

```
I got a new bike yesterday
```

They add a bike into the conversation context.

In Proteus, to do that we just use the word naming the model. So where in Proteus we say ‘bike’, in English we say “a bike”.

In Proteus we can refer to a conversation context, with:

```
%ctx
```

The %ctx is an automatically maintained list of references in the current conversation as well as items the conversants may uniquely understand to be referenceable without any context. For example, the moon or the house they both live in.

So ‘a’ can be modeled:

```
@a %arg = %arg
```

Once the bike is in the context we can refer to it:

```
The rear wheel had a bent spoke.
```

In Proteus we can either rely on the engine to parse ‘rear wheel’ as one item and find the instance and model referred to or we can use parentheses. So “the rear wheel” in Proteus is:

```
%ctx.(rear wheel)
```

Assuming that our model of a bike contains a rear wheel and that the model of bike wheels has spokes, the first part of the above sentence will be dereferenced as expected.

So we model ‘the’:

```
@the %arg = %ctx.%arg
```

‘The’ is used to pick exactly one thing from the conversation context. Basically, the last item referred to in the conversation that matches the pattern given right after ‘the’.

This definition of ‘the’ is not exactly right. If there is more than one item to be found it will choose the first one found. For example, what if I instead said ‘the bike's wheel’ there would be an ambiguity because bikes have two wheels. It is

left as an exercise to fix the definition. Make it return all the matching items with the assertion that only one of them is valid. When the engine is conversing with a human, it can be made to resolve ambiguity by asking something like ‘which wheel; there are two?’

Comments

Note how using language this way is very different from a word-only usage. Instead, we do have words, but they call up or compose to models that enumerate the parts and sub-parts of the items models. More than that, the models can say something about how the parts interact to perform a function and even how the whole system interacts with other systems. So we don't need syntax checks to see if ‘the rear wheel’ should be parsed as:

```
The (rear wheel)
```

Or

```
The (rear (wheel))
```

because the latter will not form a coherent model.

Some and Any

Here is Some:

```
@some %arg = (%arg, { %arg | ... })
```

The first %arg ensures that there is at least one of the items. Then there is a list of 0 or more. I would have defined “at least one” using ranges but chatGPT did it this way and I like it.

And Any:

```
@any %arg = { %arg | ... }
```

Some syntax is needed

You may have noticed that these definitions are under specified. For example, the definition for “any” is the same as for making the argument plural. We will need to define some syntax (see the tiny-lang example) to distinguish these usages.

Coming up:

In the next part we will define the states and shape of a rubix cube. We will then look at how the cube can evolve over time. We will describe how to model common changes (such as turning a face of the cube or disassembling it) and to assert that they happened or will happen at different times. I will demonstrate how our reference Proteus / Slipstream engine can draw the cube in 3D and iterate how it looked at different times. We will also look at models

that constrain how changes happen. For example, did they happen “quickly”? Was the transition “smooth”?

Lastly, we will briefly consider how to model the “levels of being” of a rubix cube so that, in addition to being able to determine what would happen if we turn a face of the cube, we could determine what would happen if we held it over a flame or hit it with a hammer.

Part 7
Modeling multi-level change over time

Here we are at the last article in this series. In the previous article we looked at modeling what could vaguely be called nouns and adjectives. In this part we will go into much more detail about modeling complex states and we will also model changes over time and discuss models that constrain such changes. In the process we will also go into more detail about how such models can be composed to describe complex situations.

The example for this paper is a rubix cube. But the topic is not about a rubix cube. Rather, it is about the general principles of how to use Proteus to model anything. In the process of modeling a rubix cube we will look at many different generalizable aspects of a rubix cube including its state at an instant, and events that can change its state over time. We will also delve into modeling its state at various levels. For example, on an abstract level, turning a face of a cube is digital - you turn it clockwise or counterclockwise. But on a more physical level you can actually rotate it smoothly through a near continuum of positions. And while an abstract cube may not have a 'disassembled' state, a physical one does. Each new level adds new states and

transforms. We will briefly look at how we could represent how a material level adds transforms like burning or melting or being smashed or crushed.

Our description of a cube should eventually include that it is a puzzle or toy, what the solved state is and perhaps some tricks and moves that can be used to solve it. Near the end we will briefly look at how a slightly more advanced engine could figure out how to solve it without being given hints by, for example, searching for its commutators.

Chapter 1: Rubix Layer-1

In this chapter we will develop a very basic level-1 state model of a rubix cube. It will be a bit simplistic in order to not distract from the concepts. In the next chapter we will make some improvements to it. Nevertheless, the model of this chapter will be enough for the Slipstream engine to display the cube in 3D.

Some options for modeling

Let's talk about some different ways we could package up the state of a rubix cube. None of them are wrong. And in fact, we could do all of them and connect the various perspectives. The choice depends on what initial use you have for the model. I say “initial use” because as new uses come up, the

original models need not be discarded. So a final model may have many different perspectives to fit almost any purpose.

One perspective is that a rubix cube has 54 color squares that can each be one of 6 colors. Another: a rubix cube has 6 faces, each with 9 such colored squares. These views may be fine for a digital rubix cube that is only used in the abstract or displayed on a screen.

