Hypergraph Categories and Conceptual Space Theory

Nathaniel Christen

2019/09/16

In its first incarnation, Conceptual Space Theory was rooted in linguistics, or secondarily perhaps in philosophy — insofar as any "theory of concepts" should be of philosophical interest. This theory's year zero was 2000, via Peter Gärdenfors's *Conceptual Spaces: The Geometry of Thought* ([**?**]). Soon, though, the theory migrated from linguistics to realms of science and technology: as a tool for modeling the evolution of scientific theories ([**?**], [**?**]), for describing scientific/technical data ([**?**], [**?**]), for bridging cognitive and computational linguistics ([**?**]), and in general for bringing a more nuanced understanding of cognition into computational settings ([**?**]). Advocates for Conceptual Space Theory saw Gärdenfors's perspective as more faithful to human language and conceptualization, more humanistically accurate, than reductive "mind as computer" metaphors that often dominate research at the boundaries between cognitive and computer science — Artificial Intelligence, certainly, but also Knowledge Engineering, Data Modeling, and technological fields where representing human concepts become imporant. Concepts become *technological* subject matter when we have to build computational platforms — software, databases — to classify facts and documents for human's interactive use, in a manner sensitive to human concepts and their influence on how humans will understand (and therefore want to interact with) an information-space, such as a collection of research paper, or scientific research data.

Renewed interest in Conceptual Space Theory is evinced by recent projects integrating this theory with Hypergraph Categories ([**?**], [**?**]), with the core notion of merging a Hypergraph-based grammar with a Conceptual Space semantics. Bob Cocke's 2015 paper (*et. al.*, [**?**]) is the most direct statement of this new paradigm: essentially a strategy for co-theorizing syntax and semantics in a strongly formal, mathematically oriented spirit but also receptive to Conceptual Space Theory's cognitive and linguistic nuance. That is, the proposed Hypergraph/Conceptual Space hybrid can orient both formal research (in fields like computer science and formal language theory) and more informal (or at least not logicomathematically formal) methodologies in linguistics and general "theories of meaning".

I believe the Hypergraph/Conceptual Space hybrid (which I'll nickname "**HCS**") is an important breakthrough, especially because it follows analogous research unifying theories of cognition/conceptualization with computational and technological endeavors (at a practical level, not only theoretical). The Semantic Web, for example — grounded in graph-based representations of technical data — embodies an applied technological project but also the influence of philosophy and cognitive science, particularly through the notion of *Ontologies*, or formal models of some domain of information or conceptualization, used to specify the morhology (i.e., possible morphologie) of data structures which carry information relevant to the corresponding domains. Although the technical restructuring of data — for communication between different computing environments — is a mostly bare-bones algorithmic problem, Ontologies also capture how formal systems have to model scientific concepts and norms that structure technical data in an overarching, gestalt fashion: notions like space, time, causality, points vs. regions, punctuality vs. perdurance, different spatial relations (inside, around, connected to, etc.), and so forth. Formulating logical models for these foundational conceptual building-blocks became essential for comprehensive models of scientific and technical domains, insofar as technical data is to be digitally archived and shared, so that the representation of human "phenomenological" concepts within structured, digital information systems becomes a pertinent question.

So largely speculative and theoretical philosophical territory like Ontology and Phenomenology started to become, particularly in this century, an origin for technical, digital-engineering-style research (see e.g. [**?**], [**?**], etc.) Certainly only a minority of programmers constructing, say, the Semantic Web were actively engaged with these more philosophical foundations, but the collision between speculative traditions and contemporary digital applications did engender a certain interdisciplinary circle. This milieu, in turn, overlapped with Cognitive Linguistics; as Gärdenfors puts it: .

Meanwhile, researchers like Jean Petitot and Barry Smith were trying to absorb Husserlian Phenomenology into a computational and technological environment, partly through data-modeling Ontologies (though also via analyses of mathematical/computational models of vision and perception, such as in the realm of 3D computer graphics, supplying approximate sketches of human perception — e.g. in Petitot's works like [?], [?] and in the broad literature on Mereotopoloy: ...). As Maxwell James Ramstead puts it in a 2015 master's thesis:

> Now, the "science of salience" proposed by Petitot and Smith (1997) illustrates the kind of formalized analysis made possible through the direct mathematization of phenomenological descriptions. Its aim is to account for the invariant descriptive structures of lived experience (what Husserl called "essences") through formalization, providing a descriptive geometry of macroscopic phenomena, a "morphological eidetics" of the disclosure of objects in conscious experience (in Husserl's words, the "constitution" of objects). Petitot employs differential geometry and morphodynamics to model phenomenal experience, and Smith uses formal structures from mereotopology (the theory of parts, wholes, and their boundaries) to a similar effect. [?, p. 38]

