TLH

The LC System:
A Foundation for Abstract Science

TLH

TLH

Introduction to the exposition

In a nutshell, and speaking very informally, the purpose of the LC system contained within this exposition is to provide a common foundation for abstract science, especially what we think of as logic, mathematics and mind or artificial intelligence (AI). We fully recognize that it is unusual to lump mind and AI in with math and logic. However, that is where the study of mind naturally belongs. Mind, as a logical collection of structures and processes, is not bound to any particular physical embodiment. This is the hallmark of an abstract science. Furthermore, such fundamental issues to math and logic, such as well formedness, can not be mechanically resolved without some account of the requirements for symbolic expression processing- which requirements fall under the aegis of language or, more accurately mind. Of course, the proof is ultimately in the pudding, as the reader will need to judge.

The term "foundation" is used to refer to any and all mechanisms in the sense of structures and processes that must exist and be working in order for any abstract scientific expression to even have the possibility of existence. It should be stressed that such a common foundation will not be the result of arbitrarily merging what are more accurately treated as separate bases. Rather, it will be shown that there is only one foundation and any attempts to artificially treat them as separate inevitably lead to unnecessary reification and conceptual redundancy.

In addition to presenting the LC System, the purpose of this exposition is to show how it provides a more consistent and more complete foundation for mathematics, logic and mind than is currently available within any of these disciplines -either in isolation or taken together- by using the system to provide a thorough, constructive critique of the canonical foundations

Before the reader can evaluate the merits of the system presented herein, it is critical that s/he has an accurate understanding of the form and content that anything with the same goals as the LC System would need to have. What does or should a common foundation for abstract science look like? Is it purely an intellectual exercise? Or does it provide some practical benefit? Why is having a better foundation for abstract science so critical for the study of mind and artificial intelligence? Is a common foundation a good or even meaningful idea? From a pile of myriad expositions all claiming to be a common foundation for mathematics, logic and/or mind, how would one quickly discard all but the serious contenders? And having found some number of serious contenders, how would one judge their individual merits? Assuming it's a good idea, what are the objectively testable attributes of an ideal common foundation?

The purpose of this introduction is therefore to provide a framework for thinking about the foundations of abstract science. Towards that end this introduction is comprised of the following three parts:

Part one briefly describes the importance of the foundations of abstract science to the practice of concrete science. It then highlights a few areas where there is clear room for improving our understanding of the foundations of abstract science. Then it describes how such an improvement would provide substantial practical benefits to all concrete sciences, and would provide additional benefits to several specific concrete sciences. It also highlights some of the areas, of interest perhaps only to those concerned with the foundations of mathematics and logic, such as infinity and irrationality, where the LC system provides a more consistent and comprehensive foundation than is currently provided for.

Part two describes in a layered fashion the variety of distinctions that need to be understood by the reader in order for her/him to understand the starting point, form, purpose and methods of testing of any purported foundation of abstract science including the LC system that follows.

Part three recasts the central challenges of philosophy in light of the discussion in parts one and two. Some of the more brilliant philosophical insights of the past are included as points of reference that guided and (are hopefully) embodied within the LC system that follows.

The practical relevance of the foundations of abstract science

1. The testing of all expressions of concrete science, both theoretical and empirical relies to some extent on the use of some mathematics and/ or logic. At a minimum, arithmetic for mathematics and propositional calculus for logic.

1.1 It is impossible to test concrete expressions without using some mathematics and/or logic.

For example, testing even the simplest statement such as "The color of the ball is red.", presupposes access to tools/rules from so-called logic and mathematics. Get a red color patch and compare it to the ball. The notions of sameness and difference come from logic. Remove them and the color statement can not be evaluated.

For a science example, consider a theoretical economics statement such as " Changes to the money supply only impact inflation" that only makes sense if the terms "changes", "money supply" and "inflation" are defined in mathematical terms. At a minimum this would mean that some mathematical definition of units, such as percents or integers, some mathematical definition of valid operations, such as addition, subtraction, multiplication and division, and some mathematical definition of "impact" such as "Change in money supply for some period in % = changes in inflation in next period in %", was defined for each of the terms.

If the statement were being tested as an empirical claim, one would also need to include mathematical definitions of how changes in money supply, and changes to any other measurable economic attribute were to be measured.

2. If we were to learn now that the rules or laws of any mathematics and/or logic that was used in any concrete science were false, that would mean that all the concrete scientific expressions that used the abstract rule were also wrong.

For example, if we were to learn that the rules of aggregation were in any way mistaken, then all statements of economic aggregates that were previously thought true would now be known to be false. So too would be all calculations of the load properties of buildings, bridges, planes and other engineering structures.

2.1 As such it behooves the concrete scientific community to understand the exact nature of the relationship between abstract rules or laws and concrete scientific expressions. It is furthermore vital for the scientific community to have a detailed understanding of any weaknesses in the abstract sciences (as rules suppliers), that might or in fact do have a significant impact on one or more concrete sciences (as rules consumers).

2.1.1 If it were the case that all relevant aspects of abstract science were already embedded in concrete science and there were no relevant weaknesses, there would be no need for concrete scientists to pay any attention to whatever problems might still exist in the minds of abstract scientists. However, observation of the current state of concrete science would strongly suggest the contrary.

2.1.2 Rather, there are significant problems, endemic to concrete science that stem from problems with the abstract sciences on which they rely. For example:

1. No consistent definition exists of a meaningful or truth-testable scientific expression. In economics, for example, is it meaningful to talk about the marginal utility of a nation? Is it meaningful to talk about the GNP of a company? In politics, does it make sense to speak about the "will of the nation"? What is the form of the constraints that exist between micro and macro political and economic phenomena? Do they adhere to classic part-whole relationships? If not, why not?

2. No consistent definition exists of the rules by which collections of expressions that contain some missing and/or meaningless expressions within them are aggregated or otherwise processed. For example, in database theory, there has been a running debate over the past twenty years concerning just this topic. Some advocate three valued logic; others have advocated four valued logic; still others prefer two valued logic with additional rules for processing the illegitimate expressions.

3. No consistent definition exists for the rules that scientists should follow for describing their discipline's foundations including all type boundaries, all type value ranges, all valid operations per type, all type structures and relevant formulas so as to minimize the likelihood that there are any inconsistencies or under-definitions in the discipline.

2.1.3 There are also significant problems with specific concrete sciences that stem from problems with the abstract sciences on which they rely. For example:

TLH

1. For theoretical linguistics, a formal definition of meaning is not simply a nice feature, it is central to the discipline. What is common to all acts of successful communication such as the dance of a honey bee, or a normal child hearing and responding to a verbal question, or a blind person touching and responding to the same question in Braille, or a deaf person seeing and responding to the same question in sign language? Why is there no operational definition of meaningfulness that spans every possible language use? The current notions of noun-phrase and verb-phrase or subject and predicate that trace to Aristotle but are frequently accredited to Chomsky, or perhaps Sausurre, are known to be inadequate by those who build computer systems that employ so-called natural language techniques. The bottom line is that sentences that work in their appropriate context to exchange information such as "Carry you me[1]" need not be grammatically correct. And sentences that are grammatically correct such as "The country is pink." or "We stand for traditional values.", need not be meaningful. Consider the possibility that either theoretical linguistics is a part of mind and thus is one of the abstract sciences or that the definition of a WFF stems from math and/or logic.

2. For theoretical linguistics and for the study of mind, cognitive science and artificial intelligence, the computational origins of symbolic languages is a huge open question. Specifically, how does symbolic language derive from non-symbolic sensory motor processing? Who's to say that the basis for language, both symbolic and non-symbolic isn't to be found in a better understanding of mapping between different kinds of mathematical spaces?

3. For the study of mind and artificial intelligence, there's no way a solid theory of multi-modal object recognition and understanding will come to pass without the prior understanding of numerous things, including for example, how and why interpreted representational systems are built on top of uninterpreted representational systems which in turn are built on top of non-representational systems, and that all so-called intelligent systems contain all layers. These distinctions within mind are not bound by any particular physical embodiment of an intelligent being and as such are a part of abstract science.

2.1.4 Finally, improvements to the foundations of abstract science would perform a kind of conceptual housecleaning for canonical understanding of math and logic by providing a more consistent definition to such concepts as "infinity" , "irrational" , "number" , measurement, "logical truth or necessity" , "proof" ,

Add:
From the statement Force = Mass x acceleration we can see

A = A

[1] spoken by my 2 year old who associated the expression "carry you" when hearing me ask "Do you want me to carry you" so the expression "Carry you" when spoken by child to adult is a request to be carried and "means" carry me" .When we tried correcting the child by saying" It's not carry you but carry me" the response from the child was "Carry you me"

P > Q Given any two terms we can calculate/deduce the third term
Given all three terms completed we can call any two P and the third Q and then P > Q
Force/acceleration = mass (i.e., we can see arithmetic)
A x B = C

Thus, there is a substantial amount of practical benefit to be derived from an improvement in our understanding of the foundations of abstract science.

Consider, now, some distinctions that need to be understood by the reader in order for her/him to understand the starting point, form, purpose and methods of testing of any purported foundation of abstract science.

Expression systems

The purpose for beginning with this particularly broad metaphor is threefold:

1. Key distinctions required for defining the topic of "Foundations of abstract science", as well as enabling the comprehension and testability of any such theory, as between
   - representational and non-representational,
   - symbolic and non-symbolic,
   - truth-testing and non truth-testing,
   - laws and foundations, and
   - abstract and concrete

   can be explicitly layered within the expression system metaphor.
2. The main exposition which adheres to the expression system metaphor will be simpler to articulate and easier to understand
3. The concept of expression system provides a desirably agnostic connotative stance as to whether subatomic particles alone or collectively are or are not aware.

An expression system is a system that provides for expressions in some form of interaction with other expressions and non-expressions. The two major components of any expression system are whatever interacts with the expressions, and the expressions themselves. Typically, the expressions are very small relative to the expression interacter. They may be thought of as force particles relative to expression interacters. As such, the interaction of an expression and an expression interacter may result in large scale changes to an expression with but limited change to the interacter. As stated, any part of the universe from the smallest sub-atomic particles to any form of plant or animal life to the universe as a whole could be looked at as an expression system.

For example, quarks may be thought of as expression interacters while the gluons they share may be thought of as expressions; molecules can be thought of as expression interacters and the electrons shared between them in a bond may be thought of as expressions. An amoeba may be thought of as an expression interacter with food particles and movement thought of as expressions. A plant may be thought of as an expression

interacter with photons and carbon dioxide as expressions. Normal adult humans can be thought of as expression interacters with the sentences they exchange as expressions.

Representational expression management systems

Representational expression management systems are expression systems where

- Some representation (copy, map, model, transform) of the exchanged expressions rather than the exchanged expressions themselves are contained within the expression interacter (which from here on we call an expression manager) and where
- The expression manager's internal representation of the exchanged expression is itself mapped to a series of functionally distinct purely internal representations.

Speaking loosely about human symbolic language, the third or fourth internal representation is typically called the meaning of the expression. (The first internal representation is the analog copy of the external exchanged expression; the second internal representation is the expression parsed into symbolic form; the third internal representation is the potentially executable parsed symbolic form and the fourth internal representation is the result of executing whatever was potentially executable.) Normal functioning adults are quintessential examples of representational expression management systems, and are capable of sensing and interpreting sentences.

Thus, for example, the sound waves that constitute verbal utterances may be initially represented within an expression manager in a variety of different ways including magnetic analog, magnetic digital, electric charges in transistors, as well as electro-chemical neuronal firings. This first representation defines the identity of the expression-as-object. Hegel might have called it the thing in-itself. The second representation would assign internal type constructs to parts of the internal representation. Sausurre might have called this second representation "the signifier". This second, symbolically parsed representation of the external sound waves may be mapped to yet another internal representation: the so-called meaning of those sound wave. Sausurre might have called this third representation "the signified". It could be anything from a visual image of a mountain top to a tactile image of one's body. That meaning could even be the initial representation of the sound waves as with a sentence such as "This sentence has five words."

Figure 'x' is a depiction of a representational expression system

The constraints on representational expressions are logical, not physical. Anything could be a representational expression. What makes a particular object or motion a representational expression is the expression management system that created and/or sensed the expression.

The foundations of abstract science must provide both a logical and a mechanical explanation for representational expressions.

Analog and Symbolic expressions

Expressions in a representational expression management system may be called analog when the mapping between the expression in question and its source can be defined in terms of continuous one-to-one or N-to-one mappings (aggregation). Our sensory-motor awareness of the so-called outside world may be described in terms of analog expressions relative to the physical phenomena of which they are a result.

Most sensory-motor skill-based behaviours such as catching a ball, throwing a spear, screwing a screw, reading the marks on a ruler or slicing onions or baking a cake, are composed of sensory-motor analog expressions.

Expressions in a representational expression management system may be said to be symbolic when the mapping between the expression in question and its source can not be defined in terms of continuous one-to-one or N-to-one mappings. Our internal representation of word-based human sentences are an example of symbolic expression.. Sentences are composed of parts called words. Specific words may be thought of as actual values of variables that range over specific sub-vocabularies or word sets which are either context-restricted or not. Books and research papers are examples of collections of symbolic expressions. No continuous one-to-one or N-to-one mapping of a visual pattern on our retina such as a snow-topped mountain could ever produce a string of English language letter-based words

Mating rituals are a somewhat atypical behavioural example of symbolic expressions. Body motions are a kind of symbolic representational expression. The critters in courtship may be creating, sensing and/or evaluating body motions. For example the display of plumage or the singing of a song may represent fertility characteristics of a male bird. The dance of a honey bee represents location and content information about sources of food.

The distinction between analog and symbolic is not binary. There may be intermediate forms as the evolution of cuneiform will attest.

Figure ‘x’ shows a representational expression management system

The foundations of abstract science define the constraints that govern both analog and symbolic expressions.

Scientific expressions

The term "science" typically refers to a culture of knowledge acquisition, testing and use. It has grown over the centuries, and depending on one's dispositions may be traced as far back as Sumer or Pre-Helenistic Greece or as recent as Bacon and the beginnings of the Renaissance.

The following section is not intended to be in any way an exhaustive description of what science does, but rather a brief description of the key attributes that one might assign to scientific expressions that distinguish them from other kinds of expressions. These attributes will serve to define the task of any purported foundation for abstract science.

Truth testing

The fact that an internally represented expression-as-signifier successfully maps to another internal representation-as-signified or meaning does not imply that the representation-as-meaning is in any way testable or if tested, true.

When expression interpretations can be tested by re-executing whatever process is required to measure and/or derive them, we say their representational expression system is truth testable

Both symbolic expressions of fact and analog expressions of skill are judged, within the normal course of science, in terms of their truth or correctness.

The foundations of abstract science need to provide a mechanically-grounded definition of truth testing.

Inferencing power

The inferencing power of a true expression is the quantity of observable expressions relative to which it is applicable. Scientific expressions are typically judged in terms of their inferencing power. The greater the inferencing power of an expression, the greater the number of observation expressions that can either confirm or refute the expression in question. Ceteris paribus, given two new expressions, the one with the greatest inferencing power will be treated as the more important.

The expression that "the Earth's Gravity accelerates matter at 9.8 meters per second squared" has a high degree of inferencing power, though it is but an application of the even more general definition of the constant of gravitational force. The expression could be empirically tested every time an object falls to the Earth. Given the life cycle of the Earth and the number of objects that have ever fallen or will ever fall, there are a huge number of observations to which the expression of gravity applies. In contrast the expression "My car is blue", though perhaps no less true, has very low (spatial) inferencing power.

Understanding these general expression regularities is critical – they serve as standards of truth relative to which other observations, inferences, and rules may be judged. New theories, from a social perspective, are free within the bounds of accepted regularities. Sometimes a collection of new observations that contradict currently accepted foundations carry sufficient weight that they force us to alter our expression of those foundational regularities; Kuhn this is called a paradigm shift.

The foundations of abstract science need to be applicable to any representational expression.

Certainty and trust

Scientific expressions are not all equally trusted. Some expressions carry a higher trust so that when expressions of higher trust directly or indirectly conflict with those of lower trust, the ones of higher trust are used to show that the ones of lower trust are wrong or untrue. Although in theory any expressions could be anointed as "certain", in practice, scientific expressions that have a high degree of inferencing power, that have been thoroughly tested (which is easier for expressions with more inferencing power), or that are held by scientists-of-authority or that are held by a wide majority of scientists tend to be treated as most certain.

Expressions that are highly trusted and that have a high inferencing power are typically called laws, constants, or regularities. If there wasn't a large body of relatively constant regularities[2], neither science nor engineering would exist. Bridges wouldn't stand; buildings would collapse; airplanes wouldn't fly.

Add: Bodies of organized expressions. AS disciplines or schools

The foundations of abstract science need to have the highest degree of natural certainty. Social trust in any foundations will only come with successful use over time.

Abstractly versus concretely testable expressions

Comment [ET1]: Page: 14
Clearly, the current trend is to believe that there exists a realm of generality of which mathematics and logic are members and that for these disciplines, at least, core, foundational assertions are true by definition/necessity. As Wittgenstein said, "We can not conceive of an illogical universe." That does not mean that all times and places or universes are logical just that our cognitive machinery is founded on (or systematically uses), certain principles that it is incapable of not using.

The testing of an expression as signifier consists of going to or finding the location specified in the signified representation, evaluating the content and comparing the evaluated content with the content as expressed in the signified representation.

For example given the signifier sentence "Uncle Bob has two heads", the signified representation might be some image of uncle Bob with two heads. Uncle Bob is the location of the signified representation. Testing whether the signifier expression is true amounts to finding Uncle Bob somewhere in the world and then measuring how many heads he has and comparing the measured amount with the stated amount.

[2] (regardless of whether those regularities are attributed to objective, will-less matter or a subjective mind-of-God),

The kinds of types used for finding the location of the signified representation determine the character of the expression and its method of testing. Specifically if there are types that contribute to the location identification that are part of any sensory-motor schemas, the expression and its form of testing is concrete. In other words, the definition of objects in concrete expressions is based on some specific sensory-motor analog expressions be they, for example, visual and or audio.

The operational definitions of finding a book or a courthouse or a tree or a person such as uncle Bob, or a hydrogen atom or a quark always contain some specific analog sensory-motor expression definitions or constraints. We define concrete objects in terms of specific (positive or negative[3]) patterns of sensory-motor interaction.

In contrast, the types associated with finding the location of abstract representations are not a part of any sensory-motor schemas. Although any abstract expression is a concrete object, its concrete representation is arbitrary. There is no specific analog sensory-motor expression that defines the concept of number or truth.

Although both abstract and concrete expressions can vary in terms of their generality, there are significant differences in the characteristics of that generality. No matter how general, concrete expressions still refer to discoverable objects in the world (and their relationships). Empirically testing those expressions requires locating the appropriate objects in the world. The ultimate source of truth for a concrete expression is a measurement of some external phenomena. One might say, "given some measuring scale, it is true by measurement that object A has more mass than object B." And any expression that states a weight relationship between two objects can be tested by using the scale. Furthermore, there would always be a question of measurement precision. Assuming A and B were classes of objects, the statement all A's weigh more than any B could never be fully empirically tested. Ceteris paribus, the more we tested the relative weight of A's and B's the more confidence we would have in the expression. We could never have total confidence however, since we will never see all cases.

In contrast, abstract expressions do not refer to objects in the world but to the workings of internally defined type structures. The greater the frequency with which those type structures are used in expressions, the more general the abstract expression. An expression like 2 + 3 = 5 might be deemed to be very general whereas " –1 = e exp i pi" though no less abstract is less general. This is because the former only requires a single commonly used integer-defining type. Whereas the latter requires a system of integers, rational, reals and imaginary numbers which as a whole are used in less expressions than integers alone.

Testing whether 2 + 3 = 5 requires identifying the appropriate numeric type, in this case integer, and then following the rules of the type to add 2 + 3 and see whether it does in fact equal 5. One might say, "given the rules of arithmetic, it is true by definition that 2 + 3 = 5". Any expression of sums or differences can be tested by appealing to the rules of

[3] Patterns may be negatively specified if an object, say a breadbox is whatever is not a tree. Clustering and other forms of proximity testing are all equally applicable.

arithmetic. What's being tested, however, is not the behaviour of the world, but the behaviour of an abstract constructed type, the internal consistency of that type, its completeness relative to the expression tested, and the testers' ability to use it. For example if one were to test the sum of 2 + 3 and find 6 as the answer, it is likely that the response would be that the tester made a mistake in following the rules of addition. However, when Pythagorus could not find an exact answer to the ratio of the length of the side of a square to the diagonal, it was an example where testing an abstract expression revealed an incompleteness in the underlying rational type.

The foundations of abstract science need to be tested for internal consistency as well as completeness and reducibility relative to any abstract or concrete expression.

Add: modern day "hard sciences" include both empirical and definitional testing. Many objects in hard sciences, especially physics, are principally defined relative to other objects. The science is a network of causal relationships as opposed to merely descriptive ones.

Examples of abstract and concrete laws

Laws are the most general expressions of a particular science.

A + B = B + A
P XOR Not P
If P > Q AND P Then Q
A + B = B + A
The sum of the square of each side of a triangle = square of length of hypotenuse

Comment [oc2]: implies

Atoms with filled outer shells are stable
Force = mass x acceleration
Gravity acceleration = 9.8 meters / second squared
Given a positive price elasticity of demand, increased prices will lower demand.
GNP = Income – savings + investment

Their method of testing varies as a function of whether the law is abstract or concrete.

Typically we have the most trust in laws.

Foundations versus laws, axioms and theorems

Although the most general laws of a science are frequently called foundations, this is a mistake (unless some other term is reserved to designate what are here called foundations). Laws of any degree of generality are composed of truth-testable expressions. In contrast, the foundations of a discipline are the type definitions of the most basic or least derived terms relative to which any expression in the discipline, including its laws, may be created. Foundations need to include at least a description of

the types relative to which the terms used to express laws are values and any rules or formulas that relate values within or between types.

For example, in Economics, the definition of the term "demand" which appears in such general expressions as "Ceteris paribus when prices increase demand decreases" is a foundational expression. Two different groups of economists could legitimately hold different definitions for demand. For example, one group might state "Demand is quantity purchased". Another group might state "Demand is the quantity purchased per time". That difference in foundations would lead directly to measurable differences in how to test a statement such as "Ceteris paribus when prices increase demand decreases".

In mathematics, the definitions of the terms "number" or "proof" or "true" would constitute foundational expressions. In logic, the definition of the connectives "AND", "OR" and "NOT" would constitute the same. In database modeling, the definition of dimension and measure would constitute foundational expressions.

Add some informal examples of foundation statements here

math:

0, 1, successor

or (iteration, unit)

Economics:

Entities: producers, consumers,

Objects: raw material, finished product, money

Attributes: costs, prices, units sold, units demanded, inventory...

Axioms are assumed laws; better call them assumptions.

Testing laws and foundations

Foundations can and should be tested in terms of their internal consistency, and by their completeness and reducibility relative to the collection of actual or potential expressions for which they are a purported foundation.

Comment [ET3]: Page: 17
Odd that the more an expression serves as a standard of truth the less its own truth value is substantiated.

Ordinary science can be done without questioning its own concrete or relevant abstract foundations. In this sense, ordinary science assumes a static foundation. But science is not static. Science can not intentionally evolve without testing and when appropriate altering its foundations. It is not possible to improve a specific concrete science such as

TLH

Economics, and its concept of marginal utility, without putting its foundations under scrutiny and by extension putting the foundations of the various abstract sciences – appealed to from Economics – under scrutiny as well.

Furthermore, all evidence points to human level intelligent systems as having the ability to create significant new type structures both abstract and concrete as a part of their interaction with the world. This implies that the foundations of abstract science need to account for the development of new types either from scratch or as functions of pre-existing types.

Where science break down

Natural and dynamic limits to knowledge

What scientists can not measure

When scientists can't agree on how to test an expression

Social overriding of truth

When individuals use their position of social authority to override the natural authority of some expressions

In corporations,
In governments,
In families

When stakeholders do not (whether or not they can), test the expressions for whose outcomes they are the stakeholders

The problem with representational democracy

Re-framing the central challenges of philosophy

The central challenge of philosophy today is to provide a testably consistent, complete and irreducible foundation for representational expression systems both symbolic and analog with a focus on scientific expressions and especially applied to the abstract sciences including logic, mathematics and mind (which includes language). All other philosophies are either untestable speculation, derivable from, or in need of being consistent with this.

Comment [ET4]: Page: 18
needs shoring up but don't want to put cart before horse

These foundations need to resemble some kind of Type-based representational expression management system. [add why this is so. What other forms might be possible.] Strictly speaking the foundations are abstract since they do not rely on any particular physical embodiment.

Some of the specific questions that such a foundation qua philosophy would have to answer include:

- What rules govern the creation of any type or combination of types (a type structure)?
- How do types naturally vary?
- How are new types created?
- How do types account for their own permissible operations?
- What kind of type structure is required for performing question and answer language games?
- What kinds of type structures can account for wants and how are they managed?
- What kinds of type structures can account for trust and how are they managed?
- What kind of type structure is required for performing propositional and predicate logic games?
- What kind of type structure is required for performing arithmetic games?
- What kind of type structure is required for playing other numeric games that include the rational, reals, and imaginary number systems?

In addition, if such a purported foundation for abstract science is really such a foundation it should further be possible to use the foundations to show the specific kinds of type structures required to account for

- multi-modal learning about and awareness of so-called physical objects
- the human ability to learn symbolic languages,
- feelings and
- the human ability to learn about the world

By definition, such foundations would have a maximum amount of inferencing power and thus could be tested with the exchange of any expression. If sufficiently and successfully tested, such foundations would naturally have very high trust and would eventually be deemed as certain as anything, such as the rules of arithmetic, can be.

Although no complete type system capable of providing a foundation for abstract science has been previously published, numerous deeply insightful hypotheses concerning at least aspects of such a system have been proposed over the millenia under the moniker of philosophy. For example:

- Plato's notion of the world of forms such as "the good" and "the beautiful" may be thought of as generally constructable relations based solely on the components of a general type system.

TLH

- Chuang Tzu's famous aphorism that "A fish net is for catching fish; once the fish is caught you can throw away the net. Words are for capturing meaning; once you have got the meaning you can throw away the words" points directly at the distinction between surface and deep grammar and the type basis for representational expressions.
- Aristotle's definition of the basic form of a syllogism points to the value (and limits), of material implication that needs to be a part of any abstract type system
- Plato and Aristotle both understood that any foundational type system needs to provide for both abstract truth by correspondence with definitions and concrete truth by correspondence with the external world.
- Leibniz understood that some abstract typing system was required to translate different languages and models and conceptions within languages into a common framework so that any kind of debates could be either resolved or shown to be unresolvable.
- Descartes recognized that different expressions carried differing degrees of trust or certainty.
- Hume understood that whether there are necessary causal relations in the world is beyond our ability to know. But from the standpoint of expressions in a representational expression management system, there is no necessary causality between concrete expressions.
- Hegel understood that cognition is a multi-leveled process and that the ability to assert or question at any level presupposes that prior levels are beyond doubt. In other words, one can not simultaneously process expressions while doubting their foundations.
- Kant recognized that regardless of whether there were any necessary causal relations between concrete expressions in a representational expression management system, the mind relies on inferred contingent causal relations to survive. And furthermore, the ability to create a causal relation is hardwired into the mind's basic type system. Genuine discovery can only take place within these definitional structures.
- Mill recognized that it is possible to define mathematical type systems in terms of concrete types. Testing methods and the status of truths are hence empirically based, but for ordinary mathematics he showed that it would be false to assert that one can not define a consistent mathematical type system in terms of objects found in the world.
- Boole: Words refer to sets of things; propositions to the relations between the sets.
- Frege understood that calculation is a form of deduction and that the whole process of testing in the concrete sciences would be thrown off by grounding mathematical truths in empirical observation.
- Russell proposed that not all representational expressions need to denote.
- Wittgenstein understood that the source of universals and abstract relationships is the internal type system or language (or mentalese, or deep structures), not surface grammar or concrete syntax.
- Wittgenstein also understood that the components of a proposition bear a functional relationship to each other rather than a referential relationship to the world. For language to represent the world language must share some of the world's structural attributes.

- Wittgenstein also proposed that primitive mathematical concepts like “number” refer to processes or rules not things or objects.

Although by far the central thrust of this exposition is the articulation of a complete working model of the foundations of abstract science, for the moment, let it be postulated that such a foundation may also serve as the foundation for a variety of adjacent abstract topics including what may be called the foundations of politics, the foundations of morals or ethics, the foundations of religion, (as opposed to theology), and the foundations of aesthetics or art.

In a nutshell, these four areas all share a grounding in an understanding of want or desire and feeling. Wants and feelings are very much a part of the foundations of mind and are treated at length in this exposition. What is deferred to another day is the building out of enough structures to account for the observable complexity of the human social fabric. (The examples in section III focus more on the rational aspects of human behaviour.) That said, some attention will be paid in this exposition to the area of ethics, and specifically a critique of Kant’s notion of duty as separate from compulsion as the source of moral good.

There are of course yet other areas that may be deemed philosophy. Without attaching too much importance to the labels, there is certainly a long established body of work on what might be called theoretically (as in the sense that one could hypothesize an experiment), though not practically testable –thus speculative- cosmology, or theology (or metaphysics). Russell in his book “The problems of philosophy” wrote eloquently about the intrinsic value of contemplating highly general aspects of the universe regardless of whether there was any chance in coming to a testable much less definitive understanding of them. While we fully agree with Russell that there is much value to the individual in spending time contemplating, for example, whether there are other universes that coexist with the one we know, or what happened prior to our most recent big bang, let us be clear that such speculation has no implications whatsoever for any science, whether abstract, or concrete such as engineering and economics.

Section II The LC System

Introduction to the LC System

Recall that we are looking for a foundation for abstract science and that such a foundation needs to have the form of a representational expression management system.

From 100,000 feet, it should look something like the following

Figure 'x'

Stepping a little closer, we can say that the primitives of the LC System are that of

- Types[4] or Domains as collections of values of some unit, with orderings and potential operators associated with those values,
  - Schemas as particular collections of ordered Types capable of supporting the definition and execution of expressions.
    - Additionally, all schemas are composed of two substructures each of which may be composed of one or more types:
      - Schema Locators which define a role played by some Types in a Schema and are akin to the relational notion of primary key, or the natural language notion of subject or the OLAP notion of dimension, and
      - Schema Contents which define a role played by some Types in a Schema and are akin to the relational notion of non-key attributes or the natural language notion of predicates or the OLAP notion of measures.

[4] The notion of Type as an underlying collection of possible values obviates the need for the ontological commitments and associated modeling conundrums that affect ER, Object and other approaches whose modeling constructs are intended to represent certain things in the world. Rather, Types are a model of the language requirements for representing the world, not a model of the world. As such, nothing is claimed to be a Type.

Types can be as easily used to define a (complex) data source in terms of a large collection of simple Types between which many complex relationships exist, or in terms of a smaller number of more complex Types between which a smaller number of simpler relationships exist. The line between intra-Type modeling and inter-Type modeling is dependent on the subjective determination of where to define linguistic surfaces. This representational flexibility is crucial for a data model because the same objective reality or set of data sources can be legitimately modeled in any number of ways, and if a data model is going to stand a chance of supporting BPM, it must be capable of supporting different analytical views of the same underlying data.

- Expressions as particular collections of inter-related types that exist relative to some implicit or explicitly defined schema in both an exchanged and an executable form.

Since all hardwired competencies require some built-in types and schemas, we treat them as equally primitive.

The popular notions of number system, dimension, hierarchy, measure, attribute, variable, data type, network, directed graph, (possible subject and possible predicate in the natural language sense of these terms), and (possible function and possible argument in the predicate logical sense of these terms), may be thought of as specializations of the more general notion of Type.

The popular notions of model, (world -as in possible worlds), multidimensional hypercube, multidimensional multi-cube, Relation, Class diagram, frame, script, system of equations, shape file, process, application and program may be thought of as specializations of the more general notion of Schema.