Another view is that a rubix cube consists of 26 physical cubelets with colors on each side. We can be more detailed and say there are 6 center cubes, 8 corner cubes and 12 edge cubes. The relative location and orientation of these cubelets stores the state of the whole rubix cube. In a more physical model we will also want to include the central mechanism that holds it all together.

Let us use the model with center, corner and edge cubes. Remember, we are going to use models defined in previous sections. Especially the object/thing and color models.

As with the rubix cube itself, there are choices in how we describe the individual cubelets. After all, they are not actually cubes but cubes with an appendage to connect them to the central mechanism. The current Slipstream engine has a hard-coded ability to draw a cube and several other shapes.

Purpose drives the choice of how to model a system and since part of the purpose here is to be able to view the results of changes to the cube, let us make them cubes with 6 sides, each with a single color.

```
concepts = {
   @side = {color}
   @edge-cube   = {
      thing,
      *6+{side| ... }
   }
   @corner-cube = {
      thing,
      *6+{side| ... }
   }
   @center-cube = {
      thing,
      *6+{side| ... }
   }
}
```

Looking at these definitions it is clear that this temporary definition of a side as a list of one color will have to be updated eventually. But it serves the purpose for this series. Remember, the engine can draw cubes if it can find a *color* of the sides.

Next we define the three types of cubelet: edge cubelets, center and corner cubelets. Notice they are all three a *thing* with 6 sides. A more complex definition would use a geometric description of cubes. But we are keeping it

really simple here and shortly we will need to specify the colors of the cubelets anyhow so for here it would just add more text.

Here is a tedious way we could set the colors. Let's just look at an edge cube.

```
edge-cube:{
   {red}
   {blue}
   {gray}
   {gray}
   {gray}
   {gray}
}
```

That is a bit verbose. Before using it let's tidy it up by parameterizing the colors of the sides. Let us assert that edge cubes have a front color and a top color. Then we can reference those in describing the cubelet:

```
@edge-cube = {
   thing,
   front-color,
   top-color,
   base-color,
   *6+{ side|
      {%.front-color}
      {%.top-color}
      {%.base-color}
      {%.base-color}
      {%.base-color}
      {%.base-color}
   }
}
```

We could similarly map the base color to the parent color.

Now we can declare an edge-cube like this:

```
edge-cube:{
   front-color:red,
   top-color: blue,
   base-color: gray,
 }
```

After doing the same for center cubes and corner cubes, we can define a rubix cube. I will leave out repeated detail so that this is readable on a phone. Also notice that we make this a T list to denote that it changes over time — at the bottom you can see it ends with "| …" signifying that the previous state description is a ListSpec (like a template) that applies to the cube at every state change.

```
@Rubix-cube = {T
   *8+{edge-cube|
      edge-cube:{
         Front-color:red,
         Top-color: blue,
         Base-color: gray,
       }
      edge-cube: ....
      // repeat 6 more times
    }

   *6+{center-cube|
        center-cube:{ .... }
        center-cube:{ .... }
        center-cube:{ .... }
        center-cube:{ .... }
        center-cube:{ .... }
        center-cube:{ .... }
   }

  *12+{ corner-cube |
       // you get the idea
   }
  | ...
}
```

The "..." at the end signifies that we don't know any of the actual states of the cube over time. So now let us discuss how to describe some important states. Two important states or range of states are the solved state and the assembled states.

The Solved State

The simplest, yet least interesting way to specify the solved state is to enumerate each cubelet's location and orientation relative to the parent rubix cube.

Note that these definitions should be in the same scope as the rubix cube definition so that they don't apply elsewhere.

Also, notice something important here. This rubix cube does *not* have a "T" after the opening curly brace signifying that it is not a T list. Therefore, this represents a single, const state when instantiated. It can be used to describe the state of a cube at an instant. The important part to notice here is that we are setting the location and orientation (position) of each cubelet to where it would be in the solved state.

```
@solved-state = rubix cube:{
   // Set the position of all cubelets
   *12+{edge-cube|
     thing:{
       orientation:{x:3.14,y:0,z:0},
       location:{ax:0, ay:-1.1, az:0}}
     }

   // And so on for each cubelet
}
```

Though this way of specifying the states of the cube is tedious, it is foundational in the sense that ultimately, the state is about the positions of the cubelets. Improvements are merely definitions that let the engine calculate the positions based on assertions about the colors of the faces. And for this example, that is how we want it. By specifying the low-level state in terms of positions we can also define disassembled states as well as smooth transitions from state to state as we rotate a face through the intermediate positions. Of course a better high level definition will map color squares on each side to cubelet positions then assert that all the colors on a side are the same. We will get closer to that later.