In principle, the Semantic Web and its accompanying technologies (like **RDF** — the Resource Description Format — and **OWL** — Web Ontology Language) were the primary vehicle for operationalizing this "phenomenological" background. In practical terms, Barry Smith, for example, spearheaded applied projects like the **OBO** (Open Biomedical Ontology) foundry and the National Center for Ontological Research (**NCOR**). Meanwhile, though, other scholars in the cognitive-linguistic tradition — not least Gärdenfors himself — objected to Semantic Web paradigms being oversimplistic, or too far removed from the patterns and priorities of actual human reasoning (see [?]). Applications like Conceptual Space Markup Language were, to some degree, explicitly conceived as alternatives to Semantic Web formats like **RDF** and **OWL**.

The "**HCS**" model adds a wrinkle to this intellectual history, some 15 years after debates like the implicit contrast between Gärdenfors and Barry Smith arose. Writings of Barry Smith or Jean Petitot reveal how technological applications can be one venue for extending or consolidating philosophical frameworks: in the admittedly artificial and not-fully-human digital ecosystem we can still see a revealing trace of philosophical intuitions, the analytic reach of their assumptions, the potential for isolating within accounts of *human* meaning and reason certain gestalts where are intrinsic to *meaning* or *reason* as such, even without human subtlety and sociality/situatedness/embodiment. The Semantic Web is retroactively interesting to philosophy, as if Gärdenfors-style critiques thereof.

With the emergence of the Hypergraph-based **HCS** model, which to some degree presents its own contrast to the (graph-based) Semantic Web, there is another iteration of Conceptual Space approaches that potentially address critiques that "the Semantic Web is not very semantic" (Gärdenfors) or "The Semantic Web Needs More Cognition" (Martin Raubal and Benjamin Adams). Against the precedent that Semantic Web theory (or its alternatives) retroactively shed light on philosophical, cognitive, and phenomenological issues, we can see the Hypergraph-Conceptual Space hybrid as a new technological model that has its own ramifications for philosophy, and generally outside the narrow mathematical/computer-science fields. In short, a comprehensive, multi-disciplinary actualization of the **HCS** would become an intellectual construct stretching between highly formal theories such as Hypergraph Categories and more humanistic contexts such as Cognitive Linguistics and Cognitive Phenomenology.

With that said, I propose to develop the **HCS** model in a manner that departs somewhat from both the Hypergraph analysis (on the syntactic side) and Gärdenfors's original Conceptual Space theory (on the semantic side). My own formal Hypergraph model will be oriented more toward software engineering and applied type theory than toward pure mathematics — while automated proofs can sometimes bring strong guarantees about the correctness of some software component, in the more typical applications we need well-structured data models to guide the implementation of software and data-sharing platforms. As a result, formal models may be more valuable insofar as they facilitate computational integration, rather than drive mathematical theorems or analyses (e.g., via Category Theory). I will clarify notions like "integration" and "implementation", as I see them, below. On this basis, I will both add some structure to the original **HCS** framework and also present it in a more casual, less mathematical fashion.

Meanwhile, on the semantic side of things, I will attend specifically to the constrast between applications of Conceptual Space theory to *formal* (or technical/technological) semantics or, respectively, to (natural) linguistics. Computational applications of Gärdenfors's models are often motivated a sense that Gärdenfors has isolated conceptual structures underpinning semantics that transcend the distinction between "human" (or natural-language) and artificial semantics. This trope seems to reappear in numerous papers (many I cited earlier) adapting Conceptual Space Theory to computational settings, as if Gärdenfors has unmasked a well-behaved core amidst the seeming chaos of human concepts, on which basis we can more confidently move toward a paradigm of computers understanding language on our own terms.

In this area I find the applications of Conceptual Spaces still a little too reductionistic. I think there certainly *are* both formal/technological and philosophical/linguistic applications of Conceptual Space theory, which I will try to summarize here, but as I see it the theory blossoms differently in the two contexts. It is interesting to speculate on what structures or paradigms tie together the two branches, and what implications that might have, but in this paper I will largely treat Conceptual Space models as branching into two technically distinct concretizations depending on whether we are analyzing natural language or specifying codifications for data models and formal semantics. I will start by reviewing the landscape from the Natural Language perspective before developing my own continuation of the underlying Conceptual Space theory — particularly via a specific account of applied type theory — later in the paper.