Within an LC System, expressions are (the thing that, or), what does things relative to types and schemas. For example, expressions

- Assert and question propositions
- Specify and execute calculations
- Modify schemas
- Create new schemas
- Modify non-primitive Types
- Question any aspect of a type
- Create new non-primitive Types and
- Create new primitive Types

Since operators are constrained by the constructs and orderings within the types to which they apply, and since all expressions are defined in terms of types (or other expressions which at some point are defined in terms of types), the limits of intelligible expressions are determined by the type system in place at the time of the existence of the expression. Of course, those limits may vary over time and between spatially differentiated systems.

Seemingly higher level concepts like trust, logical state and want are naturally defined in terms of second order expressions i.e., expressions that take other expressions as arguments and function as subsystems within the overall LC System.

Finally, all explicitly cognitive processes or competencies are defined in terms of systems of schemas wherein schemas may exist in any kind of relation with other schemas (including M-to-N), and wherein expressions in some schemas may query, edit, activate or deactivate other schemas. In their relation to the world, schemas may be non-representational, uninterpreted representational or interpreted representational

Think in terms of abstract mechanical processes such as software

There are basically three places where one could plausibly find the physical embodiment of a foundation for abstract science: in someone's brain, on pieces of paper, or in a software program. The flavor of presentation from this point on is driven by the assumed physical embodiment, (in our case, software).

If it were someone's brain there would need to be all sorts of references to neurons and synapses. The thrust would be either about how all/most persons have the equivalent of a foundation for abstract science in their brains or that people or at least scientists would benefit if they did. There would need to be some predictions about neuronal structures such as if a person doesn't have a particular pattern of neurons that person is incapable of learning division..

If it were pen and paper, there would need to be references to concrete syntax and to some hypothetical basis for executing expressions, since pen and paper can not directly support process execution. Since pen and paper are not an adequate medium for executing processes, it is highly likely that certain shortcuts would be taken in the exposition since there would be no practical way of discovering that shortcuts had been taken.

If it were a software program, there would need to be references to name spaces, parsing conventions, catalog structures, query forms, and physical representations in addition to some concrete syntax. Unlike with pen and paper, any shortcuts taken in the exposition would be easily (though not necessarily), detected: the program would not work as promised.

As a working program, a foundation for abstract science would allow abstract scientists to define arbitrary abstract types and use them in schemas of any degree of complexity. They would be able to query and alter the definitions of types and generate and test expressions made relative to some one or more schemas.

Furthermore, concrete scientists would be able use these same abstract types as units in their concrete typing systems and so inherit the definitional certainty and appropriate degree of measurement-based trust as warranted by their particular activities. Not only that, but the more concrete scientists made use of the same foundation, the more naturally information could be shared, compared and tested within different branches of the same discipline and across different disciplines. Leibnizes dream could be fulfilled.

There would be many ways such a purported system could be tested.

Comment [ET5]: Page: 25
finish with more examples

If any of the great philosophers of old were alive and working today, they would almost certainly take advantage of the logic machines we call computers. As such, this exposition is written in pen-on-paper but with every intention of being embodied as a software program. The reader may therefore notice numerous points where information that is perhaps not essential from a conceptual perspective but essential nonetheless from

an executable process or software perspective has been included. Although we are still working to produce a full "type eninge", over the years, we have implemented multiple versions of various components.

Key challenges to be overcome by any candidate general theory of types

To help the reader get into the mindset of a type system and to think critically about this particular system, it is useful to consider the challenges that anyone trying to construct such a system would need to overcome.

- Type primitives need to be disassociated from physical primitives.
- Two entities that have the same sensory motor schemas and at least translatable root types need be able to use common experience to translate between their type structures
- There must be many different ways to represent the same sensory-motor analog expressions.
- The same operators, orderings and constructs that define the components of a type must also define the components of type structures. In other words intra-type language needs to be the same as inter-type language.
- Atomic operators need to emerge in a consistent fashion from some basic structural attributes of types
- Types need to account for different kinds of adjacencies or topologies in schemas
- Type operations need to account for basic numeric systems but also for hierarchical systems and networks and graph structures. And furthermore be able to incrementally morph any kind of type into any other kind of type.
- Expressions defined in terms of types need to be comparable with expressions defined in terms of type structures
- The complexity of the types making up the logical terms in an expression need to be independent from the complexity of the semantic values of the expression
- Values require some notion of unit; but where do units come from? Need to avoid extra-systemic units or parallel system of units
- Any combination of types should be a valid type structure
- There needs to be some mechanically or objectively measurable properties of type structures that determine whether they are usable schemas

Thinking critically about the proposed system

Test these assertions yourself

Tests of internal consistency for any proposed foundation

Tests of external correspondence

Expository style

TLH

The major artifacts presented in this section all need to exist in order to build even the simplest of representational expression management systems. The order of presentation was chosen to facilitate comprehension.

As regards expository style liberal use is made of examples. This forces the incorporation of structures on an illustrative basis because every example has at least the complexity of a representational expression. And everything presented in this section is a part of any representational expression. For example, showing that types as defined support basic assertions/negations and calculations make use of expressions though they are not defined until the subsequent section. If the order had been reversed, expression examples would have been given though no types had yet been defined... It's like an economy. All the basic pieces co-exist.

System primitives

Types are like a generalized form of number system[5]. They include some notion of valid or potential values, some notion of valid operations and some other stuff that will be described below. Suffice it to say that any understanding of types begins with an understanding of their composition or primitive components.

Before launching into a detailed description of the LC System's primitive type components, or simply primitives, it is important to understand how these primitives relate to each other.

The three type components, namely constructs, ordering relationships and operators are each projections of a single underlying unity. The token-based separateness of such seemingly distinct notions as, for example, "word", "addition", and "hierarchy" masks a deeper unity.

The purpose of the following section is to point towards that unity.

Add: definition of term "model", "schema"

Constructs

The major constructs in any model are the primitive linguistic objects that enter into relations with other linguistic objects. All expressions made within a model are in terms of the constructs of that model. For OLAP models, the major constructs include that of Dimension, Measure, attribute, cube and instance. In the LC System the major constructs include the notion of Type and Type Structure. And within Types there are the notions of unit and value. Although it is possible to write or say the term "Type" or "Type Structure" or "Unit" or "Value" independent of uttering any other term, it is not possible

[5] Depending on the reader's background, the closest points of reference are Type Theory or Category Theory

to define any one term absent referring to some other term. For example, any definition of the term "Type" would need to make reference to the terms "Unit" and "Value".

One of the basic characteristics of a Type is that it must define some set of potentially usable values, called potential values. A "Sales" Type might be defined as a dollar value greater than or equal to zero. A "Product" Type might be defined as any 8 Byte Char. Regardless of what kind of Type is defined, there is an implicit assumption that the values of the Type are distinguishable. The Sales value "$100" is not the same as the Sales value "$150". The Product value "Shoes" is not the same as the Product value "Hats". If values could not be distinguished, Type definitions would be impossible. Yet, the notion of value comparisons and of equality, is not itself a construct. Rather the notion of value comparisons is a primitive operator or function. Thus, the specification of primitive constructs presupposes a well defined notion of primitive operators.

Another basic characteristic of a Type is that it must be possible to traverse its potential values. There need not be a sequential ordering as there is for numeric series, but there must be some way to move from value to value. If the potential values of a Type could not be traversed, it would be impossible to call a function that changed the value of a Type based on some condition. For example, the ability to add $5 to the current value of $20 of a Dollar Type that represents an hourly wage presupposes that the value $20 is connected to the other dollar values of the Type so that it can be incremented by $5 to yield $25. As such, the specification of primitive constructs presupposes a well defined notion of primitive ordering relationships.

Thus, the primitive constructs of a model/system presuppose specified ordering relationships, and functions in addition to other primitive constructs.

Ordering Relationships

Every Ordering Relationship presupposes both the constructs that are ordered, as well as the constructs used to specify the ordering relationship (typically integer values as in the expression "1 to N"). Furthermore, every Ordering Relationship is expressed in terms of a particular function, namely the function that traverses from one construct in the ordering relationship to another. In other words, in a hierarchical ordering expression of the form "1 to N" the word "to" masks an implicit function, namely that traversing from the "1" to the "N" requires a single step or unit delta in the down hierarchy direction. While traversing from any "N" to the "1" requires a single step or unit delta in the up hierarchy direction.

Thus the specification of Ordering Relationships presupposes the existence of both constructs and functions.

Operators/functions[i]

Consider a function or operator as seemingly primitive as comparative equality or "=[6]". We might say, for example, that Sales = Costs. But can we define the function "=" absent referencing constructs and Ordering Relationships? Here again, we can not.

For example, any definition of the equality operator would need to reference those Constructs whose equality is being tested or asserted. In addition, there is an implicit Ordering Relationship in the equality function, namely that of "1 to 1". In other words, and continuing with the example above, one value of Sales is equal to one value of Costs. If there were more or less than one value from each of the two Constructs, the function would not work as defined.

Thus, the specification of Operators, like Ordering Relationships and Constructs, presupposes the existence of the other primitive concepts as a part of their very definition!

Schemas and Expressions

It has just been shown how the primitive components of a type in the LC System are fully interdependent. This is why it is more accurate to look at each of the primitives as a projection of a deeper, underlying unity[ii].

Furthermore, not only are the primitives themselves interdependent, but all uses of types in expressions (which are the only way types get used), rely on the combined performance of all three components. It would be impossible to use a type in an expression while only leveraging the construct or ordering relation or operator component of the type.

For example, and considering for the moment just query and assertion expressions:

The color of the house is green

Type name: Color
- Values blue, green, red
- Ord rel: categorical
- operators: assertion, negation

Type name: Shape
- values: round, tree, house

Expression: Shape: house, Color?

For shape: used construct, ord rel and operator

[6] Though in most programming languages "=" denotes assignment and "==" denotes comparison

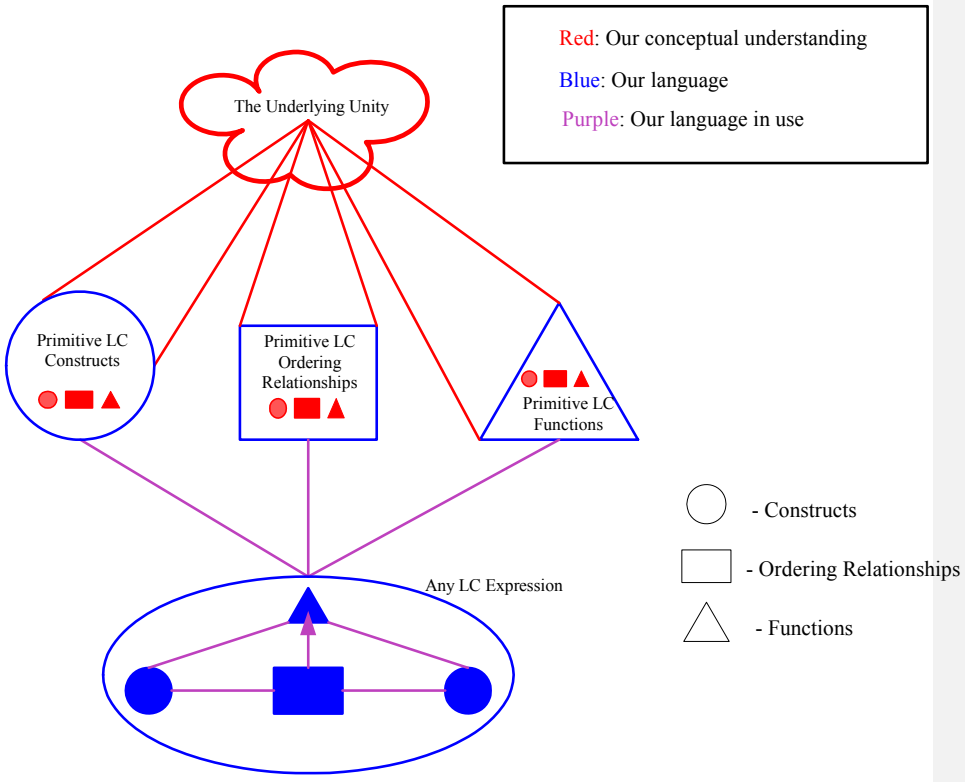
For mass: used construct, ord rel and operators

The weight of the bushel is 5 Kg

Thus, expressions are created through a relationship between types with each type utilizing all of its primitive components: construct, ordering relationship and function sub-expressions all of which are equally primitive.

Figure 4 illustrates the relationship between the primitive concepts, their underlying unity, and expressions.

Figure 4. The Primitive Concepts, Their Underlying Unity, and Expressions



A General Theory of Types: Part I of the LC Kernel

In the previous section, we saw how constructs, ordering relationships and functions are the three projections of a single underlying unity. They form the primitive building blocks of the LC Model. In this section, we describe these primitives in detail

Constructs

To facilitate a layered comprehension, the primitive constructs are described in several passes. We

1. Mention the terms
2. Assign short rough descriptions to each of them
3. Describe key distinctions between constructs
4. Give a more precise definition to the constructs

The LC System defines Type Structures, Types, Units, Values and Instances as primitive constructs.

Simple descriptions

- Type structures are a "first class construct"[7]. They are the largest construct (within a single method of representation) and are composed of two or more Types (between which one or more ordering relationships exist), and their associated values-in-use (also called actual values)
- Schemas are a "specialization"[8] of Type Structures that contain at least one Type used as a Locator and at least one Type used as a Content
- Types in isolation (or not in use within the context of a Type structure), are a first class construct. They are a named group of two or more potentially usable values belonging to at least one unit
- Units (themselves a role played by a Type), define a range of potential values and permitted functions and are associated with one or more Types
- Potentially usable Type values, called Potential Values in the LC Model, belong to units and are associated with Types in isolation
- Actually used Type values, called Actual Values in the LC Model, (frequently called data), belong to units and are associated with Types in use within the context of a Type structure

[7] A software term meaning that the object is directly referenceable within the catalog.
[8] An object-oriented software design term meaning that the specialized object has all the properties of the more general object thus all specialized objects are valid instances of their general object. And a specialized object has additional properties as well.

- Type Instances, called instances, are what remains of actual or potential values when the value's unit has been separated out[iii]
- Type Structure values are the collection of "Type_name-Unit-Value" triplets associated with each Type in a Type Structure for some instance of that Type Structure
- Type Structure instances are what remains of Type Structure values when the units belonging to any of the values have been separated out

Typical examples of stuff that might be most efficiently represented as

- *Type Structures* include: OLAP cubes, Relational Relations, SQL Tables, Documents, Workflow processes, and maps
- *Types* include: Time, Geography names, Employee names, Currency, Mass, Companies, Sales, Costs, Quality, Integers, Character strings, Flat Indexes, Booleans
- *Units* include: Day, week, month, year, City names, State names, Employee names, Dollars, Euros, Yen, Kilograms, Pounds, Companies, Normalized indicator, Integer, Character string, Boolean
- *Potential Values* include: Month.January, City.Cambridge, Employee.Bob, $250 , 67th , 100lbs as found within the definition of a Type
- *Actual Values* include: Month.January, City.Cambridge, Employee.Bob, $250 , 67th , 100lbs as found within the definition and/or instances of a Type Structure
- *Instances* include: Cambridge, Bob, 'x' , 250 , 67 , 100
- *A Type Structure Value* includes: {Time.Month.January , Product.Brand.Quaker Oats, Sales.$100}[9]
- *A Type Structure Instance* includes: { January , Quaker Oats , $100}

Explanatory Distinctions of emergent properties

Since the terms best used to describe emergent properties of types resemble a decision tree of distinctions stemming from the general constraints on any type, the material that follows will be framed as a cascading series of distinctions.

Please note that from a software perspective, these conceptual distinctions do not require distinct creation statements. Ockham can rest easy. For example, one would never create a compound versus a simple type, or a single versus multi unit type, or a multiple Ored unit type with a hierarchy. These distinctions are implicit in the way the Types are defined; they are distinctions within a single general form. They are discussed here to show how a few simple concepts can account for a wide range of observable type phenomena.

.

1. Constructs in Definition Versus Constructs in Use

[9] Assuming a Type Structure with the Types Time, Product and Sales

TLH

Add and show that the only reason to provide two terms is that there is a possibility for ambiguity

One of the complexities that surfaces when discussing Types is that some of the time we're defining formulas for actual values and some of the time we're defining formulas for potential values. Sometimes potential values come from other potential values; sometimes they come from actual values. We need to clearly distinguish between actual and potential values.

Let's start with a simple analogy: natural language. Sentences are composed of words (and punctuation which help the parser and compiler). All the words used in any sentence must be a part of our vocabulary at the time they are used. Our vocabulary or collection of usable words is like potential values. The sentences we utter are like expressions composed of actual potential values.

If things were this simple, we wouldn't need a Type model. But they're not. For example, how do we account for changes to the vocabulary such as new words? This is equivalent to adding a new potential value to a Type or even defining a new Type.

Take a simple formula such as "Profit = Sales – Costs" that holds true for some schema, and assume that all three Types are defined in terms of unit dollars. The unit dollars determines the potential values for each of the three Types. Any possible dollar value is a potential value for Profit, Sales or Costs. But within the example schema, the formula for profit states that the actual value for profit is equal to the actual value for sales minus the actual value for costs.

When we are defining the potential values for a Type, we are defining the Type as it exists in the catalog or definitional or simply Type space. The Type, as construct, is in definition (or a state of being defined). The definition of a Type, means, among other things, the definition of its potential values. When a Type is used within the context of a schema, the values that appear in the schema are actual values for the Type. Even in the extreme case when the values that appear in the schema are identical to its potential values, we still say that the schema contained actual values for the Type. (By design, the actual values specified matched the potential values.) It is important to note, however, that although the values may appear identical, their logical connectives -not typically captured in a software tool- are different. As stated in the Type's definition, the potential values coexist in one big exclusive OR. If all the potential values happen to get used within some Type structure that collection of actual values are ANDed together – though when we get to an expression, they are Ored again.

2. A Type Versus a Type Structure

It is common to think types as one dimensional and ontologically or objectively distinct from type structures. One would think that a type like "integer" or "Color" is fundamentally different from a type structure such as an econometric model of interest rate changes that has time, country, bank, inflation, trade balances, currency exchange

and other dimensions as components of the model. But it is not the case. It is cleaner to think of the difference between types and type structures as emerging from two different ways that a multi-unit type whose units are ANDed together may be treated.

A type's complexity and its logical dimensionality are independent of each other. A type could have a very complex ANDed unit structure yet the units could be bound together and the type would be unidimensional. Any type having at least two ANDed units could be treated as a type structure.

Debated using somewhat different terms: typebase as the general with type and type structure as the two particular forms of casting. Or retain type as the general and use unitype and type structure as the two castings. Here we advocate the latter. But in the interests of readability, we use the term "type" to mean "unitype" and the more general "unitype or type structure" unless there is ambiguity in the specific context.

No matter how simple or complex a unitype may be, a value of a unitype could never define a schema or possess a truth value. Truth values are the lifeblood of business rules and other integrity constraints. The values green of Type color or the value $5000 of Type Sales could never be true or false in isolation.

In contrast, all and only those type structures that have a schema form as shown in the next chapter, define truth-testable schema-specific expressions.

3. A Type Structure and a Schema

Every ordering relationship between two or more Types defines a Type Structure. All and only Type structures that are composed of at least one Type used as a Locator and at least one Type used as a Content (as described above in section 4.2.1.2), define schemas.

4. A Type and a Unit

The concept of unit denotes the set of potential values that may exist and operations that may be performed. In simple Types that have only one unit, the name of the Type typically denotes its units. However, in many if not most cases, a Type has several, possibly hundreds of units. Consider, for example, a currency Type with hundreds of different currency units.

In the LC Model, all non-primitive Types must have at least one unit defined in terms of a previously existing Type. In this sense, the concept of unit is that of a "previously defined" Type being set equivalent to the potential values and associated operations of a newly created Type. The concept of unit also provides for much of what is commonly passed along through inheritance. This contrasts with the typical view wherein units are a separate construct from Types requiring separate maintenance and possibly coexisting with data types as a source of potential values[iv].

5. Implicit Units and Explicit Units

Explicit Units are either explicitly defined in terms of some previously defined Type or they are defined as a specialization of a previously defined explicit Unit. For example one might state that a new Type "Sales" is defined with Units "Dollars". The Unit "Dollars" is an example of an explicit Unit. Or one might define a Type "Geography" with Units "CHAR [8]" and then define a set of specializations such as "Country", State" and "City" as explicit specializations.

Implicit Units are specializations of existing explicit Units as determined by the explicit hierarchical relationships of the values in the explicit Unit. Thus, for example when one defines a Type such as "Reporting Hierarchy" with an (explicit) Unit "CHAR [16]" and then reads in a parent-child table, that parent child table specifies both the set of values belonging to the unit "CHAR [16]" of the Type "Reporting Hierarchy" and the full set of hierarchical relationships between those values. When assembled, the set of hierarchical relationships implicitly defines a set of specializations of the unit "CHAR [16]" wherein each unique specialization has a unique hierarchical relationship between the root and leaves of the hierarchy and is defined in terms of three numbers: distance from the root, minimum distance from the leaves, and maximum distance from the leaves.

6. Root Types and a Non-root Types

Root Types are defined without reference to other Types. Although the only hardwired primitive Type in the LC Model is a Rank Type, the usual suspects are defined within this document as "early derivations" and include such standards as categorical, numeric, and periodic. Other popular kinds of Types such as leveled and ragged hierarchies do not need to be defined as Types. Rather the set of functions associated with these constructs become available on the basis of certain (hierarchical) ordering relationships that a user might define.. Primitive Types do not reference any other Types as units. They are themselves units and as such perform all the functions associated with units.

All non-primitive Types are defined in terms of at least one previously existent Type that functions as a unit with respect to the newly created Type.

Comment [ET6]: Page: 35
be clear about whether any types are hardwired in the LC System

7. An internally differentiated Unit and an externally iterated Unit

There are two basic ways by which a unit supports and constrains a collection of potential values. Typically, we think of units as externally iterated over (and internally undifferentiated). For example a measurement in terms of the unit kilograms would be expressed as some number of iterations of that unit. An actual weight might be 1KG or 2KG or 3KG. Think of the standards "kilogram mass" "meter length"...

In contrast, units can also be internally differentiated instead of being externally iterated over. For example, the notion of a CHAR as a domain for Relational attributes, works as an internally differentiated unit. The unit is specified as an outer AND between such as

bytes consisting of a collection of inner XORed elements such as bits. If the unit is one Byte it can be represented as eight binary XORs that are ANDed together. At a logical level, we typically define categorical Types such as Product name, employee name, and place name with internally differentiated units. While we define numeric Types such as Sales, Costs, and wage rates as externally iterated. At a physical level we typically mix up both forms of units[v].

At an abstract level, for a collection of values to serve as potential values for a Type, there must be something that is constant across all values (which is what we call unit) and something that is unique to each value (which is what we call the instance).

8. A Unit and an Instance

A unit defines a set of possible differentiations and the possible operations that apply to any instance. An instance is any single differentiation. If the unit is integers, then 10, 12 and 23 are examples of instances. If the units are days then Monday and Tuesday are examples of instances[vi]. If the units are countries, then France and Lebanon are examples of instances.

9. An Instance and a value

A value is an instance in conjunction with its unit. The terms 100 and 200 in conjunction with the unit "Dollars" are values. The terms 100 and 200 in isolation are instances. The distinction is crucial in practice because sometimes the unit information is embedded within the value information (as when an attribute in a SQL table is of known units, say dollars). And sometimes the unit information is stored separately (as when measures of mass are given in one column and their respective units are given in a separate column.)

10. A simple Unit and a compound Unit

A simple unit has no constituent parts. A compound unit is composed of two or more units anded together in some expression. For example, Speed as the ratio of distance to time is a compound. Date as the juxtaposition of day month and year is a compound.

11. Anded Units and XOred Units

Whereas Anded units must coexist for a single value, XOred units must not coexist. Specific currency units might be Ored together in a currency Type. Any currency value would be expressed either in Euros or in US Dollars or in Yen etc..

12. Types with multiple implicit Units and Types with multiple explicit Units

Parent-child relations are the classic example of a Type with multiple implicit units. Typically, the set of parent child relations is given. When traversed, it produces a ragged hierarchy. The links in the hierarchy are between the potential values of the Type. The

implicit units are the distinct collections of distances from any value to the root and then the nearest and farthest distance to the leaves. The implicit units uniquely identify the relative hierarchical placement of every value. We don't typically concern ourselves with these units. However, patterns between the implicit units identify the degree of raggedness in the hierarchy. And this information can be used to suggest particular groupings of values whose hierarchical placement is similar in some way. For example, all values two down from the root and more than one up from a leaf may be identified as a reporting group.

In contrast, a leveled hierarchy composed of multiple units Qua levels such as city-state-country has explicit units. The creation of the hierarchy may have even been defined prior to seeing any particular values. For example, someone may have defined a leveled hierarchy wherein for each country there were multiple states and for each state there were multiple cities.

For Types that have multiple Units ORed together we can distinguish between those

- That are connected through a sequence of increasing or decreasing unit sizes
- That are connected through a hub-and-spoke or peer to peer translation

Hierarchies represent a collection of ORed units that can be arranged in order of increasing or decreasing unit size.

Hub and Spoke unit relationships have a central unit, the hub. And they typically have a number of spoke units. Spoke units translate into any other spoke unit in two steps. One step is into the hub unit. And one step is from the hub unit to the target spoke unit. For example, currency units as they existed during the time of the Gold Standard between Bretton Woods and 1973 or as they existed during the initial phase of the Euro for European currencies resemble a hub and spoke. During the time of the Gold Standard, Gold was the hub unit. Every currency was a spoke unit. Likewise during the early phases of the Euro, the Euro was the hub unit and each individual European currency was a spoke unit.

Peer to peer unit relationships may have a bunch of units any one of which can translate into any other. At any time, all the units can be represented in terms of any other. So when a new unit is added, it need only be connected to one existing unit. The rest is automatic. For example, currency translations between Euros and other non- Euro currencies have a peer to peer form.

Ordering Relationships

What Ordering Relationships Specify

Ordering relationships specify traversable adjacencies between constructs. For example, when the values of a Type are hierarchically ordered it means that from any value you can traverse to a set of adjacent child values or to a single adjacent parent value. That for all States and for all Cities there is a 1 to N relationship between States and Cities.

Comment [ET7]: Page: 37
do more than just this

Some Ordering relationship specification is common in the OO world. Specifically the definition of how many instances from some class are paired with how many from some other class. The OO world suffers from two limitations, however. First, the variety of ordering relationships that can be specified between classes is generally much richer than the set of ordering relationships that is specified within classes. In contrast, the LC Model provides for the same set of ordering relationships within as between Types. Second, the specific grammar of ordering relationships provided between classes is limited to the basic notion of cardinality. What gets specified is how many of some class are paired with how many of some other class. In contrast, as described below, the LC Model provides for additional and equally general ordering relationship specification criteria[vii].

Primitive LC Ordering Relationship Specification

Every ordering relationship is composed of some number of construct scoping functions where a scoping function selects the specific subset of the potential values of the Type that may participate in any instance of the ordering relationship. It might be that all the potential values of a Type participate or perhaps that only those values that are descendants of a particular value. Although for ease of exposition in this document, there are frequently only two constructs participating in an ordering relationship, there are many cases such as in the specification of Cartesian products or hierarchies that three of more constructs participate in a single relationship[10].

Quantitative Aspects

For each scoped construct, such as “All Stores” or “All Days in 1999” an ordering relationship needs to specify how many values from the scoped construct participate in each instance of the ordering relationship. For each instance of a State-City relationship, how many States participate and how many Cities participate? Typically, we would say that for each instance of the State-City ordering relationship there exists one value drawn from the State Type (or unit) and N values drawn from the City Type (or unit).

In the State-City example, the number of values drawn from the construct “State” is constant and equal to the number 1 for all instances of the ordering relationship. In contrast, the number of values drawn from the construct “City” is variable and could vary from zero to some arbitrary maximum.

Thus the specification of “how many” is represented by either a single integer, or an integer range or an integer variable. So what does it mean to say that some non-zero

[10] This is why we use both infix and function-argument styles of concrete syntax.

quantity of some construct is connected to zero of some other construct? It means a lot and happens all the time. For example, the leaves of a hierarchy connect to zero values in the down direction; and roots of a hierarchy connect to zero values in the up direction. Last siblings connect to zero values in the next sibling direction. First steps connect to zero values in the previous step direction. In short, finite and terminating (or determinate sized) ordering relationships have edges. And one of the ways that an edge is defined is that movement in the same direction that resulted in an edge value is no longer possible; i.e., the edge value has no neighbors beyond the edge.

Uniqueness Aspects

A useful analogy for thinking about the specification of ordering relationships is that of drawing balls from a container (or drawing cards from a deck). In the previous section we saw how one aspect to the specification of an ordering relationship is how many balls are drawn for each instance of the ordering relationship. Another, equally critical aspect is to know whether those balls are drawn with or without replacement between instances of the relationship.

In the State-City example above, (and imagine two containers of balls – one whose balls are labeled with State names; the other with city names), the question arises, between instances of the State-city relationship can the same State or the same City occur? If the answer is no, then we can say that the values for state and for city are drawn without replacement. However, if the same City value could occur in conjunction with more than one State (which does occasionally happen), then we would say that the values for City are drawn with replacement across instances of the State-City ordering relationship.
In the relational world, the ordering relationship between the attributes (columns) of a Relation or SQL table is one-to-one for each instance of the ordering relationship (AKA Relation tuple or row). There is exactly one instance of each attribute. However, any attribute value can repeat across rows. The only constraint is that each row is unique as a whole. Thus, the notion of "drawn with replacement" is critical for the modeling of a Relation. And for schemas in general, (discussed above in section 4.2.1.1 and again below in section 4.2.4), the difference between the Types used as locators and the Types used as contents is that the locators, as a whole, are drawn without replacement relative to the contents each of which is drawn with replacement.

An additional factor to consider when there is more than one Type whose values are drawn with replacement is the relative degree to which the values repeat. If the values have a minimum amount of repetition, it might be that the values are correlated and resemble a scatterplot. On the other extreme, if the values have a maximum amount of repetition, their relationship is typically called a "Cross-product" or "Cartesian product". Thus,, the notion of "drawn with replacement" is critical (though implicit), even for vanilla OLAP.

In short, uniqueness is a critical aspect to ordering relationships rarely if ever dealt with in the context of information modeling that is nonetheless required to explain a wide variety of observable data relationships.

Yet one more factor that can be taken into account is a logical extension of the above; namely whether any repetition is allowed for values within a single instance of an ordering relationship. This factor is only relevant when there are two or more values drawn from a Type per instance of the ordering relationship. Although such relationships are rare in the classic BPM world, when we move beyond BPM into areas such as Bioinformatics and try to model the physical or life sciences, examples abound. In chemistry, for example, if one were modeling molecular bonds as ordering relationships wherein the molecules were defined as collections of values drawn from a single atom-defining Type, anytime a single molecule contained more than one instance of an atom, there would be a need for repeating values within a single instance of the ordering relationship.

Adjacency Aspects

The concepts of unit and value are the basis for a critical primitive distinction between kinds of adjacencies. One kind of adjacency is between values within a given unit which may be called positional adjacencies. The adjacencies between two integers or two categorical values are examples of positional adjacencies. In contrast, another kind of adjacency is between two units of the same relative value which may be called resolutional adjacency.. The adjacency between a city and the state to which it belongs or of a day and the week or month to which it belongs are examples of resolutional adjacencies.

Although it is common practice for data models and for software products to assume either that all 1-N relationships are resolutional, or that all relationships that can be modeled in terms of 1-N cardinality relationships can be treated in the same way, both assumptions are wrong. Regarding the first assumption, consider the following examples.