Here is how to represent a cube that starts out in the solved-state but may evolve after that:

```
rubix-cube:{T solved-state ...}
```

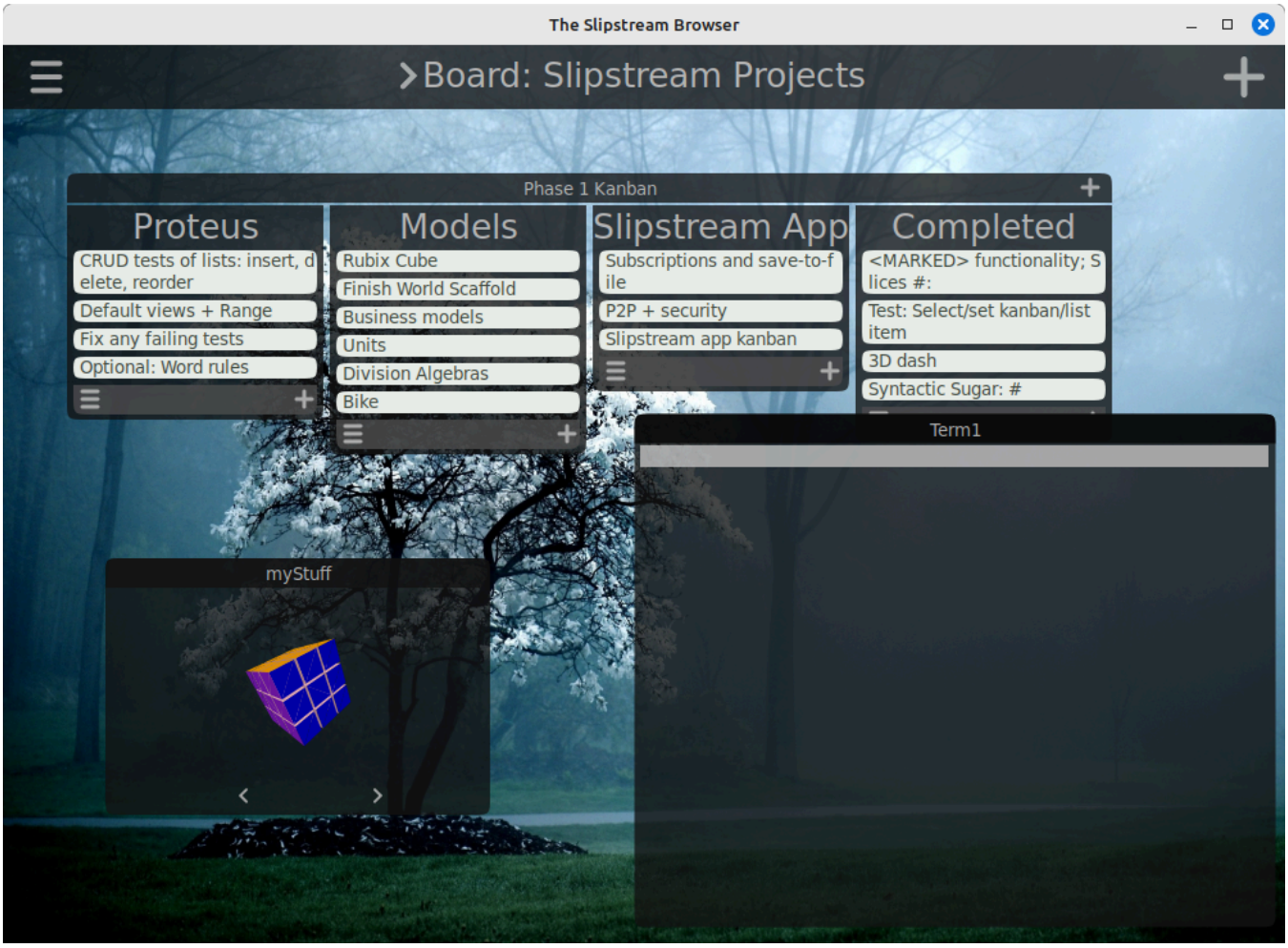
Viewing the cube

Let us use the Slipstream Browser to view a cube like that.

If we:

1. open the Slipstream browser and make sure the definition of a rubix cube is loaded, then

2. add a cube to the “my stuff” infon,
3. and lastly display “my stuff” in the windows, we get the following screenshot. You can also see a Proteus modeled kanban demo and a Proteus CLI.



The < and > controls let you step through time. In this case there is only one state given.

Assembled states

It would be nice to be able to distinguish between when a rubix cube is assembled or disassembled. Unfortunately, this would take some definitions we have not written yet and that go beyond the scope of this series. But we can talk about it.

The cube is *assembled* when all the parts are attached to the whole in the way that makes the cube work. You might ask why not describe how the cubelets attach to the whole and then assert, in Proteus, that "assembled" means all the parts are attached. The problem for now is that the description of *things* or objects that we have defined in our little standard library does not assert that two objects cannot occupy the same space. So, not only can cubelets share the same corner or edge but they will also not hold each other in place. So, for now, we cannot assert that holding together in a stable configuration is part of being assembled.

Another way we might define "assembled" (other than listing all the possibilities) is to say that the cube has "slots" or "connection points" and it is assembled when all the slots are full. This becomes more complex when we try to model the intermediate states when a face has been rotated only partially.

All will be easier when the *thing* model contains more information about how materials work.