1 Conceptual Spaces and Cognitive Grammar

In its original form, Conceptual Space Theory suggested that concepts tend to be interrelated with other concepts, and that the dimensions of their contrasts — how concepts acquire relatively fixed meanings by playing off other (potentially similar) concepts — can often be analyzed quantitatively. Concept's individual meanings may be established in the context of larger conceptual spaces, where the sense that we attribute to each concept is determined in part by the pattern or "boundary" of its differences against other concepts that are, in some sense, its peers. We distinguish *cabin*, *house*, and *mansion*, say, not only by comparative size, but by judgments that a certain building is a mansion more than a house; that it falls across some threshold where the one concept seems more applicable than the other.

The point here is not to reductively trasform concepts into purely numeric measures; instead, Gärdenfors argues that the qualitative and quantitative are mutually determinative. We can identify various qualitative aspects through which the objects of our experience acquire definitive content (supporting how we conceptualize them), but this rational-perceptual process is in turn supported by our ability to recognize a quantitative dynamic brought forward through the qualitative essence. In the case of color, for example, we can theorize various phenomena in the qualitative givenness of color; how it appears as part of experiential reality: the fact that colors are qualitatively dependent on light, that they usually appear as infusing extended regions of space (usually as the surface of material things), that they are "bound" to things as their properties — we say an apple *is* red, for example. These basic conceptual and phenomenological details are also linguistic, insofar as they structure talk about color and (individual) colors. But, according to Conceptual Space Theory, we cannot fully account for this qualitative dimension without also theoretically introducing the quantitative patterns that also determine our conceptualization of color in general: the grading of hues from light to dark, the sense of many shades being blends (in various proportions) of primary colors (reddish-purple to purple to purplish-blue, and so forth), and the tangible sense of colors being nearer or further from a handful of canonical shades (an orange almost-yellow, a brown almost-red, etc.). Gärdenfors argues that these phenomena are simultaneously linguistic (insofar as they inform color lexicons), psychological (influenced by how we see color), and mathematical (because color-spaces — such as the **HSV**[1] double-cone — formally describe the dimensions of variation and distance which become manifest in the language and psychology of colors.[2]

Gärdenfors's treatment is particularly compelling in case-studies such as color (and analogously sound, scent, and taste) which can be approached scientifically (and psychologically, mathematically) and also reflect basic components of raw experience, part of the pre-linguistic infrastructure of consciousness. Directly applying the same theoretical framework to most everyday concepts, however — which usually come more from the practical engagements of consciousness than from its phenomenal substratum — is more difficult. Taken as a kind of phenomenological maxim (although this phenomenological resonance is not especially highlighted in Conceptual Space research proper), the co-establishment of quantitative and qualitative facticity is well-taken. We should indeed be sensitive to how quantitative dimensions (of contrast and, potentially, of phenomenal appearance) are intrinsic to qualitative aspects of reality being experienced by us as bases for conceptualization and predication — for the world existing not just as a play of appearances but as a scientifically ordered, propositionally stable surroundings in which we reason, act, and believe. The problem is that the marriage of quality and quantity, while certainly meaningful

[1] Hue, Saturation, Value

[2] For instance, the double-cone space in three dimensional, so there are multiple axes on which colors can differ from (and be experienced as differing from) one another.

for consciousness and concepts, is rarely sufficient for demarcating concepts. Most real-world concepts, I would argue, arise from situational, goal-directed activity and the roles we assign to objects (and their concepts) in each enactive/pragmatic scenario.

What ultimately stabilizes concepts like *house*, *restaurant*, or *knife*, for instance, is not so much a quantifiable comparison with other concepts but rather certain prototypical pragmas pertaining to our actions — often in a social setting. We visit people in their house, for instance; dine in a restaurant; use knives as dining utencils. The conceptual root of *restaurant*, say, derives from the needs of our rational faculties to navigate the situations where we tend to engage with restaurants. Usually that means dining in them, of course (though there are other possibilities, such as noting that a previously unused building has become a restaurant). Then the restaurant concept organizes a "script" of other concepts and social norms, such as finding (or being assigned) a table, studying the menu, communicating with a server, ordering dishes, paying the bill, and so forth. The script is somewhat different in a restaurant as compared to a *cafeteria*, or a *stall* (in a food court), or a *coffee bar*. So the *restaurant* concept does play against certain peers, but the contrast is operational more than quantitative — even if there are certain quantitative patterns that we might find in this "space of concepts" (a restaurant is usually larger than a stall, pricier than a coffee bar or cafeteria, and so forth).