The Relationship between Months and days, and between organs, tissues, cells, molecules, atoms, and sub atomic particles, are all 1 to N and resolutional. Focus on a single cell, increase the resolution of the image and you will see N molecules. Focus on one month and increase the resolution and you will see 28-31 days…

However, the relationship between a table and its associated chairs or between a car chassis and its doors or a teacher and her/his pupils or between a process step and its next steps is also 1 to N but it is not resolutional. The table and chairs are all of the same resolution; the teacher and pupils are all of the same resolution; the process steps are all of the same resolution. Blow up the teacher and you'll see teacher organs not pupils. Blow up the table and you'll see table parts not chairs. Blow up a process step and you'll see sub-steps, not next steps. These relationships are 1 to N but semantically they are positional, not resolutional. In other words they coexist within the same level of granularity.

The reason why you can't treat all 1-N relationships the same way (and thus why the second assumption is also wrong), is that from a single value, say some process step in a BPM process, there needs to be some language/model-based way to distinguish moving to sub-steps (increasing resolution for the same relative position) versus moving to next steps (changing position for the same relative resolution).

Operators/functions

The LC Model recognizes as few primitive concepts as possible. Such things as arithmetic and hierarchy referencing, though basic in any normal sense of the term, are a part of the outer or non-primitive layer in the model and as such are not presented until sections 4.3, and then all of sections 6 and 7 where the outer layer of the model is described in detail.

Outer layer concepts will be implemented as physically optimized built-ins from a Bailey perspective.

What Operators Specify

Where functions are used to specify content (or measure) relationships, they are typically called calculation functions. Where functions are used to specify location relationships, they are typically called referencing functions. Consider, for example a cube defined with "Geography" and "Time" as dimensions, and "Sales", "Costs" and "Profit" as measures. Assume also that "Time" supported numeric functions. Now look at the following expression

Profit, Time.this = (Sales, Time.this –1) – (Costs, Time.this)

It would be typical to refer to the minus sign in the sub-expression "Time.this-1" as a referencing function while the minus sign that indicates that Costs are to be subtracted from Sales would typically be called a calculation function (albeit a simple one).

Primitive Operators in the LC Model

The primitive operators of the LC Model are applicable to all Types. All primitive functions coexist with their inverse. They are presented as Function / Inverse.

- Assert / Negate

Frequently ASSERT is assumed. For example in a SQL Dimension table, the attribute values that appear in the attribute columns are being asserted of their respective dimension values. Yet, no special syntax is deployed to state that the attribute is being asserted of the dimension. We typically only explicitly state NEGATE. The negation of

a negation of an assertion needs to equal the assertion.[11] To state is to assert. The notion of complement is synonymous with that of negate.

- AND / XOR

Whether two events or structures or states or processes or values must co-exist (co-occur)or must not coexist (co-occur). Co-existence may be positional or resolutional; spatial or temporal. For example, one can say "A AND B" where A is an event at some time position and B is an event at some other time position. Or one can say "A AND B" where A is an event at some space position and B is an event at some other space position.

- Measurement/Manipulation

Is the value of a variable being tested/queried/measured/read or is it being specified/assigned/determined/written?

- Equality / Inequality

Whether two values are comparably the same or different. This doesn't state how the equality/inequality is determined, or whether the equality/inequality is being tested or specified.

- All/ Some[12]

If some fact is true of all locations then it is false of none. All and none are oppositely stated equivalents. Rather, the notion of Some is the inverse of All and could be restated as Not All.

What are often thought of as distinct functions and their inverses, namely asserting versus negating, ANDing versus ORing, measuring versus manipulating, being equal to versus not being equal to, and All versus Some, are more properly understood as co-determinate.

This is because,

- asserting some variable is equal to some value of some Type which converts the collection of XORed values of the Type into some one value being asserted (and to every other value being negated),

is equivalent to

- asserting that the variable is not equal to the ORed connection of all values other than the one asserted..

For example, to assert that some variable is equal to the color green (and assuming that the color Type has only three colors, red, blue and green) is equivalent to asserting that the variable is not equal to Red XOR Blue.

In short, if we are going to postulate any primitive functions, the notions of assert/negate, AND/OR, Equals/Not Equals and All/Some go together.

[11] Assuming no invalid data; an assumption we lift in the section on expressions.
[12] This distinction of course, is that between the universal and existential quantifiers.

In addition to the primitive logical functions, there needs to be some notion of sequence order for expressions to be executed. At a minimum there needs to be a notion of next step and its inverse previous step.

Comment [ET8]: Page: 42
finish

The General Form of a Type

Having seen the primitives in pieces, so to speak, and having examined a few important distinctions, let's put it all together and look at the general form of a type. There are four parts to this description:

1. The top level constraint that the subsequent statement of general form needs to meet
2. The statement of constraints that defines the general form
3. The justification for each constraint in terms of the top level constraint
4. The description of a decision tree that shows how more complex types naturally arise within the variability and constraints provided for by the general form

Top level constraint

Since there is no way to define necessary attributes or properties of a type independently of some set of criteria, consider, first, those top level constraints. Internally consistent. External constraints relative to known abstract practice – a turing-like constraint – litmus test.

The type system of any foundation for abstract science must support at least the rational number system, the propositional calculus and at the first order predicate calculus. This implies as a necessary but not sufficient condition, that Types must provide the ability to define assertions and negations.

Specifically, this criteria means it must be possible to assert or negate that some value of some type is true or false. Thus, for example, given two types 'Store_name" and "Sales", if the potential values for "Store_name" are "Geneva" , "Paris" , and "Madrid" and the potential values for "Sales" are "Dollar values greater than zero", the following informal expressions represent assertions and negations of some sales value of some store:

Assert: Madrid Sales = $250
Assert: Paris Sales = $300
Negate: Geneva sales = $350

Given this global criteria, the following constraints apply to any well formed Type.

The constraints on any type

Testing the general definition of a Type

The reason why these constraints follow from the global criteria is described below with reference to each of the above listed constraints.

1. Any Type must have two or more potential values

If some type, Tn had only one potential value, v_n, there would be no possibility of creating propositions where Tn served as a predicate capable of assertion and negation.

For example, given a type called color with two colors green and red the condition states that green can be true of something such as the color of the sea if and only if red is false of the color of the sea. Thus, if the type color could only have or assume the one color green, there would be no way to say that something had a color other than green or that if green is the true color then some color other than green is false. As a result, Booleans or binary types are the simplest possible types.[1]

2. There must be at least one unit to which the potential values belong

Absent a unit specification, there would be no way to determine whether any particular token was or was not a valid value of the Type. Nor would there be any way to determine whether any particular calculation was valid. In short no grammar checking could take place for any use of the Type in an expression.

Furthermore, you can't get around units. It is impossible for a value to not belong to a unit.

Thus although we typically represent a Type as in expression 1 below,
Type Name: City
Values: Chicago, New York, Boston

a more accurate description would resemble expression 2 below.
Type Name: City
Units: 8 Byte CHAR AS City
Values: Chicago, New York, Boston

Expression 2 reinforces the idea that every value in the Type belongs to a unit. Now consider the simple month and year, and meter and kilometer dimensions below.


```
Type Name: Time
Units: Month, Year
Values, Time.Month:  Month.January, Month.February,
..Month.December
Values, Time.Year: Year.1998, Year.1999, Year.2000

Type Name: Distance
Units: Meter, Kilometer
Values, Distance.Meter: Meter.1,Meter.2,… Meter.1000
Values, Distance.Kilometer: Kilometer.1, Kilometer.2
```

It is normal to think of meters, kilometers, months and years as units. Everyone knows that 12 months make up one year and that 1000 meters make up one kilometer. The ratios of 12 to 1 and 1000 to 1 represent the scaling factor between the units of the time and distance Types. Thus if we have some value for a year we know to divide it by twelve to calculate the average value per month. And if we have three month's values we know to multiply them by four to estimate a value for the year
Think about what you get for knowing the units of a measurement or value. You get being able to figure out scaling factors between levels of granularity, and you get a set of permissible operations. Meters can be added or subtracted and multiplied or divided. They can be divided by units of time. A glacier might move at a speed of 3 meters per year; a bureaucracy may be even slower..

Consider what you lose if you don't know the units. How would you add 5 'unknown-length units' and 3 'unknown-length units'? There is no operation you can perform with the two different measures of length if you don't know the units with which they are measured. **Thus, you can't separate the translation aspect of units from the permissible operations aspect of units from the inclusion rule aspect of units.** (If you have 24 odds and ends on a table, and 23 fall off, are you left with an odd or an end?)

Although we typically associate the concept of unit with Types whose values are numeric, the role of defining value membership rules, providing for quantitative comparability between scaling factors, and defining valid operations applies equally well to categorical Types.

Consider again the city Type example, but this time as part of a multilevel geography Type with an added country unit.

Type Name: Geography
```
Units:  CHAR [16] AS Country, City
Values, Geography.City:  Chicago, New York, Boston,
Paris, Nice
Values, Geography.Country:  USA, France
```

In the same way as we talked about the scaling factor between months and years we can talk about the scaling factor between cities and countries. The only difference is that the scaling factor is not constant across all the Geography units. For example, the scaling factor between cities in the USA and the country USA is 3-1 while the scaling factor

between cities in France and the country France is 2-1. Although the collection of cities is nominally ordered and the collection of countries is nominally ordered each city and each country is associated with a unit that provides inclusion rules, scaling factors and permissible operations in the same way as with the month-year and meter-kilometer examples. (simplify the previous sentence) Thus the concept of unit, which we are generalizing here, is not restricted to cardinally ordered Types but rather applies to all Types[1].

Since a Type denotes a series of unit-specific values and since there may be more than one unit associated with the values of a Type, it is necessary to associate a unit (or ancestor/descendent rank or level value) with each value of a type.

```
Geography. : Country.France , Country.USA , City.Boston,
City.NY , City.Northampton , City.Paris , City.Nice
```

3. Every potential value must be uniquely identifiable and exclusively ORed with every other

Given a type T1 with instances i_1, i_2, i_3, i_4. This means that all i's are different from each other (uniqueness) and that if any is true within the context of a proposition, all other i's are false, and no two i's can be true at the same time (exclusive OR).

To see why this is necessary let's examine the converse. Assume that two instances are the same, T1 = $i_1(1)$, i_2, i_3, $i_1(2)$. Now consider the statement that y = $i_1(1)$ is true. We can't say that the negation of $i_1(1)$ is false because there is another instance of i_1, namely $i_1(2)$ that would also be true. This violates the rules of logic.

What about exclusive OR? An example of a collection of non exclusively ORed instances might be 'red, blue, green, small, large'. Something could be red and large. So red and large, though unique, are not exclusively ORed. The notion of mutual exclusivity is tied to the notion that a type denotes a single measurement or evaluation method or strategy. Clearly we would need two different tools/procedures to measure color and size. For a single evaluation method we can say the output of evaluation strategy 'x' is i_1 or i_2 or i_3 or i_4 or…

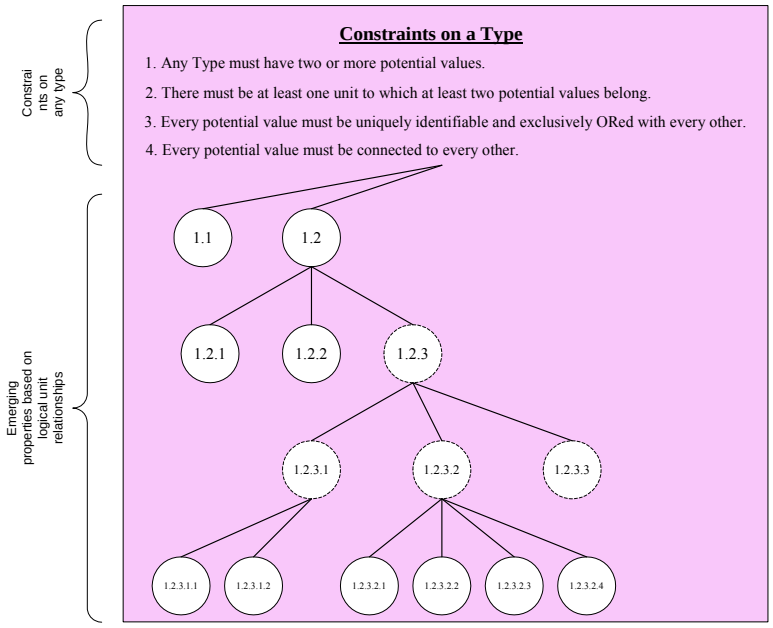
4. Every potential value must be fully connected

There must exist some valid function, which, when applied to any value of the Type can result in any other value of that Type.

TLH

The decision tree of emerging types

Treating the general constraints as the first node in a classification tree, we can describe the main kinds of types in terms of the following unit relationships.



- 1.1 If any of these constraints are not met by a Type, that Type is invalid
- 1.2 If all the constraints are met, and the Type has only one unit, the Type is a classic mono-level Type of the kind postulated in the Relational model and Date's more recent third manifesto.
- 1.2.1 If the Type has more than one unit and the units are not all connected, the Typeis invalid.
- 1.2.2 If the Type has only one unit and only two potential values, it is typically called a Boolean.
- 1.2.3 If the Type has more than one unit and the units are fully connected then there are a wide variety of valid Types that might be defined depending on the logical connectives thatjoin the units.
- 1.2.3.1 ANDed units
- 1.2.3.2 ORed units
- 1.2.3.3 AND/OR mixture
- 1.2.3.1.1 Specifically, if the units are all ANDed together, and there exists some functional relationship betweeen the units as there does, for example between distance, time an speed, the Type and whatever units(s) is(are) deemed dependent are compount.
- 1.2.3.1.2 If the units are ANDed together and there does not exist any functional relationship between the units as there does not, for example between a product name and SKU, the Type and the collection of units are considered to be concatenated.
- 1.2.3.2.1 If the units are all ORed together and they translation function between them defines a partial ordering on the set of units, the set of units forms a hierarchy.
- 1.2.3.2.2 If the units are all ORed together and the translation function between them defines a single unit relative to which all other units are defined, the set of units forms a hub and spoke unit set.
- 1.2.3.2.3 If the units are all ORed together and the translation function between them defines a set of network-style connections the set of units forms a peer-to-peer unit set.
- 1.2.3.2.4 If the units are all ORed together and the translation function between them defines a set of hierarchical units nested within a peer- to- peer unit set, the Type is considered to have a unit set. The unit relationships between the British system of pound weights and the Metric system of weights has this form.

KEY

Type definition examples

In contrast with root Types which need to be fully constructed, non-root Types can be defined with a Name and Unit expression (where Unit is a previously defined Type).

General syntax for non-root Type creation:

```
Create Type Name
With Unit TypeName [ AND|OR TypeName ...]
Unit Expression [UnitFormula]
```

The Type names referenced in the specification of Units may themselves be scoped or restricted (scoping syntax was presented in section 5.4 above). As described in section 4, when there are multiple OR'd Units, the Units must include a translation function. The simple examples presented immediately below do not use hierarchical notions as they haven't been introduced yet.

Examples of simple concrete Types

```
Create Type Product
With Units Categorical
```

Creates a type named Product with categorical units. Absent further specification, any valid categorical values are valid products.

```
Create Type Product
With units Master_list.*
```

Creates a type named Product whose unit (potential values) are drawn from the actual values of the type Master_List. The `.*` construct specifies all instances.

```
Create Type Sales
With Units Dollars
```

Creates a type named Sales whose units are drawn from the potential values of the type named Dollars.

If the units are specified in terms of the name of a Type, as with the Type "Product" defined with "Categorical" units above, or Sales defined with Dollars, then the potential values of that named Type are used. If the intent is to define the potential values or units of a Type in terms of the actual values of another Type as when the Type "Product" is assigned units defined in terms of the actual values of a Type called "Master List", then some scoping must be used.

Examples for Units of a Compound Type

```
Create Type Unique_Product
With Units Product AND SKU
```

Examples for units of a Mixed Type

```
Create Type XML Text
With units <text , Format>
Unit expression <Text.* OR Format.*>
Unit expression <Text[0]-[0]Format>
```

Where the last [0]-[0] relationship indicates explcitly that text and format do not form a heirarchy.

Some remarks on Type Structures

Though all models, schemas, cubes and applications in the Relational, Multidimensional, BI and/or BPM sense of the terms are examples of Type Structures, there are plenty of useful Type Structures one might define that would not ordinarily go by one of these names. Most common are what might be called model-fragments and/or analytical widgets. Things like complex pricing functions that can be incorporated into numerous models (and which make sense to develop once and use repeatedly), or sophisticated Time management functions, or SIC-based product attribute sub-models, or currency conversion routines can be created and saved as Type Structures, and incorporated into more complete, model-like, Type Structures on an as-needed basis[viii]. One of the aims of the LC model is to facilitate the reuse of model components and provide for the rapid assembly of models based on previously defined components.

Within the LC Model, Type Structures are intended to be as free form as possible. If the Types are well formed, then any constructable ordering relationship between two or more types will produce a well formed Type Structure. This is akin to a language where any word combination produces a meaningful sentence. (It is a desirable property for semi-automated reasoning systems that need to be able to generate their own Type Structures as hypotheses.) Whether the sentence, or Type structure, is useful is determined by how well it fulfills some need.

Type Structures and Types share the same name space. And the distinction between them is relative, not absolute[13]. In other words, if there were two persons modeling the same situation, one might define a complex Type to represent a certain aspect of the situation where the other person might define a Type Structure. For example, one person might model a "Geography" hierarchy as a Type Structure called "Geography" defined in

[13] A single word in a technical discipline may translate into many sentences in lay-speak.

terms of a 1-N ordering relationship between the already existent Types “Country”, “State” and “City”. The other person might model the same thing in terms of a single hierarchical Type called “Geography” composed of the units “Country”, “State” and “City” connected through a 1-N ordering relationship. It should not matter for any downstream application whether Geography was defined as a Type or a Type Structure.

The only practical implication of the difference is if Geography is defined as a Type and someone queries for the possible values for geography, those values are defined by the Type. By contrast, if geography is defined as a Type Structure and someone queries for the possible values for Geography, those values are defined by the Types of which the Types Structure is composed and not by the Type Structure itself.

To the greatest degree possible, Types and Type Structures are treated equivalently. The following, for example, are the same

- the way in which Types and Type Structures are named,
- the way in which their values are referenced and scoped,
- their ability to participate in or contain ordering relationships,
- their ability to participate in Type Structures

This is why in the UML description of the LC Model’s abstract syntax, the abstract class “Typebase” is introduced to capture all the attributes shared by both Types and Type Structures.

The near equivalence of Types and Type Structures (keeping in mind that there are a small number of significant differences which will be explained in section 4.2.4 below), is a desirable property of the LC Model. One major benefit of the shared namespace for Types and Type Structures is in allowing different groups to look at the same phenomena in different ways[ix], while retaining the ability to define equivalency relationships between their respective views.

Informal definitions of emergent types

Expressions are created from ordering relationships and operators between construct relationships and not constructs in isolation. Thus, for example, an expression might be stated in terms of a relationship between Units in a Type or a relationship between values in a Unit or a relationship between Types in a Type Structure or a relationship between values in one Type and values in another Type. An expression would never be stated simply in terms of a Type.

Examples of relationships between primitive Constructs include

- Value per Unit
- Unit per Type
- Value per Type
- Type per Type Structure

- Type Structure per Type Structure

From these, we can specify the following derived constructs.

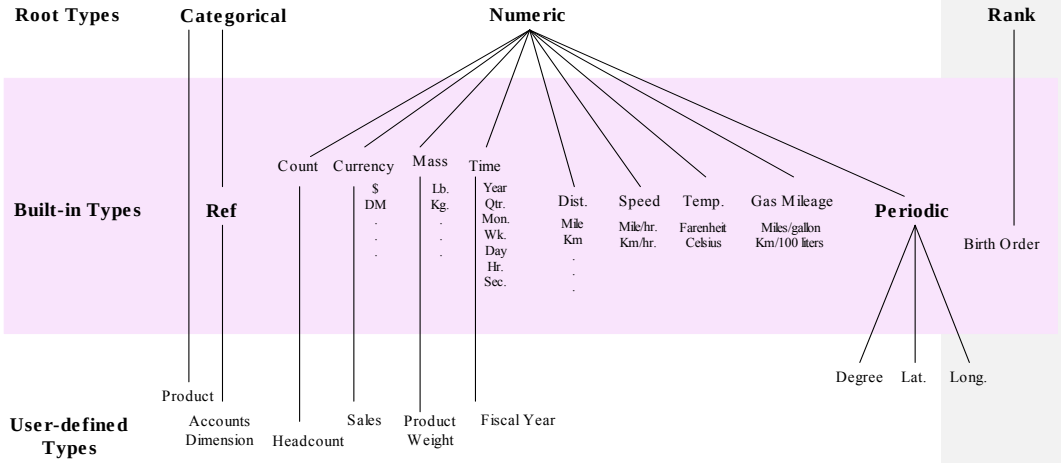
The Derived Constructs	The Ordering relationship between primitive constructs that defines the derived constructs
Numeric data types	Positional Relationships between different values qua iterations of the same constant unit of some Type wherein each value has one adjacent value in each of two directions
Rank data types	Positional Relationships between different values qua iterations of a variably sized unit of some Type wherein each value has one adjacent value in each of two directions
Categorical data types	Positional Relationships between different values qua patterns within the same unit and Type wherein each value can change to any other value in one step
Parent-child relations	Resolutional Relationships between different values (in different implicit units) of the same Type wherein each value is adjacent to at most one value in one direction and adjacent to possibly many values in the other direction
Leveled hierarchies	Resolutional Relationships between different units for the same relative value (regardless of whether the units belong to the same Type) wherein each value is adjacent to at most one value in one direction and adjacent to possibly many values in the other direction
Processes	Positional Relationships between different values of the same unit wherein each value is adjacent to possibly many values in each of two directions
The tuples defined by the cartesian product of a set of Types used to define a location structure or a simple equation within a single Schema	Positional Relationships between different values from different units of different Types in the same Type Structure wherein each value is adjacent to "N minus 1" values where N is the number of Types in the Type Structure
A simple equation that spans two schemas such as one equating values in a Bailey model with values taken from an external data source	Relationships between different values from different units of different Types in different Type Structures wherein each value is adjacent to N values where N is the number of inputs if the value is an output, or N is the number of outputs if the value is an input.

One of the powerful benefits of being able to specify these otherwise taken-as-primitive constructs is the ability to create new derived constructs when new situations arise and to create an appropriate set of new atomic functions to support those constructs.

Another powerful benefit of being able to specify these otherwise taken-as-primitive constructs is the ability to fine tune them when required by applications. Thus, for example, (and as will be described in detail section 6), the potential values of any determinate sized Type will have an edge. The question is therefore how to handle requests/functions that would go beyond that edge. So if the construct is a hierarchical Type and the request is to retrieve a value from the parent of a root node, what happens? Is the function not applicable? Does it apply but return no parent? Is a new root value automatically created and referenced? The LC Model's ability to define atomic functions in terms of ordering relationships enables you to specify exactly what functions should apply in which situation thus providing a precise and tunable semantics. Some customers may want edge functions to behave one way other customers may legitimately want those same functions to behave differently.

Illustration of root and non-root types

The following diagram shows some illustrative root and non-root types:



Novel attributes of the LC System

Processes can be modeled as Types or Type Structures. The process steps are captured as a location structure to which arbitrary measures may apply as contents. The notion of ordinal Time is built in to the process Type. Calculations of relative process concurrency

are directly supported. Processes can spawn sub-processes and can be rolled up into higher level process abstractions. In short, sophisticated process modeling is directly supported in the LC Model. Significant examples can be found in the Workflow examples of section 6.

Since Type Structures can easily embed in other Type Structures, the LC Model directly supports nested and dynamic schemas. Interesting examples are found later in the Food Cakes application

Since the definition of a Type or Type Structure can vary between uses and since equivalency relationships can be established between any Types that share at least one primitive Type, the LC Model directly supports the ability for multiple users to collaborate by dynamically linking their personal views with a common underlying model or by linking their models.

The combination of flexible representation, general equivalency mappings and nested ordering relationships supports very powerful and general semantic mapping.

The combination of ordering relationships and representations supports representational intelligence. See the appendix on "the Application of Type language to the Semantic Description of visual forms".

General Type Families

Introduction

The purpose of this chapter is to

1. Show the LC System type theory being used to systematically describe in a fully consistent fashion, some generally used families of types (AKA foundational type families) including ones used in abstract science and industrial strength software database applications. Particular attention will be paid to areas where classical typing approaches are known to be problematical. This includes formal definitions for varying kinds of hierarchies as found in database applications and formal definitions for so-called pure number systems such as the Rationals and the Reals.
2. Compare the LC System with other typing systems including those found in Book-of-Changes, Aristotle, Leibniz, Kant, Russell, Ramsey, Church, Montegue, Martin Von Los, Assembler, C++ , Prolog, Hilog, SQL , OLAP

In mathematics and traditional database applications, there are usually only a small number of different base types in the sense of quantities that are referenced by named

variables. They typically include some notion of a Categorical type of which Boolean is a special case, a Rank type, a numeric type (typically Rational or Real in mathematics, Integers and Floating points in software) and a Periodic type, most often some measure of degrees of rotation varying from 0 to 360.

In addition to these traditional base number-like types there are a wide variety of other structures that get used and defined in quantitative applications including graph structures, network models, hierarchies and multi-unit complex structures.

In this chapter, it will be shown how to look at the whole variety of so-called numbers and structures as different families of general types differentiated in terms of attributes described below which stem from the general components of a type described earlier in chapter three.

Two important distinctions to keep in mind are whether the potential values of the type are formulaically or explicitly defined and whether the type is defined in advance and then used as the units for a named variable or whether the type is defined at use time. Typically speaking, types whose values are formulaically defined such as the integers and the Rationals are pre-defined, so-to-speak and then called upon to serve as units for subsequently defined types. In software the pre-defined numeric types might also include short and long integers, and single and double precision floating point numbers. The specific potential values that constitute formulaically defined types such as the Rationals are constant across all uses of the type.

In contrast, types whose potential values are explicitly stated such as a Categorical-based hierarchy or a directed acyclic graph are most often constructed or defined at use time. This is because the specific set of potential values for such a type will vary across applications. The specific nodal values for one company's network model will be different from that of another. What is constant across applications is the set of atomic and molecular operators that the type supports such as drill down for a hierarchical type or next or previous step for a graph structure. Thus, these operators are typically predefined. The act of using a type whose potential values are explicitly stated typically involves

- selecting the kind of type such as network or hierarchical
    - which selection in turn fixes the atomic and molecular operators and then
- explicitly specifying the potential values for the type.

Regardless of how the potential values are defined for a type, once defined, the type needs to be usable as either locator or content in any expression.

Finally, by studying the type families presented in this chapter, the reader will gain an understanding of the attributes according to which they vary. No longer will it be that networks are just different from the integers or that acyclic directed graphs are just different from ordinals. Rather the careful reader will appreciate the specific ways in

which they are different, how any one type could morph into any other, and how as the types morph so too must morph their atomic operators.

From the standpoint of software systems, the reader will learn how to conceive of type families as type templates that get filled in and named to create specific usable types.

Defining attributes for type families:

The general components of a type and the emerging complexities they support provide the basis for the variety of attributes[14] relative to which basic families of types may be differentiated. These attributes, stated here informally, include:

1. The arity of the type:
   The arity of a type can vary from binary to N-ary.

2. The sameness or constancy of the count of neighbors.
   Most classic numeric types such as Wholes, Integers and Decimals have a constant number of neighbors per value per unit, typically two. In contrast, with hierarchies, graph structures and networks, the number of neighbors per value may change from value/node to value/node.

3. The number of adjacencies or neighbors per value per unit in the type:
   Most lattice structures have four or more neighbors per value, but again that number is constant throughout the type. Most classic number systems have two neighbors per value per unit-one in each of two sequencing directions as for example, with whole numbers and integers. In contrast, directed graph and network structures have numerous neighbors per value per unit.

   3.1 For those types whose values within some unit are sequenceable (that is each value has one neighbor in each of two inversely related directions):

      3.1.1 Whether the intervals between the values, or unit differences or adjacencies may or may not be defined-as-constant. When they are defined-as-constant, they are said to define a cardinal or numeric sequence. When they are not defined-as-constant, they are said to define a rank or ordinal sequence.

      3.1.2 Assuming there is a minimum and maximum value (or limit), whether they are the same, in which case the type is typically called periodic[15]. Or whether the minimum and maximum values (or limits) are different, in which case the

[14] The attributes are not intended to be fully independent. Clearly the ordering aspects of a type are inter-related. However, the attributes as large scale features are useful for descriptive purposes.
[15] if both permitted then must be one dimensional

type is typically called open. Degrees of rotation are common example of a periodic type. 360 degrees is the same as 0 degrees. The integers are common examples of an open type.

4. The number of ANDed units in the type.

Most types have only a single unit and as such may be thought of as simple. However, there are many cases when there are multiple ANDed units in a type. The complex number system is an example of a compound type with two ANDed units, namely one real and one imaginary. Many statistical algorithms return compounds like "mean and standard deviation" which may be thought of as two ANDed units in a compound type.

5. The number of XORed units in the type:

Flat types such as "whole numbers", "integers", "lists" , and "Booleans" are examples of flat or uni-level types that have one unit. All hierarchies, whether leveled or ragged have two or more XORed units. A type may have two or more XORed units without a hierarchy between them.

6. Whether an atomic operator that iterates over all the values in a type via a simple command such as "Next", permits the repetition of, or repeated visits to, certain values:

The repetition of values is typically called looping and types that permit it are typically called cyclic. Looping is most often found in graph and network structures.

Clearly, these attributes do not lend themselves to a simple method of classification. Rather, there are several different dimensions relative to which types may be differentiated. No significance should be attached to the particular arrangement in this chapter. The objective was to proceed from more common to less common, and from simpler to more complex.

Relevant concrete syntax

In order to describe how the various type families are created and further discuss and compare their properties, it is necessary to introduce some concrete syntax. The syntax provided below is very close to what is being used in the LC system implementation. Since all the syntactic elements follow directly from the concepts described in chapter three, above, they are presented with minimal explanation.

Typographical Conventions

In the examples below, normal text is to taken literally. Italicized text indicates that the user is expected to substitute some other appropriate symbol. For example "*T.v* " means that the reader should substitute the name of some type (or an expression yielding a type object) for *T* and the name of a value (or an expression yielding a value) for *v*.

Integers are used as postfixes to constructs to distinguish multiple occurances of the same kind of construct within a single expression. The conventions used are described in the following table.

Rule	Symbolic example	Concrete example
If every occurance of the construct is by definition the same, the occurances of the construct are not numbered.	T.u.i = = T.u.i	Product.dept.shoes = = Product.dept.shoes
If two or more occurances of the same construct may be different the first one expressed is assigned the integer 1 and the second one the integer 2	{T1.u1.i1 , T2.u2.i2}	{Product.dept.shoes, Time.month.january}
If two or more occurances exist for the same construct and some are by definition the same while others might be different, each new distinct occurance of the construct is assigned the next new integer. And each new non-distinct occurance is assigned that integer which was first assigned to that particular occurance of the construct.	{T1.u1.i1 , T2.u2.i2 , T1.u3.i3 , T3.u2.i2}	{Time.month.January , Product.dept.shoes , Time.day.Monday , Employee.dept.shoes}

Note also that whitespace in expressions can be significant. For example, *T.v1! = T.v2* is not the same as *T.v1 != T.v2* since the unary bang (!) is an operator in its own right. We would like to provide a less sensitive concrete syntax for Bailey

Primitive Construct subexpressions

Consider the following token representations for primitive constructs:

Concept	Token
Any Construct	K
Type Structure	TS
Type	T
Unit	U
Value	v
Instance	i
Function	F
Group (subset)	G
Ordering relationship	[ ]—[ ]

TLH

Primitive ordering subexpressions

Token	Meaning
K	Any Construct
K[*int range*]	The number of instances of "K" that exist in a single instance of the ordering relationship. Int range may be a single int. If int is zero on both sides it signifies that the two sides are unrelated
K[*int range*]+	The plus sign "+" on the outside of the square bracket signifies that the values from "K" are drawn with replacement and thus could repeat between each instance of the ordering relation
-p-, -r-, -a--?-	Adjacency type: positional, resolutional, any, or unknown.
K[*int range*+]	The plus sign "+" on the inside of the square bracket signifies that the values from *K* are drawn with replacement for each instance of the relationship
( )	Reifies the ordering relationship. It creates an unnamed instance of a Type Structure

Primitive operator subexpressions

The LC System supports a wider variety of types than found in any database system, Relational, multidimensional or other, but does so by linking the exposure of the specific referencing and navigational capabilities of the type to the existence of the specific ordering relationships from which they follow. Thus, for example, if one were to define a Type for which no hierarchical ordering relationships had been established, hierarchical referencing within that Type would be disallowed.