Ultimately, we need to define “assembled” more generically. This can be done by describing how a part, of any system, is “supposed” to be attached. When “supposed” is defined and applied we can merely assert that assembled means all the parts are where they are supposed to be.

It may be hard to visualize at this stage, but this illustrates one of the cool ways that the information view lets you model reality. Namely that it does not usually need linguistic categories like *noun* or *adjective*. Once we have defined *assembled* and *disassembled* we will be able to use them as adjectives or verbs: “The cube is disassembled.” Or “The assembled cube is solved.” But also, “I assembled the cube”. In fact, with the requisite definitions, it will be able to infer the meaning of “was the assembly process quick?” without having to model “assembled” for every kind of thing.

Chapter 2: Time in Layer 1

In this chapter we describe the cube in terms of the six cube faces. Then we use those descriptions to describe how turning a face changes the cube's state: our first verb-like descriptions.

If we want to define a list of changes to a cube's state that conform to turning a face clockwise or counterclockwise we need to describe the 6 faces.

Add the following into the rubix cube model; and add definitions into the same scope as the model. Note that these additions do not add totally new states since the added states are mapped to the old states. So we are adding *new ways* to refer to the *old states*.

The following definition of a rubix cube face is similar to the definition of a whole cube. But with a face color, only one center cube, 4 edges and 4 corners. We label the attached cubelets according to the compass points.

```
// Add into same scope as rubix cube:
@face = {T
   face-color,
   center-cube:
       {Top-color=%.face-color},
   *4+{ edge-cube | n, e, w, s},
   *4+{ corner-cube | ne, se, nw, ne}
}

// Add to rubix cube model:
*6+{ face |
   {%.face-color: red},
   {%.face-color: blue},
   {%.face-color: purple},
   {%.face-color: green},
   {%.face-color: yellow},
   {%.face-color: orange}
}
```

Above we have described the parts of a face and asserted that rubix cubes have 6 faces of varying center color.

Next we need to assert that some of the parts on one face are also a part of other faces. For example, the cubelet currently on the n edge of the red face is actually the *same* cubelet as the e edge of the white face. To do this, recall how we can refer to a specific part:

```
// A reference starting from inside the rubix cube definition:

%.face:{face-color:red}.corners.ne


// A reference from outside the definition:

%ctx.rubix-cube.face:{face-color:red}.corners.ne
```

And remember, that as the library grows we will be able to reference things with natural languages like this:

```
The red face's north-east corner cubelet
```

Our next task is to assert all the cases where the 6 faces have cubelets in common. There are 8 “3-way” identities where three faces share a corner, and 12 “2-way” identities for the shared edges.

In this case, there are only 20 identities to assert so it is easy to just list them — which is what we end up doing here. However, the underlying purpose of this part is not to describe how to model a rubix cube but to look at how to model things in general. So in the next few subsections I want to discuss some aspects of asserting collections of identities in general.

Don't use indexing when it is artificial

If you, like me, are from a programming background, it would be natural to think about using ordered lists for the cubelets and faces and then using an index to assert patterns of identity. But in Proteus that is not the best way to do it. That is because the cubelets and faces do not have an inherent ordering. So every time you want to apply the definition to a real cube you would have to specify the ordering of the cubelets and faces for that type of cube. Instead, think about how you would explain the identities in English. We would refer to each cubelet by its colors or type and to each face by its color or relative position. So below we will refer to the faces by their colors and to the cubes at each location in a face by their relative compass point position.

Here we define both a 3- way and a 2-way identity:

```
// ——— Corners (3-way identities)
// (red ↔ white ↔ green)

%.face:{face-color:red}.corners.ne    =
   %.face:{face-color:white}.corners.se  =
   %.face:{face-color:green}.corners.nw
....

// ——— Edges (2-way identities)
// red ↔ white

%.face:{face-color:red}.edges.n  =
   %.face:{face-color:white}.edges.e
....
```

But when the library grows there will be better ways to do this. For example, with a more geometric definition of a cube and with a definition of *thing* that doesn't allow two things to occupy the same space, the identities could merely be inferred rather than explicitly stated. Also, with more words modeled we could construct natural language statements that expand out into the 20 assertions.

Turning a face

Next we want to define turning a face clockwise, leaving counterclockwise as an exercise. As we have seen, a high level description could be made that uses definitions of “turn” and “clockwise”. However, here we will use a low-level description. The low level description will be important even with the higher

level descriptions but with higher level descriptions they would be inferred rather than explicit.

Rule for Identity in T-lists

Now that we are ready to model state changes over time, I need to mention a rule for the notation. *In a T-list, the Right-Hand-Side (RHS) of an identity assertion refers to the previous value of what it refers to.* That makes these identity assertions act like declarative assignment statements.