Analogously, we can find reasonable quantitative bounds on the concept *knife*: the size (and relative sizes) of the blade and handle help establish distinctions such as *knife* rather than *sword* or *dagger*. Moreover, knives qualitatively present as hard, sharp, and usually metal (though plastic knives are accepted as versions of smallish knives, e.g. butter knives, for takeout food). But these quantitative differences are significant mostly because they are products of knives' functional role — we distinguish *dagger* because daggers are not used in a culinary environment. The boundaries of the knife concept are mostly situational — butter knives are deemed knives even if they are not sharp, because the act of spreading butter (or jam, etc.) on something like a piece of bread is close to the act of cutting through food than of "spearing" pieces of food (as with a fork) or holding a liquid (as with a spoon). We can spread jam on bread with a spoon also, but the situation of taking a pat of butter and applying it over the surface is, evidently, conceptually closer to the common role of knives (working with solids) than of spoons, so the butter knife ends up affixed to or subsumed under the knife concept.

Here I am using terminology of conceptual "distance" and differentiation, so I borrow some of the infrastructure of Conceptual Spaces, but I do so against the backdrop of considerations oriented towards concepts' origins in practical activity. To quantify the contrast of *knife* and *spoon*, say, the important dimension is not so much the flat geometry of the former vs. the concave shape of the latter; it is rather the solidity of the food items to which knives are commonly applied against the liquidity of those where we reach for spoons. Likewise, the contrast between *knife* and *dagger* derives less from the different shape of their blades, but how they are used (which of course causes the shape-difference because the instruments are built for optimal utility in their primary uses). To the degree that we want to introduce quantitative dimensions here — let's say, contrast *knifespoon* on a solid/liquid axis, with spreadables (jam, butter) falling in between the two ends (and therefore marking a certain muddling of the concepts; butter knives are probably experienced as rather atypical, non-prototype examples of knives). To find relevant dimension of contrast (which are also conveniently numeralizable) we need to look past tokens of the concepts themselves toward their surrounding conceptual contexts.

This, then, will be the sort of analysis I feel most legitimates applications of Conceptual Space Theory to natural language — a nudge pulling the original theory somewhat toward a more situational, context-sensitive analysis of concepts. I will argue that modifying Conceptual Spaces in this contextual manner actually brings Gärdenfors's original theory closer to the prior concerns of Cognitive Linguistics, and especially Cognitive Grammar.

1.1 *Quantitative Dimensions and Situations*

Thus far, I have argued that modeling inter-concept relations via quantitative axes — projecting concept-extensions onto a quasi-mathematical space which captures dimensionally the grounds whereby concepts may differ — is semantically reductive, insufficiently attuned to contextual and situational cues where concepts operate. This does not mean that quantitative sketches of conceptual differences are inaccurate, however, only that the full situational contexts where given concepts are typically used has to factor in to quantitative models. Consider the contrasts between these sentence-pairs:

- (1) There are dishes all over the table.
- (2) There are dishes on the table.
- (3) There are stains all over the table.
- (4) There are stains on the table.
- (5) We sat at a booth with a Tottenham Hotspur logo all over the table.
- (6) We sat at a booth with a Tottenham Hotspur logo on the table.

The foundational contrast between these sentences is one of quantitative extent: *all over* suggests a greater spatial spread or density than *on*. The particulars of the contrast depend on context: in (1) and (2) the dishes on or all over the table are discrete objects, while in (3) and (4) the stains are understood to be extended across the table's surface (but still distinct and bounded; "stain" in this context does not typically lexicalize something fully expanded over the entire surface, like a varnish). In (6), however, we *do* hear that the crest is likely spread across the full surface, as a continuous spatial region; whereas (5) implies that the analogous decoration is found on one (smaller) part of the table-top.

Notwithstanding these variations, however, in each case the *all over* version suggests greater spatial extent and density than the alternative, adjusted for particulars on whether this extent is discrete or continuous, and partial or total (vis-à-vis the table top). So here we do find a quantitative contract underlying conceptual differences, although the differentiated concepts do not have single lexical encapsulations. What is really quantitatively modeled here are different concepts of being "on" or "all over" a table. The full conceptual details depend on *what* is thus on the table. So the concept of *dishes* being on a table is generally implicated in the act of removing them from the table; whereas the concept of *stains* on a table is more a description of the table's condition or appearance (you cannot remove stains the way you remove dishes), and likewise a sport teams' insignia on a table (part of its surface) would normally be conceptualized as a decorative element (imagine a pub frequented by Spurs supporters). These are different conceptual spaces adapted for different mundane situations, but within those "spaces" we can indeed find a quantitative contrast.