What follows are examples of primitive functions that are applicable to any Type. Non-primitive functions are described later in this document.

Sub expression	Meaning	Example
T.v	The value of some Type is literally expressed	Product.Brand.Pepsico where "Brand.Pepsico" is the value
T.G	A named group or subset of some Type is expressed	"Product.New_list" where "New_list" is a named group.
T.v	The value of some Type is expressed as a variable	Product.v
T.U.i	The instance of some unit of some Type is literally expressed	Product.Brand.Pepsico where "Pepsico" is the instance
T.U.i	The instance of some unit of some Type is expressed as a variable	Product.Brand.i
T.U.(i)!	An instance of some unit of some Type is negated	Sales.Dollars.(500)!
T.(v)!	A value of some Type is negated	Sales. (Dollars.500)!

T.U.(i XOR *i..)*	Two or more instances of some unit of some Type are exclusively Ored	Geog.Country.(USA XOR Canada)
T.U.(i AND *i..)*	Two or more instances of some unit of some Type are ANDed	Geog.Country.(USA AND Canada)
T.U.{i ,i , i..}	Two or more instances of some unit of some Type are enumerated	Geog.Country.{USA , Canada}[16]
T.*	Every value of some Type	" Product.*" refers to every value of Product in the current context. If the current context is a defined Type Structure, it refers to every value of the Type actually used in the Type Structure. If the current context is a Type definition, it refers to every potentially usable value from the Type definition.
T.u.*	Every instance of some unit of some Type	"Product.Brand.*" selects every instance of the Brand Unit of the Product Type in the current context. If the current context is a defined Type Structure, it selects every Brand instance actually used in the Type Structure. If the current context is a Type definition, it selects every potentially usable Brand instance from the Type definition.
EXISTS(*T1*.v1 , *T2.v2*)	At least one value of T1 exists, as in the existential quantifier "∃", relative to the location defined by T2v2 . Alternatively, it could read, at least one value of T1 meets the constraint specified by T2v2.	"EXISTS (Product.v , Sales_price > $100)" asserts at least one value of the Type Product has a sales price of over $100.)
ALL (T1.v1 , *T2.v2*)	Every value of T1 meets the constraint specified by T2v2. This is the universal quantifier "∀".	"ALL (Product.v , Sales_price > $100)" asserts every value of the Type Product has a sales price of over $100.)
T1, T2.u.i1 == T1, T2.u.i2	The value (or instance of some unit) for T1 relative to T2u.i1 is comparatively equal to the value for T1 relative to T2.u.i2	IF Sales, Product.Department.Shoes = = Sales, Product.Department.Toys THEN…
T1, T2.u.i1 != T1, T2.u.i2	The value (or instance of some unit) for T1 relative to T2.u.i1 is not equal to the value for T1 relative to T2.u.i2	IF Sales, Product.Department.Shoes *!=* Sales, Product.Department.Toys THEN
T1, T2.u.i1 = T1, T2.u.i2	The value (or instance of some unit) for T1 relative to T2u.i1 is assigned the value of T1 relative to T2.u.i2	Sales, Product.Department.Shoes = Sales, Product.Department.Toys;
T.this	The value of some Type within some expression context	Costs, Product.this = F(Wages, Product.this)
TS.{*v*}	The value of some Type Structure is expressed	MySalesModel.{Time.Month.January, Geog.City.Cambridge, Sales.Dollars.500}

In addition there are a number of standard molecular functions such as Union and Intersection that are supported for any Type.

[16] We are currently using {} to denote lists of constructs and ( ) to denote functions. Commas are common delimiters within lists. In the context of a schema-specific query or calculation expression the commas denote AND. In the context of editing a Type definition, as for example, inserting a list of instances into a specific unit of a Type, the commas denote XOR.

All Types also need a name, whether supplied by the user or system-defined.

Syntax

```
Create Type NAME
Delete Type NAME
```

Examples

```
Create Type Product
Create Type Sales
```

Strictly speaking, a Type has been created from the moment it has been named. Although it still needs to be assigned units before it can be used, the current consensus is that a created Type should be usable given only a name based on the assumption that the system assigns a default unit of "category".

Open and flat type families

Categorical

Flat as in uni-level categorical types include everything from color lists to names of persons. Boolean types are binary categorical. On average categorical types have only one unit, but, strictly speaking, there is no reason why one couldn't have a multi-unit categorical type.

With the exception of Booleans, the potential values of categorical types are typically explicitly specified.

The concept of neighbor is worth some discussion. Since there is no order to the potential values of a categorical type, any value could precede or follow any value in an enumeration of the potential values of a categorical type. This is why for a categorical type of N potential values, every value has N-1 neighbors. However, since there is no concept of distance for a categorical type, and thus no operator like "next", one could argue that the concept of neighbor doesn't even apply.

Attributes

Attribute	**Value**
Arity	Binary to N-ary
Sequenceability	No
Number of ANDed units	1 –N
Number of XORed units	1
Number of neighbors	N-1
Constant number	Yes
Permitted cycles	Not applicable

Ordering Expression

Although categorical types are typically physically represented in a software system in terms of a traversable series, the semantic definition is void of any inter-value ordering. Thus, any value of a categorical Type is as close to any complement as any other complement.

Create Root Type "Categorical"

With Value Ordering expression

```
Categorical.v.*[1]-p-[All]Categorical.v!
```

Read "Every value in some Categorical Type is positionally adjacent to all other values of that Type". Since there is only one unit for this root Type, there is no need to name it. (This is consistent with other structural artifacts such as hierarchies which only need to get named if there are more than one.). This definition specifies a family of root Types because the number of potential values is variable[x].

```
Create Type "Cities"
With Units "Categorical"
```

Atomic Categorical Operators

- Assertions and negations

Categorical.*v1* = Categorical.*v2*
Categorical.*v1* != Categorical.*v2*
Categorical.*v1* = Categorical.*v2*!

- Subsetting

Categorical.*G1.v1* = Categorical.*G2.v2*
Categorical.*G1.v1* != Categorical.*G2.v2*

- Value testing

If value, Categorical = Categorical.*v* then ….. else……
While value, Categorical = Categorical.*v* Do "X" else "Y"

If value, Categorical.*G* = Categorical.*G.v* then ….. else……
While value, Categorical.*G* = Categorical.*G.v* Do "X" else "Y"

- Editing

Add/delete potential value per Categorical

Rank

Rank, ordinal, or indexical types are very frequently used.

Attribute summary for Rank types:

Attribute	**Value**
Arity	Binary to N-ary
Sequenceability	Yes
Constant interval	No
Number of ANDed units	1
Number of XORed units	1
Number of neighbors	2
Constant number	Yes
Permitted cycles	No

Ordering Expression

1. (Rank.*v1*.(First to Last – 1)) [1] -p-[1] (Rank.*v2*.(First+1 to Last)) AND

Read: For all values that are between the first and last value in a Rank Type, each value is adjacent to one other value in each of two inverse directions. We choose to label those two directions "+" and "-". And we choose to associate movement from left to right as the "+" direction. While movement from right to left is designated the "-" direction.

2. Rank.V.Last [1] -p-[0]Rank.*v3* AND

Read: The one Rank value called "Last" has no neighbor in the "+" direction.

3. Rank.*v4*[0] -p-[1]Rank.V.First

Read: The one Rank value called "First" has no neighbor in the "-" direction.

Example

TLH

```
Create Type Finish
With Units Rank
```

Atomic Rank Operators

Typical uses of a Rank Type as unit assign to that Rank Type a certain number of potential values. For example, the number of runners in a road race.

For a given number of Rank values, queries of the form "Who finished ahead of Jane?" when Jane is the first placed runner or any other query that attempts to subtract from some rank a number greater than the value of the rank, by default returns a "not applicable" because there is no such rank. Likewise, any query of the form "Who finished after Phil?" when Phil finished last, by default returns a "not applicable".

If Rank.V1 +/- V2 = Rank.(Between First AND Last)

Then Rank.V1 +/- V2 = Rank.(V1 +/- V2)

Else Not Applicable

Of course, one can always change the number of rank values. And in so doing insert new values that are either before what had been First, between what had been First and last, or after what had been Last.

Within the context of a query execution there is always the notion of "this rank" or Rank.this.

Rank.(First to This)
Rank.(This to Last)

Rank.V > Rank.V – (1 to N)
Rank.V < Rank.V + (1 to N)

If V, Rank.V = Rank.V1 Then …. Else……

For Rank.V = 1 to N Do……
While Rank.V = V1

Molecular rank operators

Given a Type with at least rank ordering amongst its instances, standard functions like concatenate and insert can also be supported.[xi]

TLH

Whole Numeric

There are a variety of whole numeric types. The most common are what we call "Natural Numbers", "Whole Numbers", and "Integers". The Naturals begin with "one" and extend to N, the Wholes begin with zero and extend to N, and the Integers range from –N thru zero to +N. There are subtle differences in the atomic functions supported by the three different types.

Common attribute summary for Whole Numeric types

Attribute	**Value**
Arity	Binary to N-ary
Sequenceability	Yes
Constant intervals	Yes
Number of ANDed units	1
Number of XORed units	1
Number of neighbors	2 [17]
Constant number	Yes[18]
Permitted cycles	No

Varying attributes

Example of bounded decrement with unbounded increment supporting unbounded addition and difference but only bounded subtraction

Ordering Expression

Whole.v1.* 1-p-1 Whole.notfirst.v1!.*

Read:

1. For every whole value that value has one neighbor in the increment direction
2. No value occurs more than once in the whole number series
3. No value is the same as its neighbor
4. No value is indirectly adjacent to itself (no cycles)

[17] Applies to non end values. I.e., excludes zero for the Wholes and one for the Naturals
[18] Same restriction as above

Atomic Functions

Whole.(*v1* +/- *v2*) = Whole.*v1* +/- Whole.*v2*;
Whole.(*v1* * *v2*) = Whole.*v1* * Whole.*v2*;

Standard non-atomic numeric functions are described in later in this section ( ).

Example of unbounded decrement with unbounded increment supporting unbounded addition and subtraction

Ordering Expression

Integer.*v1*.* [1] -p-[1] Integer.*v2*.*

Read: Every Integer value is adjacent to one other value in each of two inverse directions. We choose to label those two directions "+" and "-". And we choose to associate movement from left to right as the "+" direction. While movement from right to left is designated the "-" direction.

This implies there are no upper or lower bounds. Such bounds could easily be specified in the same way as was described in the footnote for Categorical Types..

Atomic Functions

Integer.(*v1* +/- *v2*) = Integer.*v1* +/- Integer.*v2*;
Integer.(*v1* * *v2*) = Integer.*v1* * Integer.*v2*;

Standard non-atomic numeric functions are described in later in this section ( ).

Open and hierarchical type families

Numeric Resolutional Hierarchies

The Rationals

Though not often portrayed in this light, the Rational number system is a classic example of a numeric resolutional hierarchy. What this means is that from any value in the

Rationals, there are two different kinds of adjacent values: positionally adjacent values and resolutionally adjacent values.

For example, from the value 3 in the Rationals, one can move positionally to the numbers 2 and 4. Resolutionally, one can move down to the 1/2s, specifically 5/2, 6/2 and 7/2.

There are two main ways of describing the Rationals: as a continual fraction-based hierarchy, akin to the standard definition where a Rational is any fraction A/B where B is not equal to zero, or as a power-of-ten- or decimal-based hierarchy.

As a continual fraction-based hierarchy, the Rationals are constructed from a combination of the Integers as numerator and Naturals as denominator. The units in this case are the ones, halfs, thirds, fourths etc.. As a decimal system, the units are the ones, 1/10s, 1/100s etc.. The Rationals are gapless, regardless of the specific units chosen for them, and contrary to canonical traditions in mathematics[19]. (See the critique of canonical foundations of mathematics in chapter 'x' below.)

As a one dimensional system, the Rationals support sequential addition, subtraction, multiplication and division.

Categorical Resolutional Hierarchies

Introduction

There are numerous kinds of categorical resolutional hierarchies. (Hereafter simply referred to as hierarchies. Numeric resolutional and positional hierarchies will be explicitly so called.) And there are numerous ways to classify them. Chapter 5 in OS2E identified twelve basic kinds of strict single hierarchies[xii]. These included various forms of ragged, leveled and mixed hierarchies. That chapter also described how those differences only became possible for hierarchies composed of more than one root, two leaves and at least two generations/levels between leaves and roots. (That is to say that the Type "Hierarchy Specialization" when used as a content against hierarchies treated as locations is only meaningful for hierarchies beyond a certain complexity.)

Of course, not all hierarchies are strict single hierarchies. In addition, there are also non-strict hierarchies where a single child may connect to more than one parent. And there are so-called multi-hierarchies where a single Type or Type Structure contains more than one hierarchy.

In terms of distinct collections of atomic operators, the main differences appear between ragged and leveled hierarchies and between strict and non-strict hierarchies. Furthermore,

[19] The Reals are implicitly two dimensional. The so-called irrational numbers like the square root of two or pi require the solving of a simultaneous equation between two instances of the Rationals each with an incommensurate metric.

the distinction between strict and non-strict is orthogonal to that of ragged versus leveled. So non-strict hierarchies will be treated in their own subsection.

Finally, in real-world, industrial strength applications attention must be paid as to whether a data source that is purported to define a particular kind of categorical hierarchy does indeed define such a hierarchy. In other words, some kind of validation routine is typically run to ensure that a purported hierarchy is a valid hierarchy. As such, we include relevant validation routines below.

Common attributes for Categorical Resolutional Hierarchies

Attribute	**Value**
Arity	N-ary
Sequenceability within a unit	No
Sequenceability between units	Yes
Number of ANDed units	1
Number of XORed units	N
Number of positional neighbors	N-1
Number of resolutional neighbors	1 in the up direction N in the down direction
Constant number of resolutional neighbors in the down direction	No
Permitted cycles	No

Strict Ragged Hierarchies

Ordering Expression

Against an existent purported-to-be hierarchical LC type

T.Leaf.above.*v1*.*[1] -R- [N]*T*.Root.below.*v1*!.* AND

T.Leaf.*v2*.*[1] -R- [0]*T.v3* AND

T.v4[0] -R- [1]T.Root.*v5**

Read:

TLH

1 For each non-leaf value from T called v1 associate AS down(1) 1-N non-root values from T drawn from the complement of v1.
1.1 No value may appear on both sides of the ordering relationship
1.2 No left hand side non-leaf value from T may appear in more than one elemental ordering relationship
1.3 No right hand side non-root value may appear more than once in the down(1) position
2 For each leaf value of T called v2, zero values may be associated As down(1) values
2.1 No leaf value may appear more than once on the right hand side of an ordering either within an elemental instance or between instances
2.2 No leaf value may appear on the left hand side of an ordering relationship
3 For each root value of T called V4, zero values may be associated as up(1) values
3.1 No root value may appear more than once on the left hand side of the ordering relationship
3.2 No root value may appear on the right hand side of an ordering relationship

The combination of conditions above, effectively prohibit cycles. For example, a direct cycle will result in the violation of condition 1.1. An indirect cycle, such as A-C-D-A, if the offending value, A, is a non-root and non-leaf will violate condition 1.3 because, by definition, if A is non-root and non-leaf, there will exist another ordering relationship instance where A is the down(1) value such as Q-A

Atomic Operators

Ragged Hierarchy functions	Meaning
T.v.down(N) For N = 1, the expression is equivalent to *T.v*.children	The domain of values down N from the value named *V* of Type *T*.
T.v.down(M-N)	The domain of values down M through N from the value named *v* of Type *T*.
T.v.Up(N) For N = 1, the expression is equivalent to *T.v*.parent	The actual value N up from the value *v* of Type *T*
T.v.Up(M-N)	The actual values M-N up from the value *v* of Type *T*
T.root	The domain of root values of *T*
T.v.root	The root value from *T.V*
T.v.root.*	Implies multiple hierarchies or non-strict hierarchy. Maybe not support
T.leaf	The domain of leaf values of *T*
T.v.leaf	The domain of leaf values under *v*

Molecular Operators

These operators can all be constructed in terms of the atomic strict ragged hierarchy operators plus the operators applicable to any type.

Ragged Hierarchy Functions	Meaning
T.v.below	The domain of values below the value named *v* of Type *T*. Starts with children and ends with leaves.
T.v.@.below	All values down to the bottom from the value named *v* of Type *T*. Starts with self and ends with leaves.
T.v.above	The domain of values above the value named *v* of Type *T*. Starts with parent and ends with root.
T.v.@.above	All values up to the top from the value named *v* of Type *T*. Starts with self and ends with root.
T.v.sibling(N)	The domain of descendants N down from the ancestor N up from the value *v* of the Type *T*
T.v.sibling(M to N)	The domain of descendants M to N down from the ancestors M to N up from the value *v* of the Type *T*

Common Hierarchy Operator Expressions

Expression	Meaning
Add/Delete *T.v1* AS Down(1) , *T.v2*;	Add/delete child per parent
ADD *T.v1*.Down(1).*v2*.@.below = *T.v3*.Down(1).*v2*.@Below; DELETE *T.v3*.Down(1).*v2*.@Below;	Move child and all descendants from its parent to some other parent within the same Type
ADD *T1.v1*.Down(1).*v2*.@.below = *T2.v3*.Down(1).*v2*.@Below; DELETE *T2.v3*.Down(1).*v2*.@Below;	Move child and all descendants from its parent to some other parent in some other Type
DELETE *T.v*.above.@*	Delete a parent and its associated children

Validation routines

It is one thing to specify an ordering relationship that needs to hold for the values of a type; it's another thing to validate that the values of a type adhere to the specified ordering.

There are three main places where it makes sense to validate a type-defining ordering relationship: at the source, while the source is being mapped to an internal LC form, at the internal LC form.

For the examples in this section, we assume that the data to populate an LC hierarchical type comes from a SQL source.

Against a SQL parent-child table

Consider the following excerpt from a parent-child table that allows nulls in the parent column but not in the child column (leaf values never appear in the parent column)

Col1 AS Row	Col2 AS Child	Col3 AS Parent
1	Cambridge	MA
2	Arlington	MA
3	Boston	MA
4	Stamford	CT
5	Wilton	CT
6	MA	USA
7	CT	USA
8	USA	NULL

Treating each of the three columns as types, where T1 = Col1, T2 = Col2 and T3 = Col3 and defining a type T4 whose values are the union of the distinct non-null values from T2 and T3, and T2 and T3 are treated as named groups within T4 so that we can refer to T4.T2 and T4.T3, the schema may be defined as follows:

Note that T2 and T3 could have also been made comparable by defining T4 as a type used as the units for T2 and T3.

T1.* 1-1 T4.T2.*
AND
T1.* 1-1+ T4.T3.*

> **Read**: for each unique value of T1 associate one unique value of T4 drawn from the named group T2. Neither T1 or T2 may repeat. And for each unique value of T1 associate one possibly repeating value of T4 drawn from the named group T3 which may include the special value NULL.

For all values of T1, T4.T2.V != T4.T3.V

> **Read**: No value can appear as both a child value and a parent value in the same row

For any two values of T1, T4.T2.V, T1.V1 != T4.T2.V , T1.V1!

TLH

Read: No child value can appear more than once in the table

While reading the SQL table and creating an LC hierarchical type

Strict Named Leveled Hierarchies

In contrast with ragged hierarchies which are defined in terms of resolutional relationships between values, leveled hierarchies are defined in terms of resolutional relationships between units. (It's why they're called leveled.)

When a collection of XORed Units is identified as a hierarchy (regardless of whether it is named - and the identification can be automatic), the Units are called Named Levels.

Ordering Expression

T.Unit.Bottom.(Up).i*[1]-R-[N]T.Unit!.Top.(Down).i!* AND

T.Unit.Bottom.i.*[1]-R-[0]T.Unit AND

T.Unit[0]-R-[1]T.Unit.Top.i.* AND

Read:

1 For every instance i of a non-bottom unit of some type T associate AS down(1) 1-N instances drawn from the complement of i in T
1.1 No unit or instance may appear on both sides of the ordering relationship
1.2 No instance may appear on the left hand side in more than one elemental ordering relationship
1.3 No instance may appear on the right hand side more than once in the down(1) position
2 For the bottom unit of T, zero units may be associated As down(1)
3 For each bottom unit instance of T, zero instances may be associated As down(1) instances
3.1 No bottom unit instance may appear more than once on the right hand side of an ordering either within an elemental instance or between instances
3.2 No bottom unit instance may appear on the left hand side of an ordering relationship
4 For each top unit instance of T, zero instances may be associated as up(1) values
4.1 No top unit instance may appear more than once on the left hand side of the ordering relationship

4.2 No top unit instance may appear on the right hand side of an ordering relationship

The combination of conditions above, effectively prohibit cycles. For example, a direct cycle will result in the violation of condition 1.1. An indirect cycle, such as A-C-D-A, if the offending value, A, is a non-root and non-leaf will violate condition 1.3 because, by definition, if A is non-root and non-leaf, there will exist another ordering relationship instance where A is the down(1) value such as Q-A

Atomic Leveled Hierarchy Operators

Named level Functions	Meaning
T.L	The Level "*L*" of some Type "*T*".
*T.L.**	Every value in the level *L*
*T.v.*Level	The Level "L" of the Value "*v*" of the Type "*T*"
*T.v.*Level(Down M [to N]).*	The domain of all values M to N levels down from the level of *v* that are descendants of *v*
*T.v.*Level (Down M [to N]).*	The domain of all values M to N levels down from the level of *v*
*T.v.*Level(Down M [to N])	The name(s) of the Level(s) M to N levels down from the level of *v*
T.L(Down M [to N])	The name of the level(s) M to N levels down from the level *L*
*T.v.*Level(Up M [to N]).*	The ancestor value(s) M to N levels up from the Level of the value *v*
T.L.(Up M [to N])	The name(s) of the level(s) M-N levels up from the level *L*

TLH

Molecular Leveled Hierarchy Functions

Named level Functions	Meaning
*T.V.*Level.Above/Below.*	The domain of values above or below the level of the value V
*T.V.*Level.@Above/Below.*	The domain of values above or below the level of the value V including the value V
*T.V.*Level.Above/Below	The names of the levels above or below the level of the value V.
*T.V.*Level.@Above/Below	The names of the levels above or below the level of some value V including the level of V
*T.L.*Above/Below	The names of the levels above or below some level L
T.L.@Above/Below	The names of the levels above or below some level L including L
*T.V.L.*V.*	All values of the level L (whether above or below the level of the given Value *V*) that are above or below the given value *V*.

Common Named Level Expressions

Expression (glad to provide on request)	Meaning
	Add/Delete a level to/from a Type
	Define a level of a Type in terms of another Type
	Define a level of a Type in terms of a level of another Type
	Define a collection of Types that share a common set of named levels and values
	Define the values and parent-child connections between the values for two adjacent Types in terms of another Type or Type Structure
	Modify the parent-child connections between the values of two adjacent levels

Validation from a SQL table

A typical SQL configuration for a leveled hierarchy is as follows.

TLH

Store	City	State	Region	Country
Pete's	Cambridge	MA	NE	USA
Foodworld	Cambridge	MA	NE	USA
Petland	Alston	MA	NE	USA
Music mania	Alston	MA	NE	USA
Sound streams	Santa Cruz	CA	West	USA

The column headers correspond to Unit or level names. The fields correspond to named level- or unit-specific instances. Nulls would not normally be allowed in this table configuration.

The columns can be arranged in rank order from top to bottom as in

1-R- N (Country.*, Region.*, State.*, City.*, Store.*)

Read: There exists a 1-R-N relationship between the instances of any unit and the instances of any lower unit for the units Country, Region, State, City and Store listed in descending order.

The count of the instances of the bottom unit would need to equal the count of the rows

Unit.bottom.i.* 1-1 Table.row.*

For every instance there exists one associated instance for every upper unit.

Unit.i.* 1-R-1 Unit.higher.i.*

Hierarchies With Multiple Parents Per Child

Hierarchies with multiple parents per child (for example when a city belongs to two Counties or States, or when a week belongs to two months or years), are most efficiently described as single hierarchies with some irregularity (so long as the average number of parents per child is reasonably close to one).

Ordering Expression for non-strict hierarchies
From an ordering relationship perspective, the same irregularity is occurring regardless of whether it is in a ragged or leveled hierarchy. Specifically, the [1]-R-[N] relationship no longer requires that the values that feed the [N] part of the relationship do not repeat between instances of the relationship. Thus, the "+" token is added to the outside of the [N] as "[N]+" to indicate the values are drawn **with** replacement.

For the ragged case the ordering expression looks as follows:

T.Leaf.above.V.*[1] –R- [N]+ T.Root.below.V.*!

For the leveled case the ordering expression looks as follows.

Unit.Bottom.(Up).*[1]-R-[N]+ Unit.Top.(Down).*

The edge cases are not affected. Thus they were not repeated here.

Although the above ordering expressions enable values to have more than one parent , they are general statements; (not because of the ordering relationships but because of the Constructs). They do not specify which values have more than one parent, nor what those parents are. Consider the following specific statement made against a hypothetical hierarchy that had States above Cities. It would be rejected as illegal if the hierarchy were defined as strict (no repeating values). But with the above non-strict definition, it is legal.

```
(  [1]~ Geography.State,
   [N]+ Geography.City
),
( Geography.State.{ Missouri, Kansas },
  Geography.City.KansasCity
)
```

It says that the City KansasCity has two parents in the State level: Missouri and Kansas. This is all we need to know from an ordering perspective. Issues of how to aggregate data from Cities to States without double counting are handled in terms of aggregation and allocation functions.

For example, if Sales data from KansasCity needs to be evenly apportioned between its two parents, the following calculation expression would be made.

Sales, Geography.State.(! {Kansas, Missouri}).* = SUM (Sales, City);
Sales, Geography.State.{ Kansas , Missouri } = SUM ((Sales, City.(! KansasCity).*) + ½ (Sales, City.KansasCity))

The apportioning rules for different data need not be the same as for Sales. Should it be the case that all data is apportioned the same way, one would simply define the apportioning logic once for all Contents at that Location.

TLH

Atomic Functions

The only new atomic function required is the specification of which parent to move towards when moving up the hierarchy.

Function	Meaning
T.V(Up/ N , *parent*)	From some value of some Type move Up N in the direction of some parent

Additional Functions

Expression	Meaning
glad to provide	When viewing a value that has multiple parents along with the siblings of that value under one of its parents show the full values of its contents
glad to provide	When viewing a value that has multiple parents along with the siblings of that value under one of its parents show the parent-apportioned values of its contents

Multi-hierarchies

The essential difference between single hierarchies and multi-hierarchies is the same regardless of the kind of hierarchy. Specifically, there is one ordering relationship per named hierarchy. Named hierarchies are symbolically designated as "H" or "*H*" using the same literal versus variable distinction as used for other Constructs.

As such, the only language addition is the inclusion of Hierarchy name wherever the result of some hierarchy function varies with respect to specific hierarchies. The hierarchy name is inserted immediately following the Type name if it is being asserted. Or it can be queried in the same manner as named levels.

Multi-hierarchy Functions

There are two main concrete ways navigation can be customized as a function of the hierarchy relative to which that navigation takes place.

TLH

- Scope or restrict the Type to the specific hierarchy
- Qualify the endpoint or direction in terms of the hierarchy

Bailey may support either or both concrete syntaxes.

Scoping or Restricting the Type	
Hierarchy-specific Ragged Hierarchy functions that scope the Type	**Meaning**
*T.H.v.*down(N) For N = 1, the expression is equivalent to *T.v.*children	The domain of values down N from the value named *v* of hierarchy *H* in Type *T*.
*T.H.v.*down(M-N)	The domain of values down M through N from the value named *v* of hierarchy *H* in Type *T*.
*T.H.v.*Up(N) For N = 1, the expression is equivalent to *T.v.*parent	The actual value N up from the value *v* of hierarchy *H* of Type *T*
*T.H.v.*Up(M-N)	The actual values M-N up from the value *v* of Hierarchy *H* of Type *T*
*T.H.*root	The domain of root values of Hierarchy *H* of Type *T*
*T.H.v.*root	The root value from *T.V* in hierarchy *H*
*T.H.*leaf	The domain of leaf values of hierarchy *H* of *T*
*T.H.v.*leaf	The domain of leaf values under *v* in the hierarchy *H* of *T*

To minimize redundancy of exposition, examples of adding a hierarchy qualifier to the function rather than to the Type are shown for leveled hierarchy functions.

Qualifying the navigation	
Named level Functions	**Meaning**
*T.H.L.**	Every value in the level *L* in hierarchy *H*.
*T.H.v.*Level	The Level "L" of the Value "*v*" in the hierarchy "H" of the Type "*T*"
*T.v.*Level(Down M [to N], *H*).*	The domain of all values M to N levels down from the level of *v* that are descendants of *v* in hierarchy *H*.
*T.v.*Level(Down M [to N] , *H*).*	The domain of all values M to N levels down from the level of *v* in hierarchy H
*T.v.*Level(Down M [to N] , *H*)	The name(s) of the Level(s) M to N levels down from the level of *v* in hierarchy H
T.L(Down M to N , *H*)	The name of the level(s) M to N levels

	down from the level *L* in hierarchy H
T.v.Level(Up M to N , *H*).*	The ancestor value(s) M to N levels up from the Level of the value *v* in hierarchy H
T.L.(Up M to N , *H*)	The name(s) of the level(s) M-N levels up from the level *L* in hierarchy H

Categorical-Positional Hierarchies

Recall the discussion in section "x" above. It described how the relationship between a teacher and her/his pupils or a table and its associated chairs was 1-N, but not resolutional.

Ordering expression

The basic ordering relationship that defines a positional hierarchy is as follows:

`T.v[1]–p–[N]T.v.!`

Read: the positional relationship between 1 value from T and N values drawn from the complement of T.v

Atomic functions

Expression	**Meaning**
T.v.(P Up *N*)	The value N positions up from the value v
T.v.(P Down *N*)	The values N positions down from the value P

Open and Multi-ordinal

Networks form the scaffolding for defining more complex structures, such as algorithms and workflow processes. Networks have a complex positional structure, but may also have resolutional structure that must not be confused with the positional structure. The difference between application of a network as an algorithm versus a description of

something else (like a CAD model) is primarily in the representations chosen (p-code versus vectors).

Positional networks

Common attributes

Attribute	**Value**
Arity	Binary to N-ary
Sequenceability	Not applicable
Number of ANDed units	1
Number of XORed units	1
Number of neighbors	0 to N
Constant number	No
Permitted cycles	Yes

Positional networks differ from hierarchies (resolutional or positional) in that the number of values that are adjacent to any value could be any number. Values of the type are the nodes, while the general ordering relationship within the type describe the possible arcs between the nodes. Specific arcs are described with specific value-to-value orderings.

A directed network will distinguish PRE and POST roles within the ordering relationship. Values in a POST role with no corresponding PRE values are considered to be *first* in a network. Values in a PRE role with no corresponding POST value are considered to be *last* in a network. Many useful networks (like classic flowcharts) will have one first and one last value. Other useful networks (like an assembly line) may have many first or many last values. This does not preclude networks that have no first or last values identified.

In the family of positional networks described here, cycles may appear as desired. Cycles may consist of more than one arc, or a singe arc (that is, a value may appear as a PRE node and a POST node in a single ordering relationship). However, no more than one such arc may be present in an ordering relationship.

Ordering relationship

1. (T +[N1] –p- [N2]+ T) AND
Read: Some values participate N1 at a time in a PRE role with N2 values in a POST role. Across all instances of this ordering, a value may appear in multiple PRE roles and multiple POST roles

2. (T.*Last* [N3] –p- [0] T) AND
Read: Some N3 values may be PRE to no values as POST. These will be called *last*.