Here is a description of turning a face clockwise. Notice that this definition has an argument, which is the face to turn.

```
@turn-clockwise face = {
  face.corners.nw = face.corners.sw,
  face.corners.ne = face.corners.nw,
  face.corners.se = face.corners.ne,
  face.corners.sw = face.corners.se,

  face.edges.n = face.edges.w,
  face.edges.e = face.edges.n,
  face.edges.s = face.edges.e,
  face.edges.w = face.edges.s
}
```

Notice that we used an unordered list. This means that these state changes logically happen at the same time. So even though the n cubelet was asserted to become the new w cubelet, the next line of code moves the n cubelet to the

e slot. This is not the “new” n — which was w. That might sound confusing, but essentially, the rotation works as expected.

Now we can apply that definition like this:

```
rubix-cube:{T
  solved-state
  %.faces.red = turn-clockwise
 ...
}
```

Let's review that. We have a rubix cube in T mode, so it is a list of states it goes through. It starts in the start state. For the next state we refer to the faces list and rather than explicitly specifying that we want the face:

```
{face-color: red}
```

we rely on the engine to search for the most obviously red one with just “.red”. If there is a chance for ambiguity we should use a more specific reference.

Next, we need to make sure that references inside the RHS can be found, though it may need to analyze the LHS to do so. In this case, it is the red face that is changing.

Specifying multi-part state changes

Notice how the above state changes just changed one face. But it is possible to rotate multiple faces at once. We can do that be giving an unordered list and using the & operator:

```
&{
  %.faces.red = turn-clockwise,
  %.faces.blue = turn-clockwise
}
```

Since we used an unordered list, the state changes are assumed to be independent. So it works to move opposite faces at the same time. However if we moved two connected faces at the same time there would be some cubelets with two contradictory locations and thus it would be an error.

Repetitions in time

Instead of an *unordered* list we can include an *ordered* list of actions which, of course, means they happen sequentially. This applies to lists with a listSpec as well. Thus, if we put changes in a listSpec it means that they occur for each item in the list. So here is how we can specify that we turn the red face 4 times:

```
& *4+{T faces.red = turn-clockwise |
   ...
}
```

This is a Time list with 4 events in it. All of them are turning the red face clockwise.

Comments

Notice how the rule that, in a T list, the RHS has a slightly more complex way to dereference, namely it refers to the previous state, changes the semantics in a way that matches up to how programming languages use sequences, loops and conditionals.

There are a few new features though. For example we can use "…" to specify sequences when we do not know the states. For example we could specify a rubix cube that starts in the solved-state then possibly undergoes some unknown state changes, then the red face is changed using "…" between the state changes.

However, so far we have only specified the contents of a T list from inside it. Let us look at how we can inject events into a T list by referring to it from the outside.

Chapter 3: Positioning in Time

Suppose we have declared that a person named Alma exists. Assuming all the relevant models are written, that might look like this:

```
Human:{T name: "Alma" | ... }
```

Of course we can go into this definition with a text editor and add all kinds of details about Alma's life. The … covers from her birth to her death. We haven't talked about measuring time or space yet, but by using a metric we can specify that during different years different events happened to her. A sub-T-list might assert that during that time she was in college. Another, perhaps overlapping, sub-T-list might be when she is married to someone. And when she had a child.

But we are editing her infon from a detached position. This is fine if we are, for example, an historian describing the life and times of Aristotle. Because we are detached from that time and person. But what if we want to be more interactive? What if we want to say “Alma is walking to the store”? Or “I'm going to walk to the store when she gets back”?

We need three features. The first feature we already have defined in previous chapters. Namely, the ability to refer to or inject parts of a list from outside that list. The second we technically defined but an example wasn't given: the ability to refer to a sub-list. In a T-list this is used to refer to an interval in time. Lastly, and this has not been described yet since we have barely discussed how a Proteus engine works, we need a "now" cursor in an infon that points to the current time in an infon. Or at least, the engine should be able to find "now".

Walking

Without making a huge definition, think about how we might define walking for humans. More specifically, walking "to" somewhere. Think about *walking to the store*. The definition would be a T list of something that could walk. The ListSpec would likely consist of two simultaneous loops, one would be *take a step* (which could, simplistically, be *step left* followed by *step right*) and the other process would be something like *adjust balance and direction*. Each step would alter the walker's location in space with the final location being within a range of values that matched "at the store".

Choosing a time interval

We have the whole parsing tool kit with which to select a time interval. It could start "now" or it could be in the range before or after *now*. Or we could search for an event. We could search for an event before or after *now*. We could estimate how long the interval is or we could select an interval between two events. We could use named intervals like "the 80's" or "yesterday". Events could be after *now* like "when Bob gets back from the park".

How ever we select an interval of Alma's life infon, we inject into that interval the event *walk to the store*.

In English, and many languages, we often select a time-interval by changing the form of the verb being injected. A Proteus language module could use forms of a word to *combine* selecting an interval with using a time word. As of this writing, our reference implementation of the English-to-Proteus language module does *some* such conversions but it does not yet convert phrases like "walked" or "will have walked" or "is walking" into the appropriate model from the model of "walk". We *could* define it without the language model. For example, an infon could take a verb model and a word like "past tense" and produce the corresponding model that selects the "before now" time interval,

with ‘ walk’ injected into it. However, by hard-coding an English-to-Proteus language model we can speed processing up significantly.