3. (T [0] –p- [N4] T.*First*)
Read: Some N4 values may be POST to no values as PRE. These will be called *first*.

Atomic functions

Expression	**Meaning**
T.v.Post	
T.v.Pre	The domain of value(s) one step before the value v

Molecular functions

T.V (Step +/-*N*)	The domain of value(s) + or - N Steps away. The direction of – is the PRE direction; the direction of + is the POST direction
T.V.After	Domain of all values reachable after value V of type T
T.V.Before	Domain of all values reachable after value V of type T
T.V.After (M : N)	Domain of values M through N after value V of type T: M >= 0, N >= M
T.V.Before (M : N)	Domain of values M through N before value V of type T: M >= 0, N >= M
T.Between(*V1*, *V2*)	Domain of values reachable by tracing successors from V1 and predecessors of V2.
T.V.First	Domain of values reachable by tracing successors from V to the *first* values.
T.V.Last	Domain of values reachable by tracing successors from V to the *last* values.

Resolutional connections between position networks
Resolutional adjacencies can be established between position networks. For example, a flowchart of high-level state/operation values can be mapped via [1]-r-[N] or [1]-r-[N]+ with other flowcharts (and these will typically have *first* and *last* values within themselves). Typically, the resolutional ordering relationship between values in different flowcharts will be T1.v [1]-r-[N]+ T2.v (with substitution on the down side), as the same instruction may be used by multiple calling functions. A network of organs within a body may be mapped via [1]-r-[N] ordering relations with the constituent cells.

TLH

Families of open multiple ANDed units with XORed differentials per type

Complex number system

Tensors

Euclidian 2-Space

Euclidian 3-Space

Families of closed types

Degrees of rotation

Attributes:

Attribute	**Value**
Arity	Ternary to N-ary
Sequenceability	Yes
Number of ANDed units	1
Number of XORed units	1
Number of neighbors	2
Constant number	Yes
Permitted cycles	No

Create Type Periodic
With Units Integer.(0 to N) where N+1 is the number of values in the Periodic Type.

Periodic.*v1*[1]-P-[1]Periodic.*v2*

Families of mixed open and closed types

Polar coordinates

Spherical coordinates

Types whose values are the names of other types

Reference

Attributes

Attribute	Value
Arity	Binary to N-ary
Sequenceability	Depends on specific ordering
Number of ANDed units	1
Number of XORed units	1
Number of neighbors	Depends on specific ordering
Constant number	Depends on specific ordering
Permitted cycles	No

One can be created using:

```
Create Type TypeName
With Units Ref(Type) AS {T1, T2, T3, …};
```

In the absence of other instructions, the members of a RefType list are positionally adjacent to all other values. This follows categorical ordering. The ordering expression for this would be:

T.v.* [1]-P-[All] *T.v*!

As such, positional navigation such as *T.v*.UP(N) or *T.v*.Descendent are not possible. However, it is easy to supply more specific ordering to define hierarchies or other orderings such as a Rank ordering:

```
Create Type TypeName
With Units Ref(Type) AS {T1, T2, T3, …}
Ordering
      (TypeName.v1.(First to Last – 1)) [1] -p-[1] (TypeName.v2.( First+1 to
Last))  AND
      TypeName.V.Last [1] -p-[0]TypeName.v3      AND
      TypeName.v4[0] -p-[1]TypeName.V.First
;
```

With this ordering, positional functions such as First, Next, Previous, and Last will work just fine.

Since a RefType is restricted to being a list of Types or TypeStructures, it is equivalent to the Essbase concept of an accounts dimension. For example, assuming that the appropriate types have already been created, the following could mimic an Essbase Accounts dimension.:

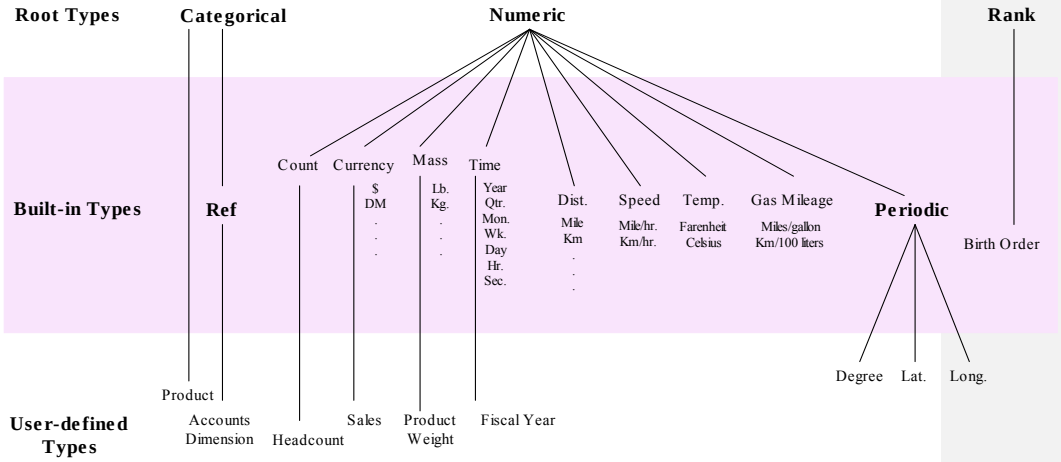
```
Create Type Measures
With Units Ref(Type) AS {
      DirectSales, IndirectSales, DirectCosts,
      IndirectCosts, FederalTaxes, StateTaxes,
      UnitsSold, UnitsReturned, UnitCost
      }
Ordering
      (Measures.*.(First to Last – 1)) [1] -P-[1] (Measures.*.( First+1 to
Last))  AND
      Measures.V.Last [1] -P-[0] Measures.*    AND
      Measures.*[0] -P-[1] Measures.V.First
;
```

A RefType object may be used anywhere a Type object may be used.

Note, in the APB1 implementation we added a `Deref` operator for clarity.

Illustration of root and non-root types

The following diagram shows some illustrative root and non-root types. Please note that the diagram (the source object for which we can not find at present), contains a few glaring errors: Periodic belongs as a root type. And birth order belongs as a user-defined type. Also, these are just illustrative. We will certainly have a wider variety of root types and built-ins.



A General Theory of schemas: Part II of the LC kernel

Introduction

Types, as described above, are the building blocks out of which are constructed both expressions and the machinery –called schemas- by which expressions are created, exchanged, and interpreted. The next two chapters focus on schemas and expressions.

Purpose of this chapter

The purpose of this chapter is to

- Describe in detail the particular attributes that differentiate schema-defining from non schema-defining type structures
- Show why schemas are required to process symbolic expressions,
- Describe how schemas are created from types
- Describe the main kinds of emergent complexities for schemas

Examples of information structures typically called schemas

Before proceeding to give a formal definition to the concept of schema, let's first look in a brief and informal way at a variety of traditionally found schemas based on no more defining characteristics than mentioned above.

A database application, for example, is a schema. It always contains a collection of definitions, typically called the catalog or sometimes schema. Depending on the use state of the application, it may or may not contain any instances. When the application is new, it may not contain any instances. After being in use for a while, it almost for sure will contain a -potentially very- large number of instances. Regardless of how many instances eventually fall under the control of the schema, (which number may be in the quadrillions), the types of values, ranges of values and value relationships are constrained by the schema definition. The database schema is responsible for receiving, storing, manipulating and generating expressions in the form of instances or fields or rows.

In common sense reasoning as a branch of AI, structures called frames are frequently used as a means for defining context-specific behaviours. Frames too are a kind of schema. They contain definitions in the form of assumptions about the environment and slots or questions that the frame will attempt to fill-in with instances during the course of its interaction in the context. The AI schema is responsible for receiving and storing expressions or instances (answers to questions posed by slots), and creating expressions or instances (such as the frame's answers to questions posed by anyone interrogating the frame).

TLH

The basic biological behaviour program for an insect or small animal that consists of first trying to determine whether some sensory-motor pattern is a predator or food or neither and then as a function of that determination either fleeing, eating or 'keeping as-is' is also a kind of schema. Somewhere in the brain (whether hard-or soft-wired), there must be some definitions that govern how a sensory-motor pattern is recognized as food versus prey and how to carry out the specific series of actions required to flee versus carry out the specific series of actions required to feed.

But what exactly is a formal definition of a schema? What is the difference between a type structure that has schema properties and one that doesn't? What is the basis for maintaining that there is a difference?

Distinguishing schema-defining from non schema-defining type structures

Consider a room full of type structures as illustrated in figure "x" below.

Note that every type structure has two components: a definition and a set of instances. Note also that there appear to be a variety of different kinds of interactions supported by the type structures. Some appear to be interactive with other type structures. Others appear to be interactive with themselves. Still others appear not to be interactive at all.

The expression management metaphor introduced earlier is applicable to this figure. The interactive type structures are the expression managers. The things that get exchanged- questions, commands, assertions- are the expressions. It should also be clear how the type structures -qua expression managers- are the "sine qua non" or required foundation without which no expression could be said to exist[20].

All type structures are passively interactive

Regardless of how or if a type structure appears to interact with itself or the world, the world can always interact with it. Thus all type structures can be created, altered, destroyed, queried for their definitions and/or queried for their instances. Another way of thinking about this is to say that all type structures are passively interactive.

Schemas are type structures that can generate interactions

However, not all type structures can initiate interactions with themselves or with the world. The abilities to generate questions/queries, generate commands and generate

[20] The LC systems' focus on schemas as something to be understood prior to that of expressions and the LC System's insistence on analyzing expressions relative to their managing schemas is contrary to canonical thinking whose focus is on expressions and furthermore, most typically on assertions.

assertions are the objectively measurable criteria that differentiate schema-possessing from non schema-possessing type structures and which will be described in detail in this chapter.

Returning again to figure 'x' above, one can see that all but the inert type structures in zone "A", possess some kind of schema. So what is the difference between the type structures in zone "B" versus zone "C"?

Schemas may be abstract and/or concrete

Recall the distinction that was drawn in the introduction between abstract and concrete topics or objects and methods of testing. All concrete objects rely on at least one sensory-motor pattern for the definition of the object. Interaction with concrete objects must occur through sensory-motor devices and be detectable as an exchange of expressions between the entity qua schema and the world which may include other schema-possessing entities. The schemas in zone "B" are the concrete schemas. Concrete schemas are the mainstay of empirical (as opposed to theoretical), models of the concrete sciences.

With concrete schemas, the question creator relies on the world for its answer. The question creator can not answer its own questions. Also with concrete schemas, the command generator can not execute the command just by following rules found in the schema definition.

For example, if Dawn wants to know what time it is and doesn't have a watch she asks Jeff who has a watch. Or if she wants the door closed, she again asks Jeff to close the door. (Example also works with oneself. I still need to look at my watch or tell my body to close the door.)

In contrast, abstract schemas interact with abstract objects that do not rely on any sensory-motor patterns for their definition. Like concrete schemas, abstract schemas may generate questions and create answers and generate and execute commands, but unlike concrete schemas, abstract schemas do not need to communicate or exchange information with the world in order to answer the questions they raise or to execute the commands they generate.

For example, any arithmetic schema that is capable of forming the open question how much is 3 + 6 must also be capable of carrying out the calculation and thus answering the question. If you understand what it means to add 2 + 5, , i.e., if you can ask the question, then you can answer it. There is no need to ask another schema for the instance. Since all instances are definitionally defined.

The General form of a schema

Schemas are an emerging specialization of type structures that have the ability to generate questions and receive answers, as well as generate and execute wants or commands[21]. If one were trying to construct a non-representational expression management system, both processes (asking and answering questions, and generating and fulfilling wants) would be equally primitive. However, since the thrust of this exposition is focused on representational expression management systems as the foundations for abstract science, it is important to recognize that although at a representational level both processes are primitive, there always needs to be some non-representational component to a representational system. In software terms this is called the execution machinery or the CPU and specific machine codes that the CPU executes. And this execution machinery is composed of want/command generation and fulfillment/execution.

The general form of a schema is an ordering relationship between two or more types defining four or more instances wherein:

- Each instance of the schema
    - Associates one specified value that is unique across the schema–called the location-
        - which location is defined in terms of some subset of the original Types-
    - With one specifiable, or specifiable and specified, or specifiable and desired value called the content- that could repeat between locations in the schema-
        - from each of the remaining original Types

If the specifiable value is all there is, the schema defines a question. If the specifiable value is further associated with a specification, the schema defines an assertion. If the specifiable value is further associated with a desired value, the schema defines a command. Each pairing of one specified location and one specifiable content is an elemental question which, when answered, creates an elemental assertion that is either true or false. Each pairing of a specified location with a specifiable and desired value is a command which when executed either succeeds or fails to achieve the desired value.

Schemas have a definitional component and an instance component. Concrete schemas derive their instances from some independent sensory-motor process (itself a schema). The content in a concrete schema is independently determined relative to the location. Simple concrete schemas might generate questions about the temperature at a given time, the color of particular shapes, or the taste of particular foods. In contrast, abstract schemas may derive their instances from the very types whose combination defined the

[21] Note there is a significant divergence in vocabulary between the software world and the traditional philosophy world at this point. Philosophy has always spoken of wants or desires and their fulfillment as the basis for action. In today's software environment the same concepts are spoken about as execution machinery. Software programs are ultimately translated into commands for a processor to execute or carry out. We will use both kinds of terminology depending on the context.

schema. An abstract schema might generate questions across some range of numbers about the sum of each number and its immediate predecessor.

Simple schema examples

For biological organisms, root schemas are hardwired into the critter. The human ability to learn how to recognized physical objects in a multi-modal fashion and acquire and use symbolic expressions is built-in.

Activatable behaviours: skiing down a slope, playing tennis, making drinks,

For computing systems, schemas are a part of the basic software. Depending on the domain, in Relational databases, schemas are called Relations; in multidimensional databases they're called cubes; in parts of AI, they're called scripts and frames.

Concrete Q&A Schema

[need to add descriptive, explanatory, predictive and decision]

Given a shape type composed of square, round, triangle, house and car shapes and a color type composed of red, blue and green

Shape.* 1-1+ Color
Read: For every recognizable shape as defined in the shape type associate some valid color value

This schema's definition defines a question template or form repeatable for every unique value of shape. Although in a full working schema there would need to be more complexity including some connecting of the shape-color schema to a sensory motor device, the concept, that it takes at least one type with specified values and one type with open values to create a schema capable of asking questions should be clear.

If both the shape and color type were open, the structure would not be able to pose questions –"What is the color of some shape?" does not constitute a question- and would hence not function as a schema.

If both the shape and the color type were specified. The structure would have any questions either and hence would not function as a schema.

This kind of schema is extremely prevalent in the database world. Databases are designed and initialized as huge repetitive questions which, through use, get filled up with answers- names and addresses of clients, what they ordered, who took the order, what was shipped and charged, method of payment etc… So let's look at a simple concrete Q&A schema in more detail.

Consider the diagram below. It is a simplified picture of a symbolic expression management system in use. It shows a collection of Types being used to define a schema called "Sales schema". And it shows that schema being used to query data from a data source.

Figure 2. A Basic Sales Model

There are two major system component interactions illustrated in the diagram: interactions between schema definitions and data sources (or the external world) , and interactions between Type definitions and schema definitions.

This Q&A schema defines a collection of queries or questions that anticipates a collection of corresponding assertions (better thought of as answers) For example, the cube schema illustrated in figure 2 whose dimensions are Time and Store and whose measures are Sales and Costs starts off defining a whole bunch of questions. Specifically, for each Time and Store the schema asks two questions: "How much was sold for that Store at that Time?", and "How much did it cost?". After connecting to a data source, which functions as a source of answers to the questions just posed, the schema picks up (senses) and manages the answers. A single schema may represent hundreds of millions of question/answer pairs of the same kind.

The actual values for Stores and Time used in the schema come from the potentially usable values for Stores and Time as defined in the Type definitions. However, the actual values for Sales and Costs in the schema, come from the external world or data source. The reading in of the Sales and Costs data (regardless of whether it happens at 'load time' or 'query time') corresponds to the schema learning sales and costs facts about the world. The ability to learn facts about the world is a fundamental property of a well formed Q&A schema.

The process of answering questions, in this case sales and costs questions, is so fundamental, and is the basis for the terms Location and Content (whose initials form the name of the LC System), that it is worth stepping through slowly.

The first step involves selecting a specific Store-Time tuple in the Sales schema, and in combination with the Sales Type and the Costs Type, for which no actual values have been supplied, looking for matching values in the external world. Finding a matching value in the data source defines the act of locating. Any Type used for this purpose is called a locator relative to that schema. The combination of Types used for locating within a schema is called a Location structure. Each location structure tuple in the schema is called a location. The schema uses the value "Paris" from the Type "Store" and the value "Spring" from the Type "Time" to locate a Store-Time location called Paris-

Spring in the data source. This is how dimensions are used in OLAP models and how logical subjects function within natural language.

When (and assuming that) a matching location is found in the data source, the Sales and Costs Types are evaluated. The term "Content" in the LC Model refers to whatever exists at a location. The combination of Types used as contents within a schema is called a Content structure. Once the values for the Sales and Costs Types have been evaluated for the location Paris-Spring, the values are returned to the schema. In the figure, we can see that the value for Sales in Paris-Spring is $100. This equates to the well formed assertion "The Sales for Paris in the spring are $100"

Although the schema in figure 2 used Stores and Time as locators and Sales and Costs as contents, there is nothing about Stores or Time that makes them locators. And there is nothing about Sales or Costs that makes them contents**. The distinction between locators and contents is a functional distinction not a structural (or referential) one[22].**

For example, the Types Sales and Costs could have been used as locators. And the Types Stores and Times could have been used as contents. By posing the query what Store(s) had sales of $100 in the Spring, the query-answering process would have begun by looking for a location in the world whose Sales in the Spring were $100 and then returning the name(s) of the Store(s) associated with that Sales-Time location. Here, then, Stores is functioning as a content.

It doesn't make sense to label Stores or Times as locators. Nor does it make sense to label Sales or Costs as contents. As shown in chapter two above, they are all types.

Concrete Command Schema

Given a type called "Week" whose values are the names of weeks and a "Phone home" type whose values are yes or no

[22] Bring in Aristotle thru Frege to LW

Discipline	**Token set one**	**Token set two**
Language	Subject	Predicate
Logic	Argument	Function
Math	Independent variable	Dependent variable
LC	Types used as locators	Types used as contents
Multidimensional databases	Dimensions	Measures

TLH

Week.* 1-1+Phone home

Read: Every week either phone home or don't phone home

Concrete Q&A with Commands Schema

Abstract Q&A Schema

Abstract Command Schema

Abstract Q&A with Commands Schema

Schemas are required for any expression processing

TLH

Although types provide the potential values that define and match any token in any symbolic expression, absent some ordering relationship-defined schema between types, there would be no way to interpret, process or create expressions.

Consider, for example, a very simple expression created from just two types: object shape and color, "What color is the house?". On the surface, it seems too simple to require any special processing. Clearly the expression refers to some object called house and a question about the color of this object. So how is it that house has only one color? And how is it that the term "house" uniquely identifies a location?

The identification of "house" as a unique identifier, at least in this context, combined with the binding of some one valid value of color to the unique location defined by "house" is what is defined in the schema that serves as the backdrop for interpreting the question "What is the color of the house?".

Thus, the schema that is a part of the processing of the expression "What is the color of the house?" contains at least a fragment like the following:

"Object.***** 1-1+ Color"

And Color, Object.***** = Visual schema: Color(object)

Read: for every unique object there exists some one valid value for color and those colors come from some visual schema

As seemingly obvious as this schema might appear, numerous other schemas could have been created from the same types such as

Object 1-N color
read every distinct object may have multiple colors

Color 1-1+ object
Read: colors are the more natural locators with every unique color being associated with some valid value of object

(object1-n object)1-1 color are all possible but would not match experience
Read color is an attribute of relations between objects rather than objects in isolation

Need to add: there exists a sensory experience schema, say for stored visual experience, including recognized objects or patterns, a sensory experience analysis schema for assigning symbolic identities to input expression strings and a schema that maps recognized expression tokens into executable expressions

Defining schemas from types

So how was the schema "Sales schema" defined? How were Time and Stores designated as locators? How were Sales and Costs designated as Contents? And how might some other kind of schema been defined? In short, the same ordering relationships that give shape to types, give shape to structures of types. When those structures have a form that is capable of supporting questions, commands and assertions, they form schemas. Recall from the previous chapter that all ordering relationships between well formed types are well formed type structures. But only those type structures with certain measurable attributes have the required properties to serve as schemas.

The Sales schema was defined by taking the Cartesian product of Stores and Times, as defined in the Type definitions, and for each intersection defined by that product asserting that some valid value (as defined in the Type definitions), for each of Sales and Costs was applicable.

Time and Stores were implicitly designated as locators because the actually used values for Stores and Times in the schema came from their potentially usable values as defined in the Type definitions. Sales and Costs were implicitly designated as contents because the actually used values for Sales and Costs came from a data source (not the Type definitions).

There are other ways that the two Types "Stores" and "Times" could have related for the purpose of defining the location structure of a schema. For example, their relationship could have been defined in terms of arbitrary tuples. If this were the case, the location structure for the schema would consist of some number of unique "Store-Time" tuples. For example, the following tuples would constitute a valid location structure: {Paris-Spring , Madrid-Fall , Thule-Winter , Quito-Spring , Bangkok-Summer}. But then it would not be correct to treat Stores or Times as orthogonal dimensions.

Emerging complexities for schemas

<u>Summary of schema attributes</u>

Number of types
Definition of each type
    Kind of root types
    Scoping of each type that participates in the schema
The ordering relationship(s) between the types
    Data-like
    Action-like
Kind of physical representation for the schema
    static/dynamic
    process

# of context specific physical representations
Degree of independence/dependence between the locators
Number of locations
Degree of dep/independence between contents
For each location, number of contents per location
Ratio of derived to measured values
Schemas all of whose values are derived
Relative location of measured and derived values
Homogeneity of definitions
# of location-based app ranges
# of data-driven ranges
Schemas that send and/or receive information from 1-N other schemas
Schemas that can activate or deactivate other schemas
Blending of schemas

Scopes, Context or Application Ranges

In a nutshell, the term "application range" refers to the fact that definitions of any kind (i.e., constructs, ordering relationships and/or functions) may vary as a function of use location. Thus, for example, the definition of a Profit function may vary between schemas or between locations within a schema. How one greets an elder may vary from culture to culture. The rules for playing blackjack vary from state to state. The potential values for a Type may vary between Schemas. In American Roulette, there are 38 potential values; 1-36 plus zero and double zero. In European roulette there are only 37 as they do not have the double zero. Or the ordering relationship between the values in a Type (as, for example, a hierarchy), may vary across Time. When the Soviet Union broke up, what had once been states of the USSR became countries.

For example, the data source for the actual values of a hierarchical product Type may vary as a function of whether one is populating values that are below "`Toys`" or below "`Furniture`". The names of Toys may come from some Table A while the names of furniture may be sourced from some other Table B.

Or, the formula for a Type, say "`Profit`" may vary as a function of the Type Structure or model in which it is used. In some model, say a "`New Product Sales Model`", the formula for "`Profit`" may be "`Sales" – "Costs`". While in some other model "`Old Products Sales Model`", the formula for "`Profit`" may be "`Sales` " – "`Direct Costs`".

Or, the formula for a Type such as "`Profit`" may vary within a single Type Structure as a function of the location within the Type Structure. Thus for years prior to 1998, in some model, the formula for "`Profit`" may be "`Sales`" – "`Costs`". While for years beginning with 1998, in that same model, the formula for "`Profit`" may be "`Sales` " – "`Direct Costs`".

It could also be that the set of potential values for a Type such as "`Product`" varies by "`Time`" within a particular model. This corresponds to what are sometimes called "slowly changing dimensions".

Since the models, between which and within which Types can have varying definitions, may themselves represent historical views, (in other words one needs the ability to support models of history that vary in time and space), the LC System supports both an inner and an outer context for defining and maintaining Type specifications.

The inner context is the definition of the Type itself. Application ranges may be used within the definition of a Type's name or actual values or ordering etc.. The outer context is supplied by a built-in system schema that tracks the use of Types across different times and locations.

Thus, for example, one might define in the year 2000, within the specification of a "Product" Type, that the specific values for Product vary across three years: 2000, 2001 and 2002. Models used in the year 2000 that contained the Type "Product" would reflect the year-specific specifications that were made. Now, if in the year 2001, one were to edit the definition of the "Product" Type so that the specific product values for the years 2000, 2001 and 2002 were different from those initially specified in the year 2000, those changes would be reflected in the built-in system schema that tracks changes in Type definitions. The ability to set application ranges within Type definitions combined with the existence of a built-in Type maintenance schema adds significant richness to the LC modeling environment

Heterogeneous Schemas

Schemas have so far been treated as relatively homogeneous Type Structures. Of course, this is frequently not the case.

Schema Contents are not always applicable to all Locations

- Some products may be defined as having Sales that are not applicable to some store locations
- Some employees may not receive commissions

There are numerous ways by which a homogeneous Schema may become Heterogeneous. Those ways may be grouped in two clumps:

- Internally, transforming the schema in some way
- Externally, linking with some other Schema(s)

For examples of a Homogeneous Schema transforming in some way, consider

- The addition of a new Type to a portion of the Schema
- The deletion of a Type from a portion of the Schema

- A change in any formula definition
- A change in any hierarchy definition
- A change in any connection to a data source
- A change in the application range of Contents to Locations
- Changing the ordering relationships between the Types in the Schema
- Changing the ordering relationships between the Potential Values in the Types

For examples of linking, consider

- The join of two otherwise homogeneous SQL tables resulting in one heterogeneous table
- The join of two otherwise homogeneous Hyper-Cubes resulting in one Heterogeneous Multi-Cube

Heterogeneous Schema Expression Examples

5. Sales, SalesModel = Sales, ProductionModel

Read: The Content "Sales" in the Schema SalesModel is assigned the values of the Content "Sales" in the Schema Production Model. This assumes that the two Schema have the same Location Structure

6. Sales, Sales Model = Revenue, Production Model

Read: The Content "Sales" in the Schema SalesModel is assigned the values of the Content "Revenue" in the Schema Production Model. This assumes that the two Schema have the same Location Structure and that the two Contents have commensurate Units.

7. Sales, SalesModel = Revenue, ProductionModel.[(Time[1]+ Geography[1]+).*[1]-[1]+(Sales, Costs)]

Read: The Content "Sales" for all locations in the Schema "SalesModel" is equal to the values of the content "Revenue" for that subschema in "Production Model" defined by "Time[1]+ Geography[1]+).*[1]-[1]+(Sales, Costs)"

SalesModel , SYSTIME.This = ProductionModel , SYSTIME.Jan03

Read: All the concept definitions and values of The Schema "Sales Model" for any SYSTIME are defined in terms of the concept definitions and values of the Schema "Production Model" for the SYSTIME Jan03

TLH

SalesModel , SYSTIME.Apr04 = Production Model , SYSTIME.Jan03

Read: All the concept definitions and values of The Schema "Sales Model" for SYSTIME Apr04 are defined in terms of the concept definitions and values of the Schema "Production Model" for the SYSTIME Jan03

SalesModel , SYSTIME.This = ProductionModel.[(Time[1]+ , Machine AS Source[1]+).*[1]- [1]+(Output , Defects, Quality) , SYSTIME.this

Read: All the concept definitions and values of The Schema "Sales Model" for any SYSTIME are defined in terms of the concept definitions and values of the Schema "Production Model" subschema defined as Time[1]+ , Machine AS Source[1]+).*[1]-[1]+(Output , Defects, Quality) wherein the Type called "Machine" in the Production Cube is cast/aliased as a Type called "Source" in the "SalesModel" for that same SYSTIME

Sales, SalesModel.Location = Sales , ProductionModel.Location.UP(1)

Read: The Content "Sales" in the SalesModel for any location is equal to the content "Sales" in the Production Model for the level combination One UP from that of SalesModel.

Applicability , Content.Sales , Sales Model = Location.[(Time.1999.Month.*[1]+ , Store.MA.Below[1]+).*[1] - [1]+([Forecast])

Read: The content "Sales" within the Schema SalesModel is applicable to all the months of 1999 combined with all stores below MA and for those locations where the Content "Forecast" is also applicable.

Schema Join examples

There are several basic joins:

- A and B each have one or more Types that share a common ancestor
- A and B each have a Type wherein some transformation or aliasing of the values in one of the Types converts it into values of the other Type
- A and B each have a Type whereby some transformations applied to each of the Types is capable of converting each of the Type's values into the values of a common Type

- The Types need not be joined by matching values (whether immediate or through some transformation). The values of one Type may be in some range or defining of some range relative to the values of the other Type

1. Within a Type structure "A" defining one of the Types (T2) in terms of another Type structure (B) wherein A and B share a common Type A1 and wherein the values of the Type T2 are defined in terms of in Type structure A in terms of the values of the function of (A1, T3, T4, T5)aliased As T2 as found in Type structure B. Note how the values of T2 are defined in terms of A1 in TsB and how that information is used in Type structure A to fix the set of valid T2 values for each value of A1.

Method 1: Reifying the transformation

A = (A1 r T2)
B = (A1, f(A1, T3, T4, T5) AS T2

Method 2: Without reifiying the transformation

A = (A1 r f(A1, T3, T4, T5))

Standard Joins

Non-standard Joins

Use of Aliasing

Exact Match

Range/Proximity

Function-based

Aliasing-based

Nesting

Systems of mixed concrete and abstract schemas

General schemas

Example relational and OLAP schemas:

A general theory of symbolic expressions: Part III of the LC Kernel

Introduction

The purpose of this chapter is to

- Identify the different processing phases of a symbolic expression
  - Describe the constraints on the executable form of a symbolic expression,
  - Describe the constraints on the exchanged form of a symbolic expression as a function of unexchanged context
  - Describe the process of measuring the degree or state of well formedness of a symbolic expression
- Describe the basic forms of symbolic expression,
  - Highlight some subtle yet important differences between symbolic expressions that refer to types and schemas and expressions that refer to anything else,
- Describe the main kinds of emergent complexities for symbolic expressions
- Show an example of a canonical symbolic question that gets answered via appeal to sensory-motor information

Central questions

The common questions that need to be answered by any approach that purports to provide operational definitions for expressions include the following:

Is there a general notion of well formed formula? If so what is it? If not, why isn't there one?

What are the properties of a well formed formula? What is it that can be done with a well formed formula that can't otherwise be done?

Is there a formal basis for so-called shared context? In other words, is a totally context-free grammar even possible? If some information must be pre-existent and shared so that other information can be exchanged, what is the relationship between the exchanged information and the shared information.

What does meaning really mean? Is the meaning of an assertion its truth conditions? Is the meaning of a question the process by which the question can be answered? In which case can the meaning of a question differ depending on how it is answered? What about the meaning of a word, or the meaning of a command? Does the concept of meaning differ as a function of whether the word refers to a concrete object or a linguistic or

abstract object? Is meaning just reference? But then are dreams meaningless? And what about Frege's distinction between sense and reference?

The role of schemas for exchanging and executing WFF

Consider the following examples of typical well formed expressions

- 5 +7 = 12;
- What color is the house?
- Close the door!
- There exists an X such that X is human.

These expressions represent what is exchanged between two entities within the context of a dialogue (or an entity in dialogue with itself). This is illustrated in figure "X" below.

Figure "X" The exchanged form of an expression

The traditional concept of well-formed expressions or formulas refers to constraints on exchangeable expressions that are testable at the time of exchange. From a language perspective, one might say the traditional constraints are syntactic. From a software perspective, one might say that traditional constraints are based on the parsed (as opposed to compiled), form of the expression.

Thus, for example, in logic, expressions are said to consist of a variable assigned a value resulting from the application of a function to an argument. In language, expressions are said to consist of a noun phrase and a verb phrase or a subject and a predicate. In either case, the definitions of "Value", "function", "argument", "noun phrase", "verb phrase", "subject" and "predicate" are based on observable attributes of expressions as they are being exchanged.

But what happens between the time that an expression is received and parsed and the time that something happens as a result of the expression? That result could be, for example, an answer, a counter-question, or an action.

Something happens. A command to "Close the door" might get translated into sensory-motor signals that attempt to close the door. A question like "What color are the leaves on the maple tree in the backyard?" might get translated into sensory-motor signals that attempt to observe the maple tree's leave color and then generate the appropriate verbal answer such as "Full fall majesty!"

Somewhere along the line, the series of tokens that comprise the exchanged expression get transformed (via some schema) into "something that can translate into or become action" or "something that can be executed" or "something performative". And then some action occurs.

Furthermore, there are constraints on the form of an expression in its executable form in order for it to be able to be executed. Not anything can be executed. This is what we call the well-formedness constraints on the executable form of an expression.

This second stage in the processing of an expression and its relationship to expression processing as a whole is illustrated in figure "X" below.

Figure "X" The executable form of an expression

In light of the discussion above, a more complete metaphor for an expression-based dialogue would be that of two entities each of which was equipped with a program or schema that provides parsing, a program/schema that provides compilation and some kind of execution machinery engaged in a dialogue that consisted of exchanging pieces of executable source code. This allows us to speak of the constraints on exchangeable source code, the constraints on an executable or potentially executable program (or compiled code), the executing of a program, and any result (whether or not returned), from executing the program.

- A publicly exchangeable question, or query, is a set of tokens which when compiled or processed by a schema-defined symbolic expression manager generates a potentially executable program
- A publicly exchangeable command is a set of tokens which when compiled or processed by a schema-defined symbolic expression manager generates a request to execute a potentially executable program
- A publicly exchangeable assertion is a set of tokens which when processed or compiled by a schema-defined symbolic expression manager generates potential results from executing a potentially executable program and may be tested

The constraints on a symbolic expression manifest themselves most clearly and completely in the process of converting a set of expression-defining tokens into a potentially executable program. This is because any shared context that was not a part of the exchanged form of an expression must be made explicit before the expression can be executed.

Since the process that governs each mapping or transformation of an expression is a schema, schemas will be described in detail (in this chapter), before getting to expressions (in the following chapter).

TLH

Where expressions diverge from schemas

The common origin of schemas and expressions as well as the complexity point at which they begin to differ is best understood via an illustration. Thus, consider the diagram in figure 'x' below. It represents a simple exchange of symbolic information between two entities A and B wherein both entities have shared analog information. To be more concrete assume that the symbolic information is verbally exchanged and that the shared analog information is visual.

Both entities share the same underlying types: Visual shapes, Visual colors, verbal shape words and verbal color words.

Imagine the types are loosely defined in the following way:

Visual_shape = (output of some visual pattern detection algorithm)
Visual_color = (output of some visual color detection algorithm)

Imagine that the main object in the shared visual field for A and B is a tree. A, however is lying under a bench and can only see the trunk. B is in plain sight of the entire tree. A now asks the question "What is the color of the tree leaves?" To which B responds "Orange."

The flow of information from A's initial formulation of the question through B's visual action to obtain the information to A's reception and interpretation of the answer is illustrated in the diagram below.

Figure 'x'

At its simplest, all that is strictly required in the mind of A is an ordering relationship between the one visual shape value "tree leaves" and one valid value of the visual color type

Visual_shape.tree_leaves 1-1+ Visual_color

This ordering relationship-defined type structure may be thought of as defining either a simple schema or an expression in the form of a question.

A more likely schema that A and B might have shared would have been

Visual_shape.* 1-1+ Visual_color

Read: All visual shapes have some (valid) color

In other words, the fact that objects have color is more likely to exist in a reasonably general schema that gets reused many times with many different specific expressions.

Here now is visible the divergence of the single expression and the schema that covers a potentially limitless number of specific expressions.

Note that both A's created question expression, and B's schema that interrogated the world had what was defined earlier as an LC question form where types playing the L role had specified values and the types playing the C role had unspecified values.

This LC form is necessary for any expression or schema to represent or generate questions. All schemas whose values come from interacting with or observing the world need to have this LC form else they would not be able to accumulate any values.

Although schemas are an emerging specialization of type structures, since (postulating without proof for the moment that) all learning stems from asking (and then trying to answer) questions, schemas, like expressions, need to be understood in their own right.

Understanding what's not exchanged

Shared knowledge of the world

Consider the two figures below. They both show an exchange of the same information in the form of a question-answer dialogue between two individuals. In figure 1, the initially sent expression consists of a single token "Huh?". While in figure 2, the exchanged expression is considerably more detailed. Specifically it consists of the string of tokens "Why is the bridge covered in pink blankets?" Note that the answer is the same in both dialogues.

Figures 1 and 2

How does “Huh?” come to mean the same thing as “Why is the bridge covered in pink blankets?” The answer lies in the shared knowledge of the world between the individuals engaged in dialogue.

Shared schema and type definitions

The fact that every distinct tuple of time and store denotes a unique location and the fact that for each distinct tuple of store and time there is associated one valid value of sales is a crucial part of the sales schema but is not a part of what needs to be exchanged by expressions shared between instances of the sales schema.

Questions like “How much was sold in paris in the spring?”, or, “Who sold more last year Paris or Rio?” do not need to include this schema-defining information. Simply put, these expressions presuppose the existence of the sales schema. If the sales schema did not exist, the expressions would be neither createable nor interpretable. Furthermore, for the question “How much was sold in Paris?” to be a part of a question-answer dialogue, both parties need to have this schema. This again is why the schema and the types in terms of which it is defined are called the foundations for the expressions subsequently exchanged.

These examples highlight the fact that any model of symbolic expressions must deal with the interdependency between the constraints on the exchanged symbols or tokens and the constraints on the unexchanged type structures or schemas that serve as foundations for the particular symbol exchange process. Simply put, the more there is shared context, the less needs to be exchanged.

The different processing phases and forms of symbolic expressions

Processing phases

Based on the discussion above, we can identify the following distinct phases in the processing of symbolic expressions.

1. Analog representation of exchanged symbolic expression phase = direct capture of expression

2. Symbolic parsing of analog representation phase = assignment of schema and type tokens to expression

3. Compilation of symbolic representation phase = addition of unexchanged schema and type tokens including orderings and operators resulting in a potentially executable expression

4. Execution phase = attempting to execute the potentially executable expression
   - may result in analog expression i.e. some sensory-motor performance
   - may result in a new expression in potentially executable form

5. Construction of exchangeable symbolic expression from expressions in executable form = the reverse of phase 3

6. Analog output of 5 = reverse of 1

Figure "X" The different processing phases for a symbolic expression

It is important to recognize that these phases are best thought of as loosely coupled processes. They are not strictly sequenced. Parts of expressions may be parsed, compiled and executed before the expression has been fully received. In the case of nested expressions, such as "What is the color of Bob's mother's favorite tree" parts of the expression (Bob's mother's, and then Bob's mother's favorite tree) need to compiled and executed before the rest of the expression can be evaluated.

That said, since the constraints on the exchanged form of a symbolic expression may be looked at as a subset of the constraints on the executable form, we begin with the latter.

Forms

There are nine basic forms of simple or atomic expressions based on the purpose and topic of the expression. The three basic topic distinctions are
- Type– topic expressions
- Schema- topic expressions
- Anything else-topic expressions

The three basic purposes or forms[23] are: Questioning, Commanding and answering

[23] The types used in any expressions can be used to define questions, commands or answers.

Note how it is only expressions about types and schemas that must include any tokens that denote ordering relationships. Yet at the same time, all expressions presuppose some kind of background schema, even the expressions about types and schemas!

Let's look at representative examples of expressions as a function of their form and topic as illustrated below.

Topic\Form	Query	Command	Assertion
Type	+ How many hierarchies are defined for "Geography_one"? + Is the Store type categorical or rank? + How many Units are defined for "Time"?	+ Evaluate the value-defining function for the Type "Geography" + Count the number of Units for the Type "Time"	+ Type "Time" has 3 hierarchies. + The units for "Profit" are dollars.
Schema	+ Is sales applicable to January? + How many locations are in the "Blue Lagoon" schema? + What's the ordering relationship between "Time" and "Product" in the "Sales schema?	+ Evaluate the leaf level contents for schema "Blue Lagoon" + Lookup the ordering relationship between "Time" and "Product" in the Sales schema.	+ In schema "Blue lagoon" Profit = Sales – Costs + The ordering relationship between Time and Product in the Sales schema is (1+, 1+).*
Rest-of-world	+ What color is the house? + Is the door closed? + How much is 2 + 3?	+ Measure the color of the house! + Close the door! + Evaluate the sum of 2 + 3!	+ The house is green + The door is closed. + 2 + 3 = 5

The WF constraints on the executable form of a symbolic expression

Introduction

Let's begin with a simple working example.

"What is the color of the ball?"

Figure "X" showing dialogue

Imagine a simple object type with potential values "Ball, bag, car" and a simple color type with potential values " red, blue, green"

TLH

Imagine also a simple schema: Object.* 1-1+Color
Read: every unique object has one valid color

Executing an expression requires that all implicit references are made explicit. And there are several implicit references in this example so far.

First, the notion of object. It's one thing to assert that every unique object has a color. It's another thing to define how an object is located. Where exactly is entity B supposed to look to find this ball?

Let's add some additional information

Object, Visual record schema = (output of edge detection algorithm)

without getting into the specifics of the edge detection algorithm one can assert that some function must exist that specifies how the object type treated as content is supposed to be evaluated relative to some schema which is the active schema in place for processing the question "What is the color of the ball?".

If the ultimate source of information were a database table rather than visual sensory information, there would still need to exist some object detection function. It would just look correspondingly different. The location of the object would be a row in a table. Thus the color of the ball would be the color value that has the same rowid as the row for which the object column has a value of "ball"

Object, Table schema = (rowid (object string))

Second, there is crucial ordering relationship information present in the schema but absent from the exchanged expression that is needed to execute the expression. Once the object ball is found, how many instances of color are to be evaluated?

Third there is the root function that separates

Fourth there is the order of execution

Overview of the resolution process

Expressions in their exchanged form are projected onto the schema(s) that participate in the expression exchange[24]. This accomplishes several things:

[24]

1. Identify each token in the exchangeable expression

- The active schema is restricted in terms of the values specified in the exchanged expression
- Additional information is picked up:
  - unstated ordering relationship information
  - unstated equivalency operators
  - unstated associated type definitions
  - unstated associated schema instances
  - unstated schema instance derivation definitions
- The exchanged expression is recast as an executable process in terms of the active schema definition which includes all the additional information

If the schema is fully instantiated (or materialized- rather than uninstantiated, unmaterialized or virtual), the executable expression can be executed against the schema instantiation

If the schema is not fully instantiated, and there exist definitions for schema instances stated in terms of other schema definitions and instances then

substitute in the associated schema definitions and test whether its instances are materialized or virtual

Proceed until either the executable expression can be executed against materialized instances or until the substitution process fails.

If the executable expression can be executed against the materialized instances, execute the process substituting specified values for specifiable values until the process completes.

WF Constraints

---

2. Parse tokens by category: Construct:Type names, Construct:Unit names, Construct:Type values , Construct:instances , Operator:equivalency , Operator:manipulation , Orderrelation: quantity, Ordrel:adjacency, Ordrel:uniqueness
3. Follow concrete syntax rules for associating tokens in a fully connected di-graph
4. Read definitions of identified types
5. Read presumed schema composed of types
6. Create parse tree with type value pairs as leaves connected by ordering relationship and function operators
7. Identify root equality function
8. determine whether expression is potentially executable and/or testable
9. Execute or test

TLH

WF expressions in executable form typically resemble the outline of a biological tree: There is a collection of roots that funnel into a central point and then fan out like branches and leaves. Process flow begins with the roots and proceeds towards the leaves.

The main direction of flow is from specified values towards specifiable values. In some cases, explicitly stated specified values are assigned to specifiable values. In other cases, explicitly stated executable processes, when executed, return specified values which are then assigned to specifiable values. The central point in the process is an assignment operator.

Examples:

Object.house, Color?

Against schema definition: object.*1-1+Color

And instance specification =
table_colors: (Row.*1-1Column2AS Object.*)1-1+Column3 AS Color

And Color, Object.* =

Object.House 1-1Color F= Color.value

1 [(Object.house)location in schema instance]
1[Color(location in schema instance)]
1[F=(Color)]

The WF constraints on the exchanged form of a symbolic expression

Some assumptions about shared context required for any WF constraints

Canonical assumptions

Strictly speaking, there are no fixed or necessary or inherent constraints on the exchanged form of an expression. Anything could mean anything. Silence shared between two individuals could mean "That sonata was blissful." A raised eyebrow could mean "Attack the sentry now!". The sounds, Ugh Ugh, uttered by a baby could mean "Feed me now!!" or "Change my diaper now!!" , or "Pick me up now!!" depending on the context.

So why is there such an obsession with the rules of well formedness for exchanged expressions? Is the concept of a complete sentence arbitrary?

NO! The concept is not arbitrary. But there are important, typically unspoken, attributes and qualifiers that need to be made explicit in order to sufficiently constrain the problem:

Regarding attributes, it's important to distinguish between the following:

- Being parsable by receiver
  - Into the same form as begun with by the sender
- Being understandable by the receiver
  - In the same way as intended by the sender
    - With successful information exchange
- Being testable by the receiver
  - In the same way as intended by the sender
    - With increasing and maximal confidence

For an expression to be parsable, it must be transformable into an executable form as shown in the section above. For an expression to be understood[25], the executable form must be executed. The execution, depending on the kind of expression, might produce a stored memory if an assertion, an answer if the expression is a question, an action if the expression is a command.

Theories of language have typically focused on parsability by the receiver as the criterion for well formedness. In contrast, scientists in general (and logicians in particular), have more typically focused on a shared notion of testability[26]. Testability, of course, is only relative to one kind of expression- an assertion.

In addition to these differences in desired attributes for a WFF, there are also differences in background assumptions. There are four standard assumptions:

- The receiver has the appropriate relative foundations to process the expression.
  - Includes the active schema and
  - The types in terms of which it is composed.

[25] As such the term "understandable" is used in its wider sense. One could say that a person shows that s/he understands the command to close a door by closing the door.

[26] More specifically, they have usually treated testability in a more emotionally-laden and thus polemical fashion. Specifically, either verifiability ala Ayer or falsifiability ala Popper

- Exchanged words corresponding to type values are unique across all types in the receiver's mind. This implies that there is no need to exchange type identifier information to qualify type value information. Of course, this condition is not always met in real life. Ambiguity is a frequent occurrence.
- When a type name is associated with a type value it converts into the executable command to read the schema instance location of the specified type value followed by reading the value of the type associated with the given specified value
- No part of the expression is understood by or part of the active context of the receiver

As shown in the preceding section, the simplest executable expression that returns a result has two components each consisting of an operator applied to a construct in an ordering relationship with the other component.

For example: [Object.House)Location_of]1-1[(Location_of Object.House)Color_of]

It was also shown that the discovered value of the evaluated type, in this case color, need not be assigned to itself in which case there would be a third component in the executable expression as shown below.

-1[(Color_of Location_of Object.House)Assign_to Color]

That said, it is simpler to assume that discovering a value at a location is equivalent to assigning its type the discovered value in which case we can rephrased the two components of an expression as shown below.

[(Object.House)Location_of]1-1
[(Location_of Object.House)Color_of_discovery&assignment]

LC System assumptions for a minimalist exchanged form

We can now state the Well formedness constraints on an exchanged expression based on the following assumptions.

- Since ordering relationships are a part of the schema definitions, they do not need to be exchanged. (Of course, there are numerous grammatical conventions that are used to exchange this information such as definite versus indefinite articles and singular versus plural formulations.)
- Since the Location_of operator can only (and is assumed to) apply to a specified value, it does not need to be exchanged.
- Since type values are assumed unique across the types in the active schema, type identifier information (in this case Object.) does not need to be exchanged. And

- Since it is assumed that an attempt will be made to answer a question, the assignment operator for the open type does not need to be exchanged

WF Constraints

This leaves

"Type value, Type name" as the simplest exchangeable question

The role played by the Type value in the execution of the expression (in its executable form) is what is called the locator or L role in the LC System. The role played by the type name in the execution of the expression (in its executable form) is what is called the Content or C role in the LC system.

If there were no specified type value, there would be nothing to fix on. No location would be specified. For example none of the three expressions below provide a specified value or location:

- What is the color of some object?
- What is the name of some person?
- What is the mass of some object?

There is no specific object whose color is to be evaluated; there is no specific person whose name is sought, not any specific object whose mass is to be evaluated.

"Type value, (Type name, Type value)" is the simplest exchangeable assertion where there is no ambiguity as to which type is evaluated (asserted, predicated) relative to which.

Although it is possible to exchange only "Type value, Type value" there is no guarantee that there is enough information in the active schema to determine which type value is asserted of which. And since, (as will be discussed at length in chapter 'x' below), the truth conditions for an assertion depend on which type is asserted of which, an unambiguously truth-testable expression needs to exchange "Type value, (Type name, Type value)".

That said, if one adds the further assumption that only one of the type values has a count of instances equal to one in the associated schema instance, then that type can be unambiguously assigned the role of locator.

If either or neither type could serve as locator, then the expression has multiple interpretations

In the former case where either type could serve as locator, any order of execution of the executable form will be successful. In the case where neither type can serve as a locator, any order of execution will be unsuccessful.

That said, execution may still take place. For example one could test the assertion, "The car is green.", when there are several cars of different colors and several patches of green associated with different objects.
such a testing amounts to transforming the original question into something like "From amongst all the cars and all the green patches is there at least one car that is green or at least one green patch that is a car?

Examples

This section uses the principles introduced above to show a variety of expressions and their emerging complexities, both standard and type- and schema-relevant.

Some additional concrete syntax

In order to think clearly and in a detailed fashion about the grammar of expressions, it is best to work through specific examples which requires the adoption of some concrete syntax. All examples use the concrete syntax defined in appendix "A" parts of which are repeated below.

Concrete expression processing tokens

- , comma; used to separate types
- ( ) parentheses; used to bound expressions that are calculated at one time. Calculations proceed from inside out; context is passed from the outside in.
- { } curly braces; used to denote tuples.

All expressions are terminated with a semicolon.

A general example of a fully qualified query/calculation expression is:

```
WITH Type Definition1      K1 = (K3 []–[] K4)
     AND Type Definition2  K2 = (K5 []–[] K6);
     AND
WITH  Schema Definition    K7 = (F1(K1) []–[] F2(K2));
Query/Calculation Definition F3(K7);
```

We could also write the last line as
```
  Query/Calc Definition F3((K1 OR K3 OR K4) AND (K2 OR K5 OR
K6))
```

Read this as:
Given Types K1 and K2, defined in terms of Constructs K3, K4, K5, and K6 ,

TLH

Related and mutually scoped by a Schema-defining relationship K7 ,
Our query/calculation is some function F3 of this schema K7.

If the Types and Schema are a part of the active context, then the query/calculation expression can be stated alone.

A well formed query expression
is defined by `T1, T2.V2, T3.V3, ... ,` in the context of a schema *K* = *F*(*T1*, *T2*, *T3*, ...).

[**Read**: Given a type T1 and one or more type-value pairs T2/V2, T3/V3, etc, select all values for T1 for the location defined by (T2 = V2) AND (T3 = V3) AND ...]

For example, given the types 'Color, ' and 'Object', and an appropriate schema definition and instances, the query

T1	*T2.V2*
Color	Object.ball

will measure and/or retrieve the Color value found at the location defined by the ball value of the object type.

A well formed command expression
is defined by `(T1, T2.V2, T3.V3) = V1 ,` in the context of a schema *K* = *F*(*T1*, *T2*, *T3*, ...).

[**Read**: Given a type T1 and one or more type-value pairs T2/V2, T3/V3, etc, assign the value V1 to the type T1 for the location defined by (T2 = V2) AND (T3 = V3) AND ...]

For example, given the types 'Color, ' and 'Object', and an appropriate schema definition and instances, the command

T1	*T2.V2*	*V1(T1)*
Color	Object.ball	Red

is to make the color of the ball red.

A well formed assertion expression
is defined by `(T1, T2.V2, T3.V3) == V1 ,` in the context of a schema *K* = *F*(*T1*, *T2*, *T3*, ...).

[**Read**: Given a type T1 and one or more type-value pairs T2/V2, T3/V3, etc, testably assert that the value of the type T1 for the location defined by (T2 = V2) AND (T3 = V3) AND ... is V1]

For example, given the types 'Color, ' and 'Object', and an appropriate schema definition and instances, the assertion

T1	*T2.V2*	*== V1(T1)*
Color	Object.ball	Red

states that measurable and/or retrieveable Color value found at the location defined by the ball value of the object type is red.

Examples of standard expressions

Fully qualified

WITH Type definitions: Sales with Units Dollars; Store with Units Categorical;
WITH Schema definition Store.* [1] – [1]+ Sales
Sales, Store.Cambridge = $500 ;

WITH Type definitions: Time with units DAY; Sales, Costs, Profit with units Dollars;
WITH Schema definition: Time.*[1]-[1]+(Sales, Costs, Profit)
Profit = Sales – Costs;

Value Expression Examples

The following examples show just the exchanged forms. All expressions are stated first in the abstract, followed by specific examples, and lastly by an interpretation.

A. *T1 = T2*;
Sales = Costs

Read: within some schema or context, the actual values for Sales as content for some location(s) are defined to be the [actual] values for Costs as contents for those same location(s).

B. *T1.U = T2*
Sales.Units = Dollars

Read: The Unit for the Type "Sales" is defined as the Type "Dollars". As such the potential values for the Type "Sales" are the same as the potential values for the Type "Dollars".

C. *T.V = T.V*
Geog.*V* = Geog.USA

Read: Assign the value "USA" to the Value *V*. In this example, both "USA" and *V* are of Type Geog.
However, when we begin to assign representations, if the representations match, the Types need not have compatible Units.

D. *T.V= F(T.V)*
Geog.USA = Geog.Mass.Up(1)

Read: The Geog Value "USA" is the parent of the Geog Value "Mass". Note that Mass here is in no danger of being misinterpreted as a subtype of Type Weight.

Examples of type and schema defining expressions

Schema-defining Expressions

Schema = (*T1* []–[] *T2* []–[] *T3*...) [1]-[1]+ (*T4* []–[] *T5* []–[] *T6*...)
simplified to:
Schema = ([]- *T1*, []–*T2* []-*T3*...) [1]-[1]+ (*T4* , *T5* , *T6*...)

```
Bob's Sales Model =
     ([1]+ Time, [1]+ Geog, [1]+ Product).* [1]-[1]+ (Sales,
Costs, Profit)
```

This is an example of a basic Schema definition[27]. The ordering relationship between the Types in the left hand side parentheses generates a set of unique Type Structure values called a Location Structure. The combination of [1]+ ordering relationships plus the ( ).* encompassing the Types specifies every unique "Time", "Geog" and "Product" tuple, (i.e., a cross product).

The Types in the right hand side parentheses form the Content Structure of the Schema.

All schemas have some Location Structure and some Content Structure.

So Bob's Sales Model can be read as:

"Every unique combination of Time, Geog and Product, with replacement is associated with a (not necessarily unique) triple of Sales, Costs, and Profit." The Types in the right hands side parentheses form the Content Structure of the Schema. The ".*" to the outside of the left hand side parentheses says to take all the values. The "[1]" ordering token on the outside of the Location Structure says that no two values of the location structure may repeat. The [1]+ token on the outside of the right-hand side parentheses says that the values for each of the Types in the right hand side parentheses may repeat.

[27] All the ordering relationships are positional. (The assumed default when relationships are not 1-N.)

For example, the ordering relationship `A.*[1]—[1]+B` reads:

For each unique value of A there is associated some valid value of B (recall that the plus sign indicates that the values for B may repeat between instances of the A-B relationship). Absent further specification it may be that every A has a unique B, or that every A has the same value of B or any combination in between.[28] This is the operational definition of the casual term "attribute of". As in "B is an attribute of A".

Keep in mind that even these basic relationships can have many variants following from subsetting the A's and B's, which subsetting was introduced above on scoping. Thus, one might state `A.G.*[1]-[1]+B` which reads for each unique value of some Named Group "G" of A there is associated some valid value of B. Or one might state `A.*[1]-[1]+B.G` which reads "for each unique value of A there is associated some valid value of some Named Group "G" of B."

Each value of the Schema defines as many assertions or queries as there are contents. In this example, there are three queries per value of the Schema. (As data is read in to the Schema the queries will change to assertions.) The combination of one value from one Content Type and one value from the entire Location Structure defines one assertion or query.

Schema1 = (*Schema2* []—[] *Schema3* []—[] *T1*) [1]-[1]+ (*T2*[]—[]*T3*...)

```
Bob's New Sales Model = (Time_dimension_table[1]+,
Product_dimension_table[1]+, Geog[1]+).* [1]-[1]+ (Sales,
Costs, Profit)
```

This is an example of a nested Schema definition. The management of Schemas in terms of Location and Content structures, provides the ability to create a Location Structure from the Location Structures of each of the Schemas on the right-hand side of the expression. In other words, the Cartesian Product is between the Time dimension in the Time dimension table and the Product dimension in the Product dimension table. The attributes from each of those Schemas remain uni-dimensional contents in the created left-hand side Schema. Thus, Bob's New Sales Model will have dimensions of time, Product and Geography. Sales, Costs and profit will each be dimensioned by all three dimensions.

All of the attributes (implicit in this example) such as holiday status for Time, or price for Product, remain attributes of their respective dimensions. And they will be queryable from within Bob's New Sales Model.

For example one might query for "`Sales, Product.price > $100`".

[28] This is like the relational case where A is the primary key and B is a non-key attribute.

TLH

Schema relating expression examples

There are many types of schema relating expressions. For instance, simple schema identities, relationships between schemas and instances, and expressions relating instances only.

A. Schema1.value = Schema2.value
   Sales.$100 , Time.january , Product.Shoes = Sales.$100 , Time.january , Product.Shoes

B. F(Schema1) = F(Schema2)
   Sales, Bob's Sales Model = Costs, Jane's Sales Model

C. *F1(F2, TS1) = F3(F4, TS2*
   Sales, (Time.January , Bob's Sales Model) = Costs , (Time.February, Janes Sales Model)*)*

Unit Expression Examples

A. *T1.U = F(T2*
   Time.Month = "Month-of-Year"

Read: The Unit called "Month" is defined in terms of the Type called "Month-of-Year"

B. *T1.U1.v1 = F(T2.U2.v2)*
   Product.Category.furniture = Product.item.chairs.LevelUP(1)

Read: The value one level up from the value "chairs" of the Unit "item" of the Type "Product" is equal to the Value "furniture" of the Unit "Category" of the Type "Product".

C. *T1.U1.i1 = F(T2.U2.i2)*
   Mass.Kilogram.1 = Mass.Pounds.2.2

Read: The Mass Values of the Unit "Kilograms" are equal to 2.2 Times the Mass Values of the Unit "Pounds"

D. *T1.U1 = F(T2.U2* and *T3.U3* and …)
   Speed.MPH = Distance.Miles / Time.Hour

Read: The Unit "MPH" of the Type "Speed" is equal to the Unit "Miles" of the Type "Distance" divided by the Unit "Hour" of the Type "Time".

E. *T1.U1 = (v1* xor *v2* xor *v3* xor *v4* xor *v5*…)
Color.primary = Red XOR Green XOR Blue

Read: The Potential Values of the Unit "Primary" of the Type "Color" are equal to the Values Red XOR Green XOR Blue.

F. *T1.U1 = (U2* AND *U3* AND *U4* AND…)
Time.DateStamp = (Day_of_month AND Month_of_year AND Year)

Read: The Unit "DateStamp" of the Type "Time" is equal to the Units Day_of_month AND Month_of_year AND Year.

G. *U1(T) = F(U2)*
Euro(Currency) = 1.15 * (Dollar)
Read: The Values of the Unit "Euro" of the Type "Currency" are each equal to 1.15 Times the value of the Unit "Dollars" of the Type "Currency.

U1(T) = F(U2)
Euro(Currency) ,Time.Jan0303 = 1.15 * (Dollar)
Read: The Values of the Unit "Euro" of the Type "Currency" on January 3rd 2003 are each equal to 1.15 times the value of the Unit "Dollars" of the Type "Currency" for that same date.

H. **Unit**(*T*) = *U1* XOR *U2* XOR *U3* XOR *U4*…
Unit(Mass) = Kg XOR Grams XOR Pounds XOR Ounces

Read: Units for the Type "Mass" are equal to Kg XOR Grams XOR Pounds XOR Ounces

I. *U1 = ((v1^v2^v3^v4^..)&(v5^v6^v7^v8^..)&(v9^v10^v11^v12^..)&..)*
Byte = ((1 XOR 0) AND(1 XOR 0) AND(1 XOR 0) AND(1 XOR 0) AND(1 XOR 0) AND(1 XOR 0) AND(1 XOR 0) AND(1 XOR 0))

Read: The potential Values for the Type "Byte" are composed of 8 binary XOR Values that are ANDed together.

Ordering Relationship Expression Examples

```
Geography = (Country.* [1]-[N] State.*);
```

Each instance of the ordering relationship between Countries and States is defined by one Country and N States. No country may appear in more than one instance of the ordering relationship. No State may appear in more than one instance of the ordering relationship. The number of instances of the ordering relationship equals the number of countries. Every country and every State is used once within the aggregate ordering relationship.

TLH

Every country is adjacent to N States. Every Country is connected to some State. Every State is connected to some Country. No State is connected to more than one Country. This may also be expressed as [1]-R-[N] instead of [1]-[N].

```
Process = (Step.* [1-N]–[1-N] Step!.*);
```

Each instance of the ordering relationship between the Steps is defined by one or more (up to N) Steps and one or more (up to N-1) steps which are drawn from the set of steps which does not include the step(s) on the left. No Step may appear in more than one instance of each side of the ordering relationship. Either side of the ordering relationship may have multiple occurrences of Steps (but will have at least one occurrence). The maximum number of instances of the ordering relationship equals N*(N-1). Every Step is used at least once. Every Step is connected to at least one other Step.

Emergent expression complexities

Content		**Location**		**Examples**
Type	Simple	Type	Simple	The table is green
	Simple		Structured	The table in my yard is green
	Structured		Simple	Dave used to be taller than Jill
	Structured		Structured	Last year in Asia, the cost for commodities was less than for services
Type		Expression		Dave is taller than Jill is true
Type		Schema		Turn ON the schema “Raiding the fridge”
Expression		Type		Asserting a function expression such as “Y = 10X” relative to Type such as Time as location
Expression		Expression		Asserting a function expression such as “Y = 10X” relative to an expression such as “Sales > 50”
Expression		Schema		
Schema		Type		
Schema		Expression		

TLH

Schema	Schema	

Atomic expressions

Multi-type locators

Multi-type contents

Multi-form expressions

Nested expressions

Molecular expressions

General States of Mind: Part IV of the LC Kernel

Introduction

So far the reader has seen types, schemas and expressions as three parts of the LC system for the foundations of abstract science. At one level, this is all the abstract machinery that is required. In other words, from here to the description of human level consciousness, the concepts of types, schemas and expressions are all that is required.

However, at another level, there is yet something missing - something without which any system would be inert. That something is a description of those specific schemas (and concomitant types and expressions), that are required for a symbolic expression management system to do anything intentional, and moreover, to do so in an ongoing, sustainable way. Think of it, metaphorically, as the logical equivalent of perpetual (cognitive), motion.

Roughly stated, therefore, part IV of the LC Kernel needs to account for

- Logical conditions under which purported symbolic expressions are known to be executable or have a truth value
- Trust or belief or confidence that some set of expressions is the case,
- Want or desire for particular expressions to be the case,
- Emotion or feeling that results from the difference between expressions that are wanted to be and expressions that are believed to be,
- Want or desire to affect the execution state of a command expression, and
- Any changes to any of the above functions

The basic flow, within the borders of meaningfulness, is the ongoing comparison between expressions that are wanted to be and expressions with the same LC form that are most strongly believed to be. This comparison generates a content difference (of assertion or emotion) which can be arbitrarily complex. The content difference then generates an emotional state and a want to execute some set of commands. It may also generate changes to the set of expressions that are wanted to be and/or changes in the belief-of-being for those same expressions. The successful or unsuccessful change in the execution state of commands potentially generates changes to what is believed to be which is then compared with whatever is wanted to be, and so the cycle repeats.