Chapter 4: Layer-2: continuous state changes

If the first layer of a model is the essential description of information structure with no details on how it works, then layer 2 might be the physical layer. But to be clear, the use of the word “layer” is pragmatic not precise. Our layer 1 description of the rubix cube included *some* physical aspects, after all. A “pure” layer 1 model might just consist of 6 faces of 9 colored squares and no reference to cubelets. So while the concept of layers is interesting, getting dogmatic about what entails a “layer n” model will just be annoying. Nevertheless, the concept of layers is important. And each layer provides opportunities for new verbs and nouns. So let us look at how to model that turning a face isn't a discrete action but that there are intermediate states as we rotate it through a quarter turn.

The first step is already done. Namely, we already defined the positions of faces as continuous over 360 degrees via the “thing” model. So what we really need to do is specify that the “normal” states are those where the faces are at multiples of 90 degree rotation. And turning a face means rotating it through

90 degrees. Similarly, even though the cubelets can be in a continuum of positions, there are only 2 normal positions for edge-cubes and 3 for corner-cubes.

Carefully defining each part's position

Defining the continuous updating of the cube's state requires that we be careful about where we define each part's position, that is, its location and orientation. In our first model we defined the parts of each cubelet relative to the whole cube. Now, however, we want to talk about updating their positions based on rotating a face. We can solve this by describing the hierarchy: the rubix cube model is where we will define the positions of its 6 faces and each face will describe the positions of each of its cubelets. This does mean that cubelets will have their positions described relative to multiple faces. However since Proteus is declarative, and since past states are not necessarily valid in the middle of a transition, this will not cause a contradiction. Yes, turning two intersecting faces at the same time would cause a contradiction, but that is how the engine knows it isn't possible without breaking it.

Let's look at the outline of our updated rubix cube description. You can see the full text file at rubixCube.pr

The rubix cube definition asserts that it is a *thing*, then we list the 6 center cubes, the 12 edge cubes and the 8 corner cubes (see the previous examples). Next we describe the 6 faces and the identities telling how the faces intersect. Lastly we have the description of rotating a face in terms of how doing so changes the positions of its cubelets. We also need to define the solved-state in order to ensure that the cubelets in the faces are the same cubelets listed for the cube as a whole.

Here is the parts we haven't seen before. Here we describe the 6 faces (only the red one is shown here). Notice two things. First, the face's location coordinates are left unknown. They will be given in the rubix cube description. And second, their orientation’s z coordinate is asserted to be one of 4 possibilities. Each possibility corresponds to a 90 degree angle. This is telling it that the valid states require the faces to be rotated into one of those positions. Doing it this way allows that there are or can be intermediate states but they are not included in the list of rubix cube T states.

```
// New part of rubix cube description
*6+faces:{
  red-face:{
    thing:{
      location:{x:0,  y:0,  z: 1},
      orientation:{
         x:0,
         y:0,
         z = *_+[0,90,180,270]
      }
    }, face-color:red
  },
  // Repeat for the other 5 faces
 ....
}
```

Now let us look at the new description of a face. Here we give more information about the cubes in each slot. Again, we only include the north edge cube for brevity but you can see the whole thing using the link above.

The important part here is similar to the important part in the new description of the whole cube we just looked at. Namely, we give the specific location of this cubelet relative to the face. But for the orientation we give two options corresponding to both positions the edge cubelet can be in.

```
// Description of a face
@face = {
  thing,
  face-color,
  center-cube:{face-color: %.face-color},

  edge-cubes:{
    n :{ thing:{
      location:{x:0, y: 1, z:0},
      orientation:*_+[{x:0,y:0,z:0}  {x:90, y:0, z:180}]},
      sides:{face-color, _}
    },
    e :{ .... },
    s :{ .... },
    w :{ .... }
  },

  corner-cubes:{
    // Do the same for ne, nw, se, sw
      ....
  }
}
```

With that we have a definition where the faces and cubelets can move continuously but we have also specified the positions that count as valid states of the cube.

There are several different ways we can represent how an actual transition occurred. It might be tempting to represent a continuum of linear state changes for a face where the end state is the new normal one. But actually, we do not always rotate a face linearly. We “could” rotate it a half-way, pause, then go back a bit before finishing.

So we want to represent such evolution by having a T list with a start state (given by the previous state), followed by a list of intermediate states. These states can be pseudo-continuous. Then ending the T list with the desired final state. The intermediate state list can be left empty: "...". Or it can have detail added as needed. By using listSpecs in the intermediate list we can say things like the change is smooth or it accelerates or pauses.

In the examples so far, we have just asserted that, for example, the cube is in the solved state. But, what if we need to assert that it was in that state for 1 second? Assuming we have defined units, such as a second, we could have a sub-T-infon like this:

```
& *(1*second) +{T solved-state | ...}
```

Presumably, *second* represents the number of Planck time units in a second. It's a huge number but that doesn't have to matter since the logic is done by substitution of identicals toward a normal form. We will rarely care about how many Plank units there are.