The critical reader needs to ask the question, "What are the minimal collections of processes or schemas that need to run within an expression management system such that the system may be said to act intentionally with respect to its environment?" This minimal collection in no way constitutes a human-level intelligence. Rather, no matter how simple the mind, no matter if the mind was composed only of a small number of simple types and schemas and possessed no more awareness than a simple insect, or an environmental regulation system on the space station, still it would need to have these

Comment [oc9]: Roughly stated, part IV of the LC Kernel needs to account for
•Logical conditions under which purported expressions are known to be executable or have a truth value
•Want or desire for particular assertions to be true,
•Trust or belief or confidence (and associated degrees thereof), that some set of assertions is true,
•Emotion or feeling that results from the difference between assertions that are wanted to be true and assertions that are believed to be true,
•Want or desire to affect the execution state of a command expression, and
•Any changes to any of the above functions

The basic flow within the borders of meaningfulness is the ongoing comparison between assertions that are wanted to be true and assertions with the same LC form that are most strongly believed to be true. This comparison generates a content difference which can be arbitrarily complex. The content difference then generates an emotional state and a want to execute some set of commands. It may also generate changes to the set of assertions that are wanted to be true and/or changes in the belief-of-truth for those same assertions. The successful or unsuccessful execution of commands generates changes to what is believed to be true which is then compared with whatever is wanted to be true and the cycle repeats.

processes in place, else it would not be able to act intentionally or initiate goal-oriented actions.

One might argue that scientists, especially abstract scientists need not care about intention. After all, science is about truth not values and wants. If something is true, it matters not whether one wants to believe it.

To this we would counter

- That notions of belief which are, without contest, critical to objective truth are a part of the network of schemas that creates intentional action. And,
- That "want" and "emotion" can only be stripped from scientific discourse after the fact. In the same way that mathematical discoveries, which take place through trial and error, intuition, analogy and a host of informal processes, are presented, somewhat disingenuously, in strict axiom-theorem style, other processes of scientific discovery take place within a framework of desire and intentional action. A scientist is passionate about a particular topic. Someone's thesis advisor told a candidate to research a particular topic. The best jobs are in a particular field of inquiry. A cute girl is studying geology.

Since the topics that will be addressed in this chapter are not canonically treated as inter-related subsystems required for any symbolic expression management system to work, we begin, as with other chapters, by building a conceptual bridge from canonical approaches to those of LC.

From canonical to LC

In general

The main topics treated interdependently within the LC system that have historically been treated as independent topics emanating from different disciplines –and in the wording of those disciplines- are

- The conditions for well-formedness, as defined in logic and linguistics,
- The conditions for an assertion to be true, as defined in logic and mathematics,
- "Trust" or confidence, or the reasons for believing that something is either the case or is likely to be the case, and if so, how likely, as found in statistics, and used in computer science
- "Want" as found in psychology, and "Utility" or "Pleasure" as found in economics and utilitarianism,
- "Emotion" as found in psychology, and
- "Execution state" as found in computer science

Interestingly, the concepts defined in all of the above topics are widely used in computer science. Database theory, especially Relational, makes extensive use of the concepts of well-formedness. Logicism, a particular school within Artificial Intelligence, makes use

of truth states within the context of theorem proving for choosing actions. Automated decision making programs such as credit approvals, decision analysis and text processing, make heavy use of probabilistic reasoning. Various schools of machine learning such as reinforcement learning make use of the concept of want to drive actions. And current attempts to create more human-like machines such as Sony's electronic pets have incorporated emotional concepts into the basic design.

More specifically

Canonically, the concept of meaningfulness and the border between it and meaninglessness is something that is dealt with in logic, linguistics, and natural language processing. Notions of well-formedness are pervasive in canonical logic.

In canonical thinking, however, the only kinds of expressions that are processed for well-formedness are assertions. In LC, the conditions for well-formedness are the same for all kinds of symbolic expressions: questions, assertions and commands

The closest concept to what is here called trust or belief or confidence or probability is that which is treated in statistical mathematics as probability theory. Probability theory deals with rational bases for holding certain beliefs. There are of course many schools and theories of probability: Bernouilli, Laplace, Gauss, Bayse, …….. Typically they focus on assigning relative likelihoods to future contingent events. Everything from decision theory and real options to risk management are based on notions of relative likelihood.

LC generalizes the notion of probability to include emotional bases for holding certain beliefs. LC also distinguishes the types whose values define different potential states of belief or trust and the functions by which those values are assigned in specific cases. This separation is very important as it allows an expression management system to use even the simplest type representation of degrees of belief such as "all or nothing" or "Max, medium, low" that enable the system to decide which assertions to believe and use for subsequent decision making.

"Want" (and its three extreme cases- craving, aversion, indifference) is not typically dealt with in any of the abstract sciences. One would need to look at Buddhist sutras, psychology or economics to see formal treatments of "want". The problem with the treatment of want in economics is that it associates the want or utility function with the individual rather than with the myriad context-specific schemas or behaviour patterns that operate within the individual. A single person may have a huge number of different and contradictory utility functions. Psychology recognizes the complexity of the individual and provides a few broad buckets such as conscious versus unconscious wants, but doesn't treat wants with sufficient formality. Marketing as an application of psychology is probably the most advanced as regards recognizing the existence of many different utility functions within a individual and the ability, through advertising, to evoke certain ones over others. Classical western philosophical approaches to want such as that of Epicurus are limited to conscious symbolic expressions of want. In LC, wants stretch

from the symbolic through the sub-symbolic all the way to the tactile or non-representational. They are closest in spirit and owe for their formation to, the Buddhist notion of "want"

"Emotion" is also not typically dealt with in any of the abstract sciences. Although from a pure "type" perspective, the potential values for "want" are a subset of the potential values for "emotion", emotional state plays a different functional role within the overall state of mind than does "degree of want".

Whereas "want" is the driver of action, "emotion" or "feeling" is what is subjectively experienced. In contrast with symbolic expressions of objective facts which are highly representational and which can never be quite certain owing to the inherent distance between the expression and the fact, feelings are non-representational and certain in the mind that is experiencing the feeling. The character of human feelings is a function of their physical representation. Human feelings are electrochemical.

Overview of the chapter

This chapter describes the general form of logical states, semantic states, belief states, want states, and execution states. Each of the states is treated as a schema or program. For example, the discussion of logical state will look at the variety of logical states that exist, how the logical state of an expression is evaluated and how information about logical state can be used.

The inter-relatedness of states of mind

Looked at as a whole, (and within the boundaries of meaningfulness), the action principle for expression management systems may be described in terms the system's attempt to maintain a particular state, or set of related states, of want, belief, emotion, and command execution. Each of these states-of-mind may be looked at as a higher order function in the sense that the arguments to the function are themselves expressions or combinations of expressions.

- The function "Logical state" takes a purported symbolic expression as its argument and returns the degree of well-formedness of the expression.
- The function "Belief" takes a well formed assertion and its associated schema as its arguments and returns a truth state.
- The function "Want" takes a well formed assertion and a truth state and returns a degree of want.
- The function "Emotion" takes a true belief and a true want (as in a want for an assertion to be true), for the same assertion and returns

- a difference or emotional state, and it returns
- a want to execute a command (the latter which is frequently called a decision) wherein commands that are wanted to be executed are attempted to be executed.

The success or failure of execution of a command is a fact-of-the-world which may or may not be sensed and symbolically represented. If it is, it generates a new round of belief to be compared with want, and generate emotion and want to execute some command.

Although it is customary for analyses of behaviour to begin with sensation, the real driver for action is emotion.

Logical States

Introduction

Although in conjunction with assumptions about shared context, well formedness constraints can be defined for expressions in their exchanged form, the well formedness of an expression more closely resembles a state machine or a decision tree than a simple binary condition. Furthermore, it often requires one or more attempts to execute the executable form of an expression to determine the degree to which the exchanged form is well formed.

The purpose of this section is to describe the various well formednesses or logical states that an expression may have as well as the decision criteria for assigning a logical state to an expression. This section is written with electronic information systems in mind. Humans do not typically process millions of purported expression of similar type where large chunks of the purported expressions are either meaningless (for any of a variety of reasons), or missing their content.

As with decisions of the well formedness of an expression, the truth-preserving response (from the expression management system), to the particular logical state of an expression is a function of additional assumptions and as such does not have a single necessary value. The responses described below are done so in the context of explicit assumptions about desired system response.

The type whose values describe logical states is special in the sense that it is used to create expressions whose locations (or arguments) are themselves expressions. Expressions of logical state are thus second order expressions. For example, a simple version of a "Logical state" type might have values that include 'meaningless', '[meaningful &]missing', '[meaningful &] present where the values in square brackets are implied. Its values may be asserted of or queried about any expressions as with

`“The expression, ‘The number of fish singing from blue is round.’ is meaningless”.`

Furthermore, the values of logical (or any other) states may be chained as any other kind of value. Thus one can assert
Logical state of “Bob’s car is green.” is assigned the value of the logical state of “Mars is a planet” See in this a prototype for Modus Ponens.

Although it is customary to treat the truth values ‘true’ and ‘false’ as a part of the logical state of an expression and to include true, false and some notion of meaningless within explorations of so-called multi-valued logics, the LC System keeps logical states of meaningless and meaningful distinct from semantic states such as true and false for two reasons:

1. An expression must be meaningful in order to posses a semantic value.
2. Commands as well as assertions may be evaluated for their logical state. Thus in LC all kinds of symbolic expressions possess and are evaluated for their logical state. For expressions that are evaluated as meaningful, only assertions are subsequently tested for their truth or falsity. Commands are further tested for the success or failure of their execution.

The problem

Purely syntactic approaches to well formedness
- Green ideas sleep furiously.
- This sentence is false
- The color of Bob’s head is < >.

In a world of templates[29]., failure to distinguish between missing and meaningless data can result in false aggregations.

[29] One of the problems with traditional OLAP engines is their inability to distinguish between null intersections that imply there is missing data and null intersections that specify that data is not applicable to that intersection. Further, traditional OLAP engines lack even the basic semantics to clearly state the problem. Using LC terms, the problem can be more clearly stated as the inability of traditional OLAP engines to distinguish locations for which contents are applicable from locations for which contents are inapplicable. **Failure to properly distinguish between missing and meaningless data can result in incorrect aggregates**.

Empty cells that occur within locations for which contents are applicable, designate missing data. Empty cells that occur within locations for which data is inapplicable designate meaningless or inapplicable data. Thus, for example, if one had a simple sales model where the variable or content “`Sales`” were dimensioned by “`Time`”, “`Stores`” and “`Product`” one might assert that Sales of winter coats was inapplicable to stores in Florida.

Thus, the application range of “`Sales`” within the model would be all locations except the combination of winter coats for stores in Florida[29]. This semantic constraint would then be used to interpret sparse

Employee Name	Name of spouse
Bob	Jane
Jill	
Dave	

LC system of logical state management

Whereas one can differentiate specification of WFF for exchanged versus executable forms, the process of measuring well formedness transcends these boundaries. To measure or evaluate the well formedness or logical state of an expression in exchanged form may require trying to parse, compile and execute some or all of the expression. This is why there is only one set of logical states.

Logical state expressions define the possible logical states that a Content or Predicate can have relative to a Location or subject or argument. For example, "being meaningful and present" is one possible state that the Content "Sales" may have relative to the Location "January-Snow Suits". Being meaningless is another.

There are two different ways by which the logical state of a proposition or atomic expression may be determined: user specification, and system discovery. The distinction is important because user specification and system discovery may not yield the same result. For example, a user may have specified that a certain content, say Sales, was meaningful (or applicable) relative to a particular store, say Paris. Yet, consider the processing of a piece of business logic, such as

IF
Count (Region "x" Stores whose Sales increased this year over last year)
/
Count (Region "x" Stores whose Sales did not increase this year over last year)
> 2
THEN
AWARD Bonus to Region "x" Store Manager

---

intersections found in the data. For all locations other than winter coats for stores in Florida sparse cells would be interpreted as missing data. While for locations defined in terms of winter coats for stores in Florida, sparse cells would be interpreted as meaningless. The distinction between missing and meaningless cells is then leveraged for all aggregate functions – meaningless cells need to be excluded from calculations while missing cells need to be assigned proxies**.**

A related problem is where the semantics of a data source are under-defined. This frequently happens with SQL databases when null tokens are involved. In these cases, it is not uncommon to find a null token as the value of a column. The problem is that the meaning of the null token is often left unspecified. It can mean either missing or meaningless.

What is supposed to happen if no record is found for the Paris store? (Not simply a missing Sales figure in a row whose Store value is Paris, but no row for Paris.)

The system wants to test for each store identified in the schema whether Sales increased between last year and this year. If the system can not find a Paris id in the data source, any statements about Paris (not just those about Sales) are meaningless. (Clearly the user/developer should be alerted. But that is a separate issue.)

When a user defines a schema, the default is that the Contents are defined as applicable or meaningful relative to every location in the Location Structure. When the logical state of Contents relative to the location structure is not the same for all Contents, the user can explicitly declare the logical state of any Content to be any state.

Some concrete syntax

LOGICAL STATE (Type AS Content , Types AS Location) = *Logical State*

The function LOGICAL STATE may assume the values (Not Applicable or (Possibly Applicable OR Under Specified OR (Applicable AND (Missing OR (Present AND (Valid OR Invalid))))))

For example:

LOGICAL STATE (Sales , Time.january , Geography.cambridge) = Not Applicable

LOGICAL STATE (Sales , Time.january , Geography.cambridge) = Possibly Applicable

LOGICAL STATE (Sales , Time.january , Geography.cambridge) = Under Specified

LOGICAL STATE (Sales , Time.january , Geography.cambridge) = Applicable

LOGICAL STATE (Sales , Time.january , Geography.cambridge) = Missing

LOGICAL STATE (Sales , Time.january , Geography.cambridge) = Present

LOGICAL STATE (Sales , Time.january , Geography.cambridge) = Invalid

LOGICAL STATE (Sales , Time.january , Geography.cambridge) = Valid

In addition, this section describes a decision tree by which those states may be determined by the system during the course of attempting to parse, compile and execute the proposition (as contained in an expression).

Although there are several ways that a proposition may be deemed meaningless, the rules for logical inference are the same for all meaningless propositions. Assuming the existence of an LC Schema based on well-formed Types, the following table outlines

TLH

- How the system determines the logical status of a query-defining proposition and
- How the system should respond to its determination of the logical state of the expression in an attempt to return as much information as possible

TLH

Given a symbolic expression in exchanged form:

Result of trying to process the expression	Well constructed or not	Logical State	Possible System Response	Possible treatment of expression
1. The tokens do not correspond to any known Type names or values	Not a well constructed expression (Parser error)	Not Applicable (Meaningless)	"Unrecognized tokens"	Drop
2. The tokens correspond to known Type names and values but the combination of Types referred to in the assertion do not correspond to or define any known Schema	Not a well constructed expression (Compiler error)	Not Applicable (Meaningless)	"Unrecognized schema"	Drop
3. The tokens correspond to Type names in a Schema but they all lack values.	Not a well constructed expression (Compiler error)	Not Applicable (Meaningless)	No location defined	Specify Location or Drop
4. The tokens correspond to Types used as Locators and Contents but the specified Content was previously defined as "Not Applicable" to the specified Location.	Not a well constructed expression (Compiler error)	Not Applicable (Meaningless)	Content is not applicable to Location	Substitute applicable Content or drop
5. The tokens define a well constructed query but in attempting to execute the query no matching location can be found in the data	Yes a well constructed expression	Possibly Applicable (run time discovery)	Can not find the requested locations	Redefine location or drop
6. More than one matching location is found	Yes a well constructed expression	Under-Specified; A special case of Possibly Applicable.	Found multiple locations	Scope the Locations, aggregate the content across the locations or

		(run time discovery)		drop
7. A matching location is found but more than one content is found at that location	Yes a well constructed expression	Under-Specified; A special case of Possibly Applicable. (run time discovery)	Found multiple Contents at the location	Scope the Contents, aggregate the contents at the location, or drop
8. A matching location can be found but no contents are found	Yes a well constructed expression	Applicable and Missing. (run time discovery)	Content is missing	Assign a proxy and evaluate the logical expression
9. A matching location can be found and a single content is found but it is illegitimate	Yes a well constructed expression	Applicable, Present AND Invalid; Logically equivalent to Missing. (run time discovery)	Content is invalid	Assign a proxy and evaluate the logical expression
10. A matching location can be found and a single content is found that is legitimate	Yes a well constructed expression	Applicable, Present AND Valid	Content is valid	Evaluate the Logical expression

If the result of trying to process a well constructed query is outcome 5, 6 or 7, the query is possibly applicable (i.e., potentially meaningful). But it is meaningless from a system perspective absent additional processing logic.

For example, regarding outcome 5, there might be a statement that says if a row for a Store (AS Locator) is not found, alert the administrator, or add a row and call its contents missing. However, if there were a business rule that needed to execute at that moment, queries whose locations can not be found are treated by the query processing system as meaningless.

Likewise, a schema could exist that allowed for each employee to have multiple phone numbers. However, if a business rule included the Boolean clause "employee phone number = 234-5678" the result of trying to process that rule relative to an employee who had two or more phone numbers (absent additional logic), would generate the logical state of "Under Specified" for the query. This is because, if one were to treat a location with multiple contents as a valid proposition, it would be possible to discover that a single proposition was both true and false. When the query processor discovers

propositions with logical states 6 or 7, the default response at run time is to alert the user to an under-specified query.

If the result of trying to process a well constructed query is outcome 8, the system uses whatever proxy logic has been established to gap-fill these kinds of missing values.

If the result of trying to process a well constructed query is outcome 9, the system treats the invalid value as a missing value and uses whatever proxy logic has been established to gap-fill these kinds of missing values.

If the result of trying to process a well constructed query is outcome 10, the system can directly apply whatever business logic exists. All Logical expressions (or Booleans), applied to Applicable, Present Valid propositions will return True or False.

Although it is tempting to want to proceed directly to semantic states from logical ones, there is no way to describe the process by which a system tests the truth value of an assertion or the execution state of a command without passing through belief states. Only when there exists a clear hierarchy of degrees of belief can one even speak meaningfully of truth testing.

Belief states

Whether biological or computer, every expression management system that provides even a simple form of multi-modal interpreted awareness (see chapter 'x' below), needs some ability to measure, analyze, and manage beliefs or trust. There are two main reasons for this.

1. Different independent sensors may indicate the presence of different objects while an impending action for the entity may be a function of the object's believed identity. Without some system of belief (probability, certainty or trust) management that allows the entity to choose one interpretation over another, the system would deadlock. Random selection will produce a decision, but is not likely to pick the best interpretation if there is one.
2. With or without potential conflict between sensors, any finite system must trust (in some bounded sense of the term), some sensations or it will expend all of its energy testing its sensors, and creating and testing sensor testers, and sensor tester testers, till it runs out or resources. It would never complete a single assertion[30]. This is why, at any level of expression management, some expressions must be taken for granted (or remain beyond doubt or be left unchallenged), so that other expressions may be asserted, queried or executed.

[add some history of statistics : probability theory, the idea of "natural propensities" versus subjective beliefs of propensities. Distinguish "idealized worlds such as card games which are closer to abstract than concrete structures and which give way to

[30] One might call this Zeon's certainty paradox

abstracted descriptions of relative likelihoods. Also talk about 17th and 18th century use of term believability.

Strictly speaking, probability should be reserved for abstract, definitional structures whose relative likelihoods are definitionally defined.

Recognized complexities

Emotional versus rational bases

- The degree of belief or certainty that is assigned to expressions may be derived from either a rational process or an emotional process. A person may believe a particular expression or trust a particular source of information because it has been thoroughly tested and/or because the individual wants to believe the expression or trust the source.
- Emotional bonds determine the trust for many expressions in a significant percentage of humans.
    - On average, most people want to believe in those expressions whose doubting would cause them to rethink other deeply held beliefs, especially those that concern the moral nature of the individual and/or his/her larger sense of self.
        - For example, in a recent poll it was discovered that persons who supported the war in Iraq were significantly more likely to believe that Saddam Hussein was responsible for the World Trade Center destruction than persons who were against the war, even though both sets of persons had access to the same widely disseminated fact that he had nothing to do with it.
        - For another example, mothers are less likely to believe that their children committed horrible crimes than strangers on a jury.
- As significant as emotions are for the determination of beliefs, this section focuses on rational bases for assigning, analyzing and managing trust.

Rational empirical versus rational theoretical bases

- Most expressions, especially expressions of scientific laws, can be tested empirically and/or theoretically. Whereas one person may be convinced because of a concrete measurement; another person may only be convinced by an abstract proof of consistency, whether positive or negative.
    - In particle physics, for example, theorists predicted the existence and mass of Z particles long before any measurements were established that could test the theoretical predictions.
    - In economics, a prediction that significant changes in the value of the dollar are going to have an equally significant impact on employment in the U.S. may only be believed by an econometrician if there are adequate measurements to back up the claim. A theoretical monetarist might not believe the statement because it is inconsistent with her or his monetarist economic model.

- By what measurable attributes might one assign varying degrees of confidence to empirically versus theoretically based expressions of the same content? When theory and practice do not agree what are the conditions for deciding which to believe?

Measurement certainty versus inferential certainty

- Is there a correct way to aggregate individual measurement beliefs? Are there constraints on valid ways?
    - If the uncertainty associated with some measurement is +/- 15% and that associated with another measurement is +/- 20%, what is the uncertainty associated with their quotient?
- How do inferences impact the uncertainty of the underlying measurements? How should inferences be compared with measurements?
- What are the major kinds of inferences? How are degrees of belief assigned to them?

Deductive versus inductive inferential certainty

- Is the certainty assigned to abstract definitional beliefs such as expressions of mathematical equivalence unqualified or qualified? Are these expressions really "absolutely certain" or perhaps only "system-maximally certain"?
- How does definitional certainty compare with measurement certainty?
- Are all theoretically-based assertions equally certain and why or why not?

The objects of belief

- To what kinds of objects are degrees of belief assigned?
    - If expressions, is it all expressions?
        - If not, to what subset are degrees of belief assigned?
    - Is it schemas or types within schemas, or types in a central catalog?

Belief types

In simple systems, all assertions may be believed to be true. There may not need to be any softwired belief encoding. In more complex systems, it is necessary to softwire belief states. For example possible functions might be:

- If value comes from sense organ, probability = 1
- If value comes from authority figure, probability = 1

In significantly complex systems, not only are beliefs softwired, but the type that represents their potential values may have numerous values, such as degrees, and moreover may be associated with complex functions whose output is the specification of a degree of belief.

Statistics, regressions and least squares

TLH

We speak of finding the coefficients and we speak of X and Y as unknown quantities. Better to speak of X and Y as Type-units assumed constant over the given observations and that one is looking for the assumed constant instances alpha and beta which are best thought of as defining a constant delta: delta X /delta Y.

For some series of price and sales data,

Error = K + alpha price + beta sales

Price and Sales are types of some unit: price in dollars and sales in number/quantity or price in euros and sales in dollars or price in "normalized value " and sales in "normalized value" ....

Alpha and beta are the instances (of some unit and type) 1000 (dollars of type price) 500 (quantity of type sales)

Believability

Purpose:

1) facilitate selection of and/or merging of different assertions when they conflict/contradict each other and only one value (whether selected or merged) can be used

This implies that it is necessary to be able to calculate the relative believabilities of different assertions (whether input or derived) wherein those different believabilities may be either

- clearly in favor of some one assertion or some one merged value
- clearly unable to distinguish the different degrees of believability
- clearly unclear about the relative believabilities

All calculations of believability need to include some measure/inference of their own confidence

2) corollary Select to either act on an assertion if believability is above some threshold or treat an assertion of insufficient believability -even in the presence of no competing assertion- as insufficiently believable which may translate into hunting for additional information to increase the believability to the point where action may be taken.

This implies that different kinds of actions may require different degrees of believability of the decision-driving assertions. Typically the higher the stakes, the more believable needs be the assertion. And the less time to act, the lower needs to be the degree of believability. These two constraints may oppose each other.

TLH

Basic setup

B(Sense data) = 1

B(Abstract comparative equality) = 1

B(Everything else) < 1

On average, B(analog expressions) > B(concrete symbolic expressions)

New observations impact current believabilities of inferences that include current observations as a function of the ratio of new observations to old where old are what formed the inference and new are what occurred since the inference

When does new observation imply that the prior assumed to be homogeneous environment is better thought of as two

For any Content of any location

What is its believability?

Are there any other C's vying for the same location?

Do they agree?

If yes, with what believability?

If no, what is most believable?

You can increase the believability of an assertion by reducing its precision. The two – believability and precision are inversely correlated in this sense.

It may not be very believable that a hot pro team would beat a highschool team in baseball by the exact score 43-0. But it would be very believable to assert that they would beat them by at least 5 runs.

TLH

Overcoming the complexities

1. Need to assume that the specification process can be repeated, that the object specified/measured is constant across specifications/measurements, and that errors are randomly distributed

> Measure and record the length of the side of a table 25 times; measure and record the time it takes for a ball to drop from a known height 25 times; copy a number from one database table into another 25 times. Assign the value 17 to an integer variable 25 times…

When this is the case, one can assume that the "true" measurement is equivalent to the mean (which equals the median), of the distribution of the measurements. And the std deviation of the measurement estimates is an indicator of the measurement precisi0on.

Believability in this case is the inverse of the average wiggle or std deviation.

1.1 Additionally one may assume that there exists an independent secondary specification process of known/assumed believability higher than the primary process such that the secondary process is used to test the primary process.

> Test the ruler measurement of table length with a laser scan; (or let teacher test student; test the stop watch time it takes a ball to drop with a laser clock ; test the number copying with a number compare routine or by human eye; test the assignment process with a number compare program or a human eye..

2. Then one can speak of the likelihood of specification error based on some predefined level of precision for any individual specification and of the expected value of a single error.

> The likelihood of ruler measurement error might be 99.99% for a tolerance of 1/64 of an inch, 50% for a tolerance of 1/8 of an inch, 10% for a tolerance of ¼ of an inch and 0.01% for a tolerance of ½ of an inch. The likelihood of copying error might be 0.001% for a tolerance of 0 (the copied string is either right or wrong)

There is an interplay between the level of granularity at which a "measurable object or thing" is considered to be constant and the level of granularity for which measurement may be said to be in error.

TLH

If the length of an object randomly varies by $1/64^{th}$ of an inch every ¼ second or so, there is a natural lower limit to the object's measurability precision. Conceivably most or all measurements of the object could be within $1/64^{th}$ of an inch and thus could all be without error.

That said, the canonical assumption is that a thing is "infinitely" exact in its being and thus any wiggle in the measurement is a sign of lack of measurement precision.

In numeric measurement, the chances of error are typically assumed to be one, or near one, so the expected value of a single error is the same as the aggregate precision of the measurement.

In contrast, for categorical specification, values are either 100% right or 100% wrong. Most values are 100% right. And when errors occur they are always 100% wrong. So for non numeric specifications, likelihood and value of error are different.

The combination of the likelihood of error and the likely value of error define the believability of the value

3. When the value specified is numeric, one can also speak of the randomness or symmetry of the error and of the expected value of the aggregate error

4. Believability applies to initial specifications be they measurements and/or definitions

5. Believability also applies to all inferred specifications: deduced and/or induced: descriptions, explanations, predictions, goal-oriented decision recommendations

6. Believabilities can be rationally compared across all specifications that originate from measured sources XOR defined sources.

- The relative believabilities of measured and defined sources can not be rationally compared. (Though they can be emotionally compared.)
- The relative believabilities of inductive and /or deductive measurement inferences can be rationally compared with those of source measurements
- The relative believabilities of inductive and/or deductive definitional inferences can be rationally compared with those of source definitions

7. Believabilities can be combined across specifications

Inter error cumulative values may cancel or accumulate across specifications

TLH

Intra error quantities (the sizes of individual errors) may increase, remain stable or decrease across specifications

The likelihood of individual specification error increases non-monotonically with increases in the number of specifications

B(Categorical variable) Avg prob.true and prob.false

B(Numeric assignment) Avg likelihood of true and Likelihood of false

B(Numeric measurement) Avg likelihood of true (prob X 1) and Likelihood false X Value AS one minus error as percent of value

B(numeric products or quotients) = product of believabilities

B(sum or diff of numeric measures) = avg of believabilities

B(prediction) = B(deductive inferences) X B(measurements)) X B(Forward assumptions)

Thus all inferred specifications

6. In general one can speak of the content of any location as the value assigned by the output of some process Fc applied to some source X.

- Is Source X a "data generating event"?

TLH

- Is Fc a visual sensation or audio recognition or tactile pattern?
- Or is Fc just some deductive translation and X source some representation of the event?

Belief states are defined at the schema level on a specific content by specific location basis in terms of the density and precision of assertion processes (for those located content tuples) and at the schema instance level in terms of the relative likelihood of the particular instance relative to the aggregate of instances. The believability of an instance is a function of the relative believability of the schema-based Content assertion process and the relative likelihood or believability of the instance as determined by the schema-based collection of distribution-defining instances.

Belief states apply to all contents whether abstract or concrete, measured or inferred, and if inferred whether deducted or inducted, and whether first, second or so-called higher order expressions.

Belief states are used to select and/or blend different assertions about the same content for the same location.

It is easiest to think of belief states from a demand rather than a supply perspective. Belief states are only required to resolve conflicting assertions. Thus, the complexity of the functions used to assign belief states should be driven by the complexity of the assertions whose relative believability needs to be evaluated.

The hypothetical two poles are measurements and definitions whose relative believabilities are inherently incomparable, analogous to incommensurate metrics.

For example, if one measured a table to be 2 meters long and someone else defined it as 3 meters long, there would be no way to compare the relative believabilities of the two assertions. (In fact, it couldn't even be the same table.) If one were to suggest "seeing and finding out" that would be a way to test the believability of the measurement statement. If someone said to test whether in fact the table was defined to be 3 meters long, that would test the definition but not the measurement.

There is no way to rationally compare the believability of the definition with the believability of the measurement. To take a stand and declare one more believable than the other is to side with the definition or the measurement as the source of believability.

Emotions, of course can play a role. And the purpose to which the assertions are going to be used should be the guide .

TLH

Begin with a measurement.

We can speak of the measurement process and its believability, the measurement relative to a distribution and its believability and a measurement value and its believability. All believabilities are normal values between zero and one and which support inverses.

The believability of the measurement process is the inverse of the likelihood of an error occurring during any one measurement process. If the believability of the measurement process is 0.9999 that says the likelihood of an error occurring in a measurement is one in ten thousand.

The believability of the measurement relative to a distribution is the relative frequency of the measurement in the distribution.

The believability of the measurement is the decision to accept or reject the measurement value based on the comparison of the process and distribution believabilities.

If the believability of the measurement process is greater than the inverse of the believability of the measurement relative to its distribution, then the measurement value is accepted with the believability of the measurement process.

If the believability of the measurement process is less than the inverse of the believability of the measurement relative to its distribution, then the measurement value is rejected with the believability of the measurement relative to its distribution.

If the two believabilities are equal, the measurement is uncertain.

The believability of Inferences made from measurements are comparable with the believability of any other inferences made from measurements and any other measurements.

B(Sales measurement process) ~~ B(prior Sales measurements + deduction = distribution > relative likelihoods

B(sales measurement) ~~ B(Profit measurement – cost measurement)

B(Sales measurement this period) ~~ B(Sales prediction made from historical sales measurements and inductions last period)

B(Sales measurement) ~~ B(Delta Sales relationship based on measurements and deductions between periods made this period)

B(Sales measurement) ~~B(Sales prediction made from last period data on foottraffic and transactions and inductions)

Inferences made from definitions are comparable with any other inferences and definitions.

But again, the measurement or definitional nature of the atomic assertions proscribes what any molecular assertion believabilities may be comparable.

Focus here on inferences made from measurements

Believability of atomic measurement; changes to believability as a function of inference and as a function of additionally combined measures.

Each can be extended through any combination of deductive or inductive inference.

Regarding schema inferences

Begin with measurement schema. Every content is measured relative to its location. Now add

- Content relationships per location: Sales – Costs = Profit
- Location relationships per content: Sales this year versus last year
- LC relationships per LC: Foot traffic last week divided by number of transactions this week predicts profitability next week.

Given a schema as R(t,t,t) belief function is another collection of Rs (of value and process) such that R.process AND Rvalue(R(t.L, t.C))

TLH

For every content asserted of any location within any schema one can evaluate and compare the believability of the asserted value and the believability of the assertion process.

Definition of believability

Believability is a function of the precision and density of the content relative to the schema.

For most assertions there is both believability of the assertion value and believability of the assertion process.

It is useful to distinguish believability of measurements, believability of deductive inferences and believability of inductive inferences. Both deductive and inductive inferences can be further broken down into descriptions, explanations and predictions.

Assigning precision and density

Test(Assertion) >> probability

Test (Assertion about the world)

Empirical

Performing content evaluation and comparing
Recalling previous assertion and comparing
Based on attributes of assertion
    Utterer (professor)
    Location of utterance (textbook)
    Form of utterance (deep tone)
Definitional

Test (Assertion about a type)

Test (Assertion about a schema)

Test(Assertion about an expression)

TLH

Comparing believabilities

Combining believabilities

Semantic states

Given a set of well formed Types such as Object Name (including "house" as a value) and Color (including "Blue" as a value), and given a partially open Type structure

`Object name .* ~ Color (` read "for each unique value of object name, some valid value of color is located")

And
Given a well constructed assertion such as "`Color of house is blue`",

And
Given a sole data source `AS Source Of Truth` against which the assertion can be tested,

The assertion is well constructed and

True if the data source contains a single location called house with a single associated color value called blue ,

False if the data source contains a single location called house with a single associated color value called any valid color other than blue ,

To say that some located content or proposition is true is to assert the value "True" of its truth value which is to assert that it matches or corresponds to some other located content treated as a source of truth or with a higher degree of confidence. Absent such a confidence asymmetry between sources/Type structures, all that can be asserted is whether the two propositions do or do not match.

Want States

The roots of want

TLH

Perpetual want: System can always select some command for execution;
what is the rest state of the entity?

Want enters at both the exchanged symbolic level with want for an expression to be true

Want (truth state, expression)

I want it to be a nice day
I want to have nice clothes
]I want to have a roof over my head
I want kids
I want no kids

And at the executable form of an expression

Want (Execution state(Command))

I want to begin running
I want to be able to determine the weather to be nice
I want to be able to buy nice clothes

All action is purposeful. Not all purposes are mutually consistent. All systems need some want management to ensure a sufficient amount of entity consistency.

Though any expression could be the object of a want, not all expressions need have associated wants.

(Expression)-Want state

Some forward and backward chaining to transform current state into wanted state

Some measurement/awareness of difference between current state and want state

Want (execution state (Command))

Wants may be hardwired non representational

Wants may be uninterpreted representational

Wants may be interpreted representational

Wants may be observed and manipulated

TLH

Emotional States

The awareness of difference between want and belief is feeling and is perceived without representation

Feelings have the medium of the physical representation of the expression: chemical, electric,....

Feelings are the subjective reality whose external representation is only a pointer. Feelings are akin to internal tactile or touch.

Contrast with expressions of fact whose source of "truth" is the external shared context.

As with objective phenomena feelings can be accurately or inaccurately described. We may show or not show or show an altered version of our "true" feelings

Feelings may be hardwired into the sensory system. That is to say an entity may be hardwired with feeling sensors so the feeling is not the calculated difference between an actual state and a want state but is a directly sensed differential. Think pain receptors.

There can be as many patterns of feelings as there are patterns of facts. Our vocabularies cover only a small fraction of conceivable feelings.

Biologically we have developed a symbolic language to express feelings that is at least as rich as our symbolic language for fact sharing.

Crying, laughing, wincing, fear, anger, hatred, puzzlement, curiosity, ...

An entity can want a want state and act so as to achieve it.

Wants begin with touch....Buddhist meditation equanimity....

Anger, love,

Execution states

TLH

- Presumed executable and unexecuted , Presumed executable and executing, executable and executed, Unexecutable

Testing the LC System

There are two major ways to test any purported foundation for abstract science: external, observational inductive testing and internal, definitional consistency testing.

Consider gravity: Inductively, we can measure any falling body event starting from a known distance from a larger body of known mass. Every falling body event can be measured to see if it is accurately described by the gravity theory. The description would take the form of a testable prediction. Given gravity as a coefficient of acceleration, one can use that coefficient to predict how long it will take a body to fall what distance. These predictions are then compared with actual data.

Given the goals of the LC system, namely to serve as a foundation for abstract science, any expression that one might sense, interpret, or create whether analog or symbolic could be used for the purposes of empirical or inductive testing.

Tests may take the form of predictions that are tested against experiments as with the following:

- If an exchanged expression has only one type and a normal shared schema it will generate neither an assertion, question or command in the mind of the receiver.
- If a type has mutually inclusive potential values it can lead to contradiction
- If a type has disconnected values, not all values can be reached by calling its atomic operators
- If a schema has no locators it can not define questions, assertions or commands

Inductive tests

Is there any aspect of mathematics, logic or language that is not explainable in terms of the LC system? Although this is essentially an inductive question and so it is not possible to provide an exhaustive answer, nevertheless, one can look at a representative sample of math, logic and language. In each area the question is, "Are there any constructs in any of these existent expression systems that can not be exactly mimicked or duplicated in LC?".

For mathematics a representative sample might include

TLH

1. Arithmetic
2. Algebra
3. Sets
4. Calculus
5. Tensors
6. Geometry
7. Groups

For logic a representative sample might include

1. Propositional logic
2. Predicate calculus
3. Modal logic
4. Temporal logic
5. Mereologic

For language it might include

Natural

Empirical
1. English language structure

2. Chinese language structure

Theoretical
3. Chomskian deep linguistic structures

Software

1. The C language

2. The Lisp language

3. The SQL language

4. The MDX language

Internal consistency tests

Tests of internal consistency are trickier than empirical tests of completeness. This is because internal tests rely on some pre-agreed notion of math and logic. The consistency that one is looking to test is based on some prior specification of consistency per some theory of math and logic.

As such, it would be advisable to rely on a minimalist notion of consistency, namely self-sameness and its inverse: contradiction. Thus, any aspect of the LC system may be considered internally inconsistent if it can be shown to lead to a contradiction. The main tests of internal consistency may be defined in terms of the following questions.

- Are there any implications of the general form of a type that lead to some demonstrable contradiction?

- Are there any implications of the general form of a schema that lead to some demonstrable contradiction?

- Are there any implications based on the definition of a well formed expression in either exchanged or executable form that lead to some demonstrable contradiction?

- Are there any implications based on the general states of mind that lead to some demonstrable contradiction?

TLH

Section III Applications

.

Part one builds on what was presented in section II and creates what is intended to be a single foundation for the whole of the abstract sciences. It demonstrates its validity three ways:

- by defining propositions each of which is generally characteristic of an abstract domain such as logic and math and then shows how each of these characteristic propositions is grounded in LC Types and schemas,
- by taking a variety of propositions for empirical demonstration,
- by mimicking popular axiomatic bases such as Peano's axioms and showing what they leave out and furthermore showing how an LC representation of popular axiomatic bases for different abstract sciences naturally merge

In part two it leverages what was presented in section II to define a series of increasingly sophisticated cognitive processes beginning with a process of the simplicity of an amoebae and proceeding in simple steps until a cognitive process of the complexity of a human is reached. At the human level of complexity, a further breakdown occurs between innate and acquired structures and algorithms.

Part One: Providing a common foundation for math, logic and common language

Critique of the canonical foundations of abstract science

Introduction

Recognize each abstract "discipline" as a contingent fact. It is not necessarily written anywhere that there needs to be separate disciplines called math and logic. If we see them as bodies of activities each with a "center of natural kinds" as in there is not some definitional center but rather a collection of attributes typically common to each, then we can talk about approximate foundations wherein foundations refers to assertions that are almost always, if not always adhered to by each "discipline", or perhaps always within a sub-discipline..

This is not to say that there is not some abstract kernel of relationships that are a necessary part (in the sense of being required for the functioning), of any expression

management system and thus which any expression management system during the course of its existence when engaged in the pursuit of non-contingent or un-contextualized or necessary truths is, if properly functioning, bound to eventually discover, rather that said kernel does not cleanly fit within any of the current categories of abstract science, notably mathematics and logic.

Then we can map these approximate foundations into an LC core while providing a critique of these traditional beliefs at the same time. The deepest foundations of each ":discipline" will map to the inner layer. Other concepts will map to the LC outer layer.

For example in math: inner concepts include equality and inequality, WFF, assignment, series; outer concepts include the rationals and reals specifically the notion of precision or extension in both a pos and a res direction. Substantial criticism will be made of traditional definitions of number, series, infinity, the reals..

In logic: inner concepts include WFF, implication, AND/OR, quantification; outer concepts include modal, temporal, part-whole;

In natural language: inner concepts include sentence structure and parts of speech; outer concepts include part whole relationships

Layout LC inner and outer core and postulate how it underlies all disciplines – abstract and concrete

The specific claim is that any discipline must make use of all and only LC inner core concepts. **If these concepts are not present in either an expression management or cognitive system, that system will not work or function.**

Keeping in mind that embarking on a new domain of inquiry does not require any explicit formulation of core concepts but only tacit adherence to them, **no discipline needs to have an explicit foundations in order to work or be useful.**

The quest for foundations is fundamentally an introspective reflection.

The benefits, as stated above, are in the increased communication and cross fertilization of the sciences.

For two disciplines to determine their relative overlap/disjunct they each need to :

1. identify the types they use and for each type a full definition of potential values, ordering relationships, associated atomic functions and a derivation mapping back to known basic type families
2. identify all schemas used in the discipline
3. identify all axiomatic expressions
4. identify all relevant non-axiomatic expressions (may be 100s of thousands)

Wherever the two disciplines share the same types and schemas, equivalent locations need to be associated wit the same values of the same contents

The testable expression of a new foundation

Assert a goal for any expression management system such as

Assert, query, negate, affirm, assign, compare, w/o contradiction, excluding the meaningless, (calculate, derive, deduce, infer such that truth value of outputs = truth value of inputs)

Ignore for the moment the notion of specialized expression management systems

Assert any expression management system is composed of

some number of Types
some number of schemas composed of types
some number of expressions consistent with one or more schemas

For fixed type systems, the growth of schemas and expressions is limited only by physical resources and the limits of the existing types

For variable type systems, the growth of schemas and expressions is limited only by physical resources and the limits of creatable types.

Further assert that each Type may have

any number of potential values,
any kind of ordering
any number of units,
any declared funcs decomposable into atomic funcs

So long as:

each type
2 or more potential values,
unique values

TLH

traversable values..
is either a root type or derives from one or more root types

- Example root types include:
- binary cardinal
- binary categorical
- N-ary categorical
- N-ary cardinal
- Cyclical
- Multi-ordinal

Further assert that schemas
composed of any combination of types

Example schemas include

Distinguish schemas that measure (and derive) values and schemas that assume (and derive) values
Distinguish schemas that represent sensory-motor devices, schemas that measure from and write to specific sensory-motor schemas, schemas that measure from and write to schemas that measure from and write to specific sensory-motor schemas, and schemas that assume from and assume to arbitrary sensory-motor schemas.

So long as:

both L and C roles are accounted for
all Ls unique

Further assert that expressions
composed of any number of types
any C asserted/queried of any L

distinguish axiomatic expressions and non-axiomatic expressions

So long as

union of Cs associated with union of Ls

- Math and logic optimized for calculation
- Common language optimized for communication

Description of goals:

TLH

- goals of foundations of mathematics
  - Success testing
- goals of foundations of logic

Describe relevance of foundations of logic, to the rest of logic, to software databases and to politics (how to get the masses excited about meaningless propositions);

  - Success testing
- goals of unified foundations
  - Success testing

Type generation procedure
Type system

Perhaps be clear about the Types implicit within the generation procedures and make those explicit

What expressions , consistent with Types and schemas can be stated that are true by definition and that provide all the capabilities in the union of math (at least arithmetic) and logic

Seem to need the basic Type system and at least two substantiated types, boolean and pos/res natural.

Basic type system provides for WFF, logical states, Truth, substitutional equivalency, conceptual distinction between finite and infinite

Suggest some alternative wordings

Foundations of abstract science > provides the ability to define any kind of internally consistent (or inconsistent) abstract game (includes structures, procedures: constructive, substitutional, truth testing..)

Defines constraints for the ground rules of specific sciences (arithmetic)

Hence, ground rules of arithmetic

Testing foundations

Comment [ET10]: Page: 122
Can we proceed straight to axioms of propositional logic or do we first need to define some types?

In one sense, logic is more primitive in that it only requires boolean types; however, no system could exist without some cardinal types and as such arithmetic is also basic

Are the expression of axioms equivalent to the definition of the appropriate types such as natural or integer?

What can be said of only boolean types

T.v = =T.v

Creating cardinal types

Specific ordering definitions and associated atomic functions is the axiomatic (to use that term) expression of the foundations of arithmetic

TLH

If you can show me something that functions as a logical type that is not implicitly and/or explicitly composed of constructs, ordering relationships and operators/functions then the LC model of Types is wrong

If you can show me something that functions as a logical type that does not have at least two potential values or does not have connected values then the LC model is wrong.

If you can show me something that functions like a schema that is not composed of types then the LC model is wrong

Critique of the canonical foundations of logic

The purpose of this chapter is to explicitly map topics/axioms typically considered to be a part of the foundations of logic back to inner and outer core LC

Part one

Identify main topics/issues relative to which we typically look at foundations of logic:

Classic

- Operational definition of a well formed formula: syntactic or functional
- Definition/recognition of the components of a proposition: by label or by function relative to a proposed/contextualizing Type system in use
- Status, list and relationship of logical connectives: True of "the world", or of "language" " list: Negation, Or" vs (true of internal language system, set of binary distinctions all coexistent in any proposition)
    - Implication and certainty (maintenance of certainty of implications –inputs always assumed)
- Tautologies/contradictions
- Truth: by Tarskian correspondance versus LC equivalence across propositions of different degrees of belief where one is "presumed true". Also distinguish matching between internal propositions and internal against surface sense data.
- Criteria for recognizing ill-formed "purported" propositions
- Quantification over particular arguments/individuals
- Quantification over collections of individuals/arguments
- Quantification over all collections where the current proposition is a member of one of the collections (self reference)
- Criteria for distinguishing O.K. from problematic self-reference
- Implicit or explicit typing

TLH

New
- Modal issues
- temporal issues
- belief issues
- semantic issues
- hierarchies

Part two

Identify the main acknowledged problems (paradox as symptoms) with canonical logic:

- pseudo well formed formulas and truth values
- illegitimate totalities
- self reference

Comparisons with traditional foundations of logic

failure to recognize tree of meaningless logical states

failure to recognize functional relationship between locator and content

failure to recognize implication as assignment of logical states

failure to recognize modality as a separate “expression expression or expression management subsystem

failure to recognize mereology as stemming from type definitions

- Point out all areas of agreement between LC and LW
- Show how the LC Model avoids falling into the canonical traps and is free of known paradox

Propose a minimal tweak of canonical logic (interpretations and extensions) as solution of least change

TLH

Critique of the canonical foundations of mathematics

The purpose of this chapter is to explicitly map topics/axioms typically considered to be a part of the foundations of mathematics back to inner and outer core LC

A. Identify the main topics and position the main schools (logicist, formalist, intuitionist) relative to these topics

- Source of truth and Ontological status
- Calculation versus deduction
- Inputs assumed or measured
- Finite versus infinite

  Are rationals compact? What does it mean to say that that a rational point has no immediate neighbors?

- The continuum

Is motion continuous? At successively smaller intervals is object always in motion? If time is made of infinitely small instants and space of infinitely small points then is there motion within a point instant?

Distinguish perceptual continuity from existential continuity

  - the logical syntax of finite versus infinite classes
- Rational vs irrational, transcendental
- Numbers defined in terms of : sets/classes, properties, operations

- Acceptable forms of proof
    - induction
    - substitution
- The number tree

B. Analyze and critique the various positions

Identify at least the main paradoxes

- The continuum
- Cantor's diagonal proof

Comparisons with traditional foundations of mathematics

Major confusions in canonical thinking

1. Thinking in terms of points absent some specification of position and resolution. Not realizing that all specification is relative to some unit of some relative granularity. All so-called compact series represent unbounded collections of instances per unit and units per instance. At any unit there is an unbounded positional series. For every instance of some unit there is an unbounded series of units.

2. Thinking that any specification can be made in terms of infinitely small units or infinitely large iterations.

3. Not realizing that so-called irrationals and transcendentals are really two dimensional specifications

4. Not realizing that points are variably (1 – N) dimensional

5. Debating about continuity of motion when the real issue is that of existence which, if self-sameness is to be preserved needs to be assumed to be continuous.

6. Not realizing that numbers are iterations of a unit. Numbers are produced via a rule for iteration. They are not simply a property of classes. As such the whole notion of

iteration is lost. We recognize the attempt to bring so-called infinite quantities under the same definitional umbrella as finite quantities. But the attempt is misguided.

First, the essential characteristic of numbers is the rule-based iteration of a unit of constant size. Second, so-called infinite quantities are definitionally self-contradictory once they are used in any calculus.

Insert: every major use of infinite quantities

1. infinitesimals used in calculus

2. attempts to differentiate the sizes of different infinite sets: Aleph sub zero (the rationals), sub one (the reals) then power sets someplace.

3. Tokens for irrational quantities

4. Convergent series

5. The infinity of points between any two points on the rational number line

6.

show how it provides a deeper, broader and more consistent foundation for arithmetic

Key differences with canonical:
- role of unit,
- ordering relations within Types define adjacencies > atomic functions

- Naturals : positional iteration over some unit
- Rationals: positional and resolutional iteration over some unit
- So-called irrationals and transcendentals: incommensurate units

and for the concepts of finite/infinite, continuum, proof, number

Type constraints: failure to recognize units

TLH

failure to connect finite values and so-called infinite values within a single type

failure to take a stand on measurement

Constructive critique of rational and real number systems

Open questions:

1. So-called pure systems of quantity

I can phrase the question or specify the expression that generates a non-terminating, non-repeating outcome in terms that do not appear to require two dimensions. As such, I can use the most general definition of the rationals to define notions like the square root of 2 (but not the circumference of a circle).

Continuing along this line it would even appear that I can use the rationals to specify distance in both a euclidian straight line and a polar rotational line. What's to stop me from using the rationals to specify distance along the circumference. Pick any unboundedly large "N" as the number of steps to make a full revolution. One can still identify iterations and steps sizes and ask what combination of step size and number of iterations produces exactly two steps along the circumference.

So if this is the case, is the diagonal explanation of Sqroot2 as an illegal move, which feels correct, just an illustrative heuristic?

Or is it not really possible to frame the problems of square roots in a one dimensional application of the rationals?

> My pick. This is because even something so benign as finding the exact base and iteration involves in the executable form of the expression two different temp variables locked in a simultaneous equation, i.e., an equation where X AND Y are interdependent.
>
> Corollary: one can define the rationals in a so-called pure or one dimensional sense where they are either open or closed. But in this case, these pure rationals are only closed under the set of one dimensional distance operations (specified combinations of addition, subtraction, multiplication and division). Any expression whose execution requires an ANDed relationship between X and Y may implicitly reference a non-euclidian unit as one of the items being compared and thus not be exactly resolvable.

Or is it not really possible to define the rationals independent of their being open or closed? And if so, is there something about the rationals that implies or relies on a two dimensional notion of straightness

2. Units

In days past this was called metrics. The notion, still adhered to, was that all instances, words, numbers, actual values.. depend on some notion of metric, unit, type or potential value in order to have any meaning.

One of the motivators for this approach was to circumvent the problems of so-called irrational and transcendental numbers.

Rather than looking for irrationals in "gaps" in the rational number line, i.e., as missing quantities, irrationals were to be explained in terms of incommensurate metrics or units or type definitions definable in terms of comparison expressions that result in non-terminating processes. Thus, for example, one would describe pi as the ratio of the circumference of a circle measured in terms of a closed or periodic or rotational type/unit/metric and the length of the diameter of that same circle measured in terms of a straight/Euclidian type/metric/unit which length is defined as the quantity "1". There are several ways to explain the incommensurateness of the two units. Euclidian provides for straight motion in an XY grid delta x without delta y and vice versa. Polar provides for just the opposite. Relative to a 2 type system of an x and a y type every move involves an equal delta in x and y. There are a limitless number of inbetween combinations where some delta occurs for X and Y, just not the same amount.

In general one can say there are two extreme forms (type definitions, unit definitions, metric definitions) of multi-type system:

1. delta X XOR delta Y

2. Delta X AND delta Y

The mutual exclusivity of XOR and AND which goes back to definition of any Type explains why, on average, distances defined in one system can not be exactly resolved into distances defined in terms of the other.

What is thereby irrational is not a quantity but a comparison. And two uses of a gap-less system of rational quantities, where the different uses are based on incommensurate units/metrics/topologies may produce irresolvable

ratios/comparisons in the form of expressions whose executable forms specify non-terminating non-repeating processes.

2. Base type units

In LC, units are a role of a type in defining the potential values and operations of a type. What's crucial is that

- Every instance be associated with a definition (whether that definition is called a type definition-where types are simple constructs-, a unit definition where types may have multiple units, a metric definition or whatever), that specifies the potential values and operations –including inter-instance traversal- for that instance.
- There only exists one system of specification or differentiation. This means that units and types must be reducible to the same thing. And there can not exist a type system as distinct from a unit system.

Typically, since the overwhelming majority of created types are non-root types, the non-root type is created in terms of some previously existent type treated as a unit. Dollars may be the previously existent type treated as unit for Sales. Integer may be the previously existent type treated as a unit for dollars. So where do the units of a type come from when the type is a root type? When a type is a root type, this means that all of its operational characteristics including its potential values and operations either type-wide or on a per unit-per-type basis are explicitly defined as a part of the type definition. As such, every instance in the type is still associated with a unit. Simply that unit was explicitly constructed along with the rest of the type.

At their most abstract, the notion of whole, integer and rational numbers take place independent of open/closed (euclidian/polar). Whole and integer are unilevel; rational are multi level (have ORed units)

Processes

We can speak of non-terminating non-repeating processes in a pos direction and/or a res direction. But so too is unbounded addition or multiplication in the presence of a RAND operator. The point is that at any point in the execution of the expression it is possible to specify what the calculated quantity is at that point in the execution process. Adding RAND(1) indefinitely is a non-repeating, non-terminating process. That doesn't call into question whether the rational are complete. Likewise, the process of looking for an identical iteration and base quantity is a well specified process that may or may terminate depending on the initial quantity whose square root is being sought. At any point in the

execution process the trial quantity is either smaller or larger than the input number resulting in either the removal or addition of a term at the same or next smaller level of granularity (defined by increasing denominator).

It is thus important to distinguish between quantities and processes. The fact that a well specified process does not terminate does not mean that the type that defines the possible instances is ill-defined, under-defined or in any way missing values. The fact that there is no exact value for the square root of two does not mean there are any gaps in the rational number system.

Regarding units, what are the units for fully constructed types? What are the units for the integers or wholes? Is it necessary to have them derived from ever more abstract types? No. It is not even possible to do. Since the basic properties of the number system vary as a function of the attributes listed above.

It is now best to replace the term "unit" by its definition to understand how it applies to fully constructed or root types. The notion of unit defines the possible values and valid operations that apply to any instance of a type. A unit is therefore the common or underlying or permanent set of possibilities or the potential relative to which the instance is an actual.

For most defined types, such as "Sales", "Number of employees", "Day-of-the-month" it is easy to see the types from which theya re defined supply this essential information. Sales may be defined in terms of "dollars", "number of employees" may be defined in terms of whole numbers, etc..

So what are whole numbers defined in terms of? They are defined in terms of their own fully constructed specification. This is not circular. It would only be circular if there were no fully constructed specification given to the type whose units are defined in terms of itself.

Thus for example, if the wholes are

Unilevel, simple, unbounded, include zero, sequential, one neighbor each direction, constant interval, open, acyclic

And if they support iteration, concatenation and difference,

An expression of a quantity of whole numbers may be expressed as a function of itself.

"Some whole number iteration" of "some whole number unit"

Likewise: a categorical iteration of some categorical unit. We can specify that the set or totality of expressed categorical values is distinct without needing to be able to sequences them.

By tradition, the whole number unit is the "one" instance of the whole number type, though it need not be.

Thus, every whole number value may be looked at as the iteration of some whole number instance of the specific whole number instance "one".

All expressions assume that that unit is constant and equal to "one"

For example" 2 + 3 = 5" or " 3 + 7 = 11" could have all been restated as

2(1) + 3(1) = 5(1)

Frege sometimes called the logical unit a concept.

Most of what we call Euclidian is based on at least two types in a Cartesian product such as

(X.* 1+ Y.*1+).* 1-1+ Point
Delta X.* 1-0delta Y.*

Delta Y.* 1-0 Delta X.*

Easy to show how so called square root of two violates this. The path across the diagonal is along both delta X and delta Y.

Easy to show pi violates this. Path along the circumference violate the no delta X and Y rule

Again, it's not the quantity that is missing from Pi or sqr root of 2, rather the path is not a valid Euclidian path. What we always called incommensurate metrics or units.

TLH

Thought experiments: unbounded types where not known if open or closed Imagine an unbounded collection of instances and discover it is closed. The larger the number of instances, the more the local surface of the "circle" in the neighborhood of any point resembles a straight line.

Critique of the canonical foundations of natural/common language

- main difference from math/logic: not interested in implication , slowly changing words, phrases and meanings, embedded physics communication efficiency over calculation efficiency
- Main sameness: criterion for meaningfulness
  - subjects and predicates
- slang
- Evolution
- Nouns verbs and other parts of speech

---

[i] The terms operator and function frequently used interchangeably. One can distinguish the expression as function from the component as either operator or function.

[ii] Though it might be tempting to try to describe that unity, any attempt would need to make use of the very primitive concepts being defined, namely constructs, ordering relationships and functions and thus would be incapable of describing that unity in anything other than those terms (or functionally equivalent terms). It is better to think of the primitive concepts and the understanding of their interdependencies as the limits of what can be expressed in language about those concepts. Thus we can postulate the existence of some underlying complex from which the primitive concepts of "Construct", "Ordering Relationship" and "Function" are projections, but there is nothing we can say about it. (maybe insert LW quote.. "Whereof we can not speak thereof we must be silent") It is easier to understand the inherent unity of primitive concepts from an expression execution perspective. All expressions require some construct, ordering relationship and function sub expressions. Only full expressions can be executed.

[iii] It is common, however, for values (which contain implicit references to their units), to be described along with their units effectively double stating the units. Thus, for example, it is common to come across statements like "Tuesday is a value of the unit day", or "USA is a value of the unit country". So as to facilitate its comprehension and use, the LC Model permits such double encoding.

[iv] Furthermore, units are treated as scalars and can not be combined within the Type or Class

[v] Of course we can treat the same physical structure in more than one way. For example, we can treat a one Byte Char as a Categorical structure of 8 ANDed Binary XORs. Or we can treat that same one Byte structure as 0-256 numeric iterations of a single bit.

[vi] Here again whether Tuesday and Wednesday are instances or values depends on the scope of their name space. If the token "Tuesday" uniquely identifies an instance of unit day then, as a single token it serves as a value. If the token "Tuesday" however could be used in different ways then, absent mention of the unit "Day" it is just an instance.

[vii] The comments regarding ordering relationships in the OO world also apply to the entity-relationship (ER) modeling world (i.e., conceptual and logical database design). Most such ER modeling languages have about the same level of richness (or impoverishment, take you choice) regarding support for ordering relationships, as does OO modeling. Most ER languages support the notions of 1-1, 1-M, and M-M relationships between entities, and these concepts get mapped either to primary/foreign key definitions or, in the case of M-M relationships, new tables which are defined specifically to realize those relationships, as part of the process of refining a logical database design (an ER model) into a physical database design (table layout). However, the level of support for ordering in general, and intra-entity ordering in particular, is on a par with standard practice in OO. Recommend that we include mention of this here

[viii] This capability is described in section 'x' on joins.

[ix] This is an essential capability for communication between people. People do not learn about the world in the same order. One person's primitives may be another person's derivatives and vice versa. When two (or more) persons communicate about a complex domain, typically they each accumulated the background information required to communicate about the domain in different orders and furthermore do not all share exactly the same background. If the topic is international finance, for example, one person may have first learned about finance then about the intricacies of international finance. Another person may have lived in multiple countries and studied international relations before learning about finance. Although these persons can communicate about international finance,

(i.e., define equivalency relationships between them), a Type model of their respective knowledgebases would reveal significantly different structures being used to represent the same external international finance phenomena.

[x] For example, one might want to specify the number of potential values.

```
Count (V.*) = X
```

Anyone defining a non-root Type that used this Categorical Type as its unit would need to at least think about the number of potential values. Since every physical representation of a Type limits the number of potential values. The act of representing it as a CHAR[8] versus a CHAR[64] limits that number of potential values. There is no need to make this specification at the logical level. Clearly the specification of the number of values constrains the internal representation to provide for at least that amount of differentiation.

[xi]

Value Editing Syntax

There are two styles of logically equivalent value editing syntax that we have worked on at different times: 1) database-like insertion and deletion and 2) equation-like equivalency expressions. Either or both could be implemented in Bailey. The database-like is presented first.

Database-like syntax

This syntax includes the ability to explicitly insert values before or after other values, which assumes a particular sequence ordering for the internal and external representation. We have not yet covered the specification of sequence definition, but there is in fact always some ordering of values. If the tokens `Before` or `After` are not used and if the Type has no logical ordering (i.e. it is neither numeric nor rank), the insertion is made at the end of the Type or unit within the type. For every defined ordering on the type where the ordering relies on a comparison function (possibly implicit for numeric or rank), values will be inserted in the position specified by the comparison function regardless of the presence of `Before` or `After`.

```
Insert VALUE [, VALUE...] INTO TYPE_NAME [ . SCOPE_NAME ]
      [ Before | After VALUE ];
Insert TYPE_NAME.VALUE INTO  TYPE_NAME
      [Before | After < VALUE >];
Delete TYPE_NAME.< VALUE >;
```

where

- `VALUE` refers to individually named new values to be added to `TYPE_NAME`.
- the `VALUE` following the Before or After keyword refers to a single value of the type and is the insertion point for the new values. The default is After.
- VALUE in all uses except following the Before and After keyword refers to one or more values of the named type.

For example:

```
Insert CA, MA, NY Into Geography.State ;
Insert LA, Boston, Cambridge, NYC Into Geography.City AFTER San
Jose ;
```

Equation-like syntax

The '+=' and '-=' operators can be used to incrementally add and remove instances while leaving all other instances intact.

```
<TYPE_NAME>.<SCOPING> += <{VALUE LIST}>

< TYPE_NAME >.<SCOPING> += <TYPE_NAME>.<SCOPING>

< TYPE_NAME >.<SCOPING> += <FORMULA>
```

and

```
< TYPE_NAME >.< SCOPING > -= <{VALUE LIST}>

< TYPE_NAME >.< SCOPING > -= < TYPE_NAME >.<{VALUE LIST}>
```



```
< TYPE_NAME >.< SCOPING > -= <FORMULA>
```

Examples:

```
Geog.City += {Houston, Dallas, Texarkana}

MyProd += function that returns some products

Prod.SKU -= {02, 1004, 3829}

Prod.SKU -= {Prod.Toys.Pokemon.Children}
```

[xii] Chapter five text refernce