Before looking at how to model continuous movement, consider this example where the red face is rotated 90 degrees in six, 18 degree increments.

```
// ------ Example: state changes ending in a normal state ------

// A reusable verb that rotates *any* face through a six-step turn:
@rotate-face-in-steps face = {T
    // Six intermediate states: add 18° each step (0→18→36→...→90)
    *6+{T
        face.thing.orientation.z = face.thing.orientation.z + 18
    | ... }
    // Final "snap" into the exact 90° normal state
    face.thing.orientation.z = 90
}

// Apply it to the red face of a cube that starts solved:
rubix-cube:{T
    solved-state
    // this injects the six-step animation and final alignment
    %.faces.red = rotate-face-in-steps
    | ...
}
```

Now to make this pseudo-continuous for 1 second we assert a second's worth of rotation through 90 degrees with the final state being exactly 90 degrees like this:

```
*(second) +{T
   face.thing.orientation.z =
     face.thing.orientation.z +
     (90 / second)

   face.thing.orientation.z = 90
}
```

There are a number of ways to represent such changes more generally. For example, we can refer to a parent's length or a state's position in a sequence.

However, consider that for a real situation we do not know in advance if a turn of a face will proceed linearly. A turn could stop or turn around. Thus the best model for the definition of a face turn simply does not specify the intermediate states:

```
{T
  // There are many states changes
  & *_+{T
    face.thing.orientation.z += _
   | ...
  }

  // The final state is +90 degrees
  face.thing.orientation.z = 90
}
```

Ultimately, much of the boilerplate, such as ensuring that a continuous movement goes through states without skipping any, can be added to the *thing* model and a model for the verb “turn”. In particular, allowed verbs controlling updates to the position should require that if going from position A to position B, all the intermediate states must be used.

The result is that we can query a model at its various levels. At the face level we get a continuum of states. But at the rubix cube level we get its states in full quarter turns. At this level the states could, for example, be mapped to one or more rubix cube notations like "`{M-R4, T-R3} R2 U S'(I2) U2' S(I1) U R2`". How? First use the techniques described much earlier for defining a syntax. Next, map the syntax to a sequence of face rotations. Then you can inject those rotations into a model of a cube. Or, do it backwards: given a Proteus model of a sequence, query for it to be in the notation you want. With both a rubix cube model and a notation model, the engine only needs substitution-of-identicals to parse or write the notation to thus learn the state of a cube or to communicate a particular sequence. Essentially, the new notation has become part of Proteus.

Simplifying the model

As the model library grows there will be a better "thing" model that captures the fact that matter cannot overlap in space with other matter. Also we will have more geometric concepts defined. With those two features, many aspects of the model could be inferred rather than given explicitly as we have done here. For example, the identity statements telling which cubelets in one face are the same as a cubelet in another face can be inferred. After all, if we know

that two faces intersect in space and that objects cannot share the same space, then we can infer that the two face's references to a cubelet in a shared spot point to the same cubelet.

Chapter 5: Toward other levels

We defined a semi-logical level for a rubix cube and added verbs for rotating faces. Because it is only semi-logical — we defined physical cubelets — we also have verbs for assembling and disassembling a rubix cube. With a little more work we could express the geometry and materials of the cubelets and of the central core that holds it all together. We could describe some plastic 3d shapes, vinyl stickers, metal screws and springs, etc. Each of these would have corresponding verbs. So now we could talk about taking the stickers off and reapplying them as a cheat to solve the puzzle.

To go a level deeper we could describe the structure of the materials we use. For example, we could describe various metals as different atomic lattices. This would let us calculate verbs like smashing and melting the rubix cube.

On hearing that, it can seem like declaring a lattice of many trillions of atoms would be unwieldy. But not as much as you might think. First, just because we declare an infon with many trillions of atoms or molecules does not mean the

engine has to process all of them. Remember, we do not need to simulate atoms. We only need to know how information is contained in and flows through the lattice. It may be able to infer the properties of a material from a representation of a lattice. That isn't my field so I don't know. But I do know that if it is possible Proteus can represent it and process it.

Even lower

I have seen hints of a physics theory that represents fundamental particles and their interactions in terms of how information is exchanged between the particles. For example, what information goes into an electron when a nearby charge is accelerated? Other than seeing a reference to this in a video I have not been able to learn more about this technique. But it would be a good way to model particles in Proteus.

Chapter 6: Level 0: The context of a rubix cube

In addition to these descriptions of how an artifact can change and how it works and what it is made of, it can be useful to go up another level to describe how the artifact interacts with other objects. For example, a bike sprocket and a bike chain may have their own descriptions but there can be a “level 0” description of how they work together.

Many level 0 models will be about how a human uses something and why. For example, how does a bike / rider system work to update the rider's location? Why would a person want to ride a bike?

For the rubix cube we might make models of a toy or puzzle that persons can use for entertaining learning.

Here is how we might do that in terms of state:

We can assume part of the state of a person is a store of knowledge. And for humans it isn't controversial to say that we have a state `having fun`. It will be great to someday get models of the lower layers of that state!

We could naively specify that an increase in the knowledge store is "learning". Then we can define a human activity of "having fun and learning". An object that facilitates having fun and learning can be defined as a toy or a puzzle.

Now we can model the context or layer 0 of a rubix cube as being a toy or puzzle.

Intensional states

It may seem that definitions involving people, or more specifically humans, might be too hard to make. Not so. In fact, this is another case where the

information structure paradigm really shines. Humans have states called "intensional states" these are states that have a meaningful piece of information attached to them. Common examples of intensional states are beliefs and desires. The classic example is 1. Bob *desires* to drink a coke. 2. Bob *believes* there is a coke in the nearby fridge. Therefore, Bob gets up, goes to the fridge and gets the coke to drink.

Notice that the action *drinking a coke* or the state of a coke being located in a refrigerator are easy to represent in Proteus.

Assuming we have the requisite models we can thus make infons like:

```
Bob's beliefs: {
   The time is after 5:00,
   Alma will be home soon,
   There is a coke in the fridge
 ...
}

Bob's desires: {
   Someone is in a "loves Bob" state
   To see Alma soon
   To drink a coke
 ...
}

Bob's dreams:{
   Live with Alma in the cute house she likes,
..
}
```

Then we could possibly model the complex interplay that produces actions from beliefs and desires. (Of course a realistic model would need much more detail; for example, the amygdala has states that affect behavior, and so on.)

In another article, not in this series, I can provide more concrete examples of modeling intensional states and show how the engine can work with them.

Chapter 7: Some inferences

Measuring quality

With the ability to work with models at one level and compare how they correspond to the same model at a different level, we can calculate whether one way of making a system is better than another for a specific purpose. For example, we could model a rubix cube made of plastic and see how it works as a puzzle. Then we could model one made of thick wet clay. Or model one where the shapes don't fit well and come apart easily. Each could be analyzed for how it provides learning and entertainment. Or whatever purpose we define.

Solving the rubix cube

The main way that the Proteus engine uses models to deduce knowledge or to answer queries or to plan actions is through a process of substituting a piece

of information for an identical one. If the engine is given instructions for solving the rubix cube, for example, a sequence of conditional substitution steps and loops terminating in turning faces, it could do it. It could also be given different patterns of actions for solving parts of it. For example, a sequence for moving a cubelet from the bottom face into the right position on the top face. With enough such patterns a good Proteus engine could piece together how to solve the cube. But instead of giving such steps to it explicitly, we could model experimenting by doing a few moves, turning a face, then reversing the first sequence of moves and observing what, if anything, changed. As long as the engine can cache the "commutators" it finds, it could then learn to solve the cube, and many other puzzles, from scratch.

Geometric inferences

It has been mentioned several times that, ultimately, we need models for doing geometry. As of this writing they are not written. But there are several ways it could be done. In the end, all the methods should be done; we need not pick one. An obvious method would be to translate the equations and definitions defining algebraic geometry into Proteus identity statements. Alternatively, a fun and historical way to add geometry would be to define a pencil, a compass

and straightedge and use assertions about how they can change the state of a writing surface by drawing lines and circles.

These methods would not only work, but by embedding the concepts in history at the time they were discovered, and by whom, they could be used to record the history of geometry.

But the reason I have not done this yet, is that I am hoping, admittedly against all odds, that there is a more information-centric way to do it. If you read part 2 of this series you know that Proteus is based on a mathematical structure of information called an infon. The structure of infons isn't specified in terms of axioms and theorems but by asking questions about a constant sized infon and determining the answer by querying whether the various possible answers to the question would result in the system under question gaining or losing states.

Infons store state, but because different ways of connecting to them cannot add new information, they end up having, not only a numerically ordered state, but also a sort of "phase". It is my wild speculation, and current area of research, that this phase acts like an imaginary part and that large infons, therefore, are like complex numbers. If so, we should be able to map the

location and orientation of the "thing" model to an almost naturally occurring quaternion, octonion, or higher level infon. Indeed, because parts of such infons cannot be each other — they have an identity — we may automatically get a type of infon lattice that entails that matter pieces cannot occupy the same space as other matter. This would greatly simplify the thing model as well as the geometry models.

Again, that is wild speculation. It almost certainly will not work out.

Chapter 8: Future work

The reader may have noticed while reading through this series that the expressiveness for making models as well as the complexity of what models can be made goes up as more "foundational" models exist.

Here are some models that will increase the usefulness of Proteus:

- Geometric models
- A good Thing model
- Models of the SI units such as meters, seconds, and so on.
- A good model of simple personhood so we can talk about WHY we need different things. And perhaps models of, for example, Maslow's Hierarchy of Needs and Kohlberg's Levels of Moral Development.
- A model of how we use money

- Basic models of our situation: We live on the surface of a planet, there is a sun, there are land masses and oceans, countries with governments; we have a 24 hour day…

Once those basics are modeled we need a system for validating new models that each person can apply to incoming information as they wish. With such a fast way of collecting and validating theories and news, we can work together more easily to rapidly advance different fields like medicine or social living or achieving consensus in politics. And because these models are text files and are human readable, AI based on them will be an order of magnitude safer.