Cognitive Linguistics, in the decade or two prior to Gärdenfors publishing Conceptual Space Theory, had certainly developed much of its analytic methodology around prepositional patterns and contrasts like *all over* vs. *on*. Which choice of prepositional words or phrases a speaker executes becomes a cue to how the speaker is appraising the present situation: *dishes all over the table* implies that the speaker is concerned with the task if removing them, because she chooses wording seemingly emphasizing their number and extent. The (2) version could have the same implications in context, of course, but it could also be said by someone telling where the listener could find a plate, or someone midway through *setting* the table. Spatial configurations are an important criteria for how people cognize situations, so it is reasonable that the building-blocks for significations marking speakers' conceptualizations would include usage patterns highlighting spatial extent, repetition, or interrelationships insofar as these communicate conceptual details vis-à-vis the relevant situational context.

Analysis of cognitive schema, which in this sense orchestrate cognitive linguistics treatment of prepositions, are equally applied to analyses of *verbs*, and here again I would argue that we find interesting quantitative contrasts between concepts. Speakers' choice of verbs, as with choice of preposition, indicates how speakers are conceptulizing the relevant situation. This can include various parameters of cognitive consrual — the intent behind actions (compare *spill* and *pour*), for instance, but certainly spatial and/or temporal structuration can be foregrounded. We can accordingly think of sentence-pairs, analogous to (1)-(6), which similarly use a difference in verb-phrase to show situational contrasts:

- (7) Water poured out of the hose.
- (8) Water trickled out of the hose.
- (9) The car hit the wall.
- (10) The car scraped the wall.
- (11) He painted over the wall.
- (12) He painted on the wall.
- (13) Both tourists and locals flocked to the new museum.
- (14) Both tourists and locals visited the new museum.

Assuming the rest of each situation is constant, we can see gradations of applicability for different (but somehow similar or overlapping) verbs being modeled via quantative contrasts, analogous to (say) color-space axes in Gärdenfors's treatment. Speed and pace presumably mark the difference between *run* and *walk*; speed and/or force between *hit* and *scrape* (or, say, *touch*); volume constitutes the contrasts between *pour* and *trickle*, or *flock* and *visit*; meanwhile the preposoitional or/over contrast propagates to verb phrases in *paint over* vs. *paint on*, so the latter implies greater extent.

I could give similar analyses for adjectives and adverbs — indeed, Gärdenfors's original examples such as color are arguably predominantly ajectival in themselves; the noun "red" is probably derivative of the adjective. In fact, it seems as if the quantitative metamodel behind Conceptual Space Theory is strongest for every part of speech *except* nouns, and that the theory needs revision to be a comprehensive account of concepts lexified as nouns. This can perhaps be explained by how the "ontological" ground of verbs, prepositions, or adjectives/adverbs — actions, events, properties, spatial relations — are prone to gradations: the speed or force with which an action is performed, the size of an object or a stretch of time, the nearness or distance of two points, are all continuously varying measures. Nouns, by contrast, tend to be constrained by the categories of natural objects or (for man-made artifacts) the uses to which objects are put, which favors the distribution of objects into discrete classes rather than graduated concept-spaces. It is not as if, say, a continuous gradation in shapes varies from knives to swords, so that there is some boundary where the knife-concept gives way to the sword-concept. Instead, the conceptual difference reflects the different purposes for which the two instruments are designed; the fact that these differences also engender a correlative, quantifiable spectrum of size and shape where knives are clustered in one region, and swords another, is largely epiphenomenal.

Even in the context of verbs, say, gradations involve judgments of applicability more than the definitive ground of conceptual meanings. Consider examples like:

- (15) Cars sped along the highway.
- (16) Cars crawled along the highway.
- (17) It started pouring while we were walking the dog.
- (18) It started drizzling while we were walking the dog.

We can observe that *sped* vs. *crawl* invokes comparisons in speed, and *pour* vs. *drizzle* in the volume of rain. But we should not thereby reduce our semantic models of the verbs, i.e. to treat "speed""crawl" as definitionally equivalent to "go quickly" (respectively, go slowly) or *pourdrizzle* as "rain intensely" (respectively, rain lightly). We instead should evaluate these signifiers in context: (2) implies, for example, that cars are moving unusually slowly becaue of some impediment or traffic jam. Someone saying (2) is presumably not just describing the cars' speed, but evoking a situation where cars are being held up; so (2) is indirectly asserting the existence of a traffic jam or some similar obstacle (likewise (1) suggests cars traveling quickly because there are no such blockages; indirectly reporting that traffic is "good"). Likewise, *pour* vs. *drizzle* is significant mostly in how the characterization of rain suggests alternative responses: people tend to walk through drizzling rain, but try to seek shelter from pouring rain.

So even in the verb case, conceptual meanings are still situational: we have a certain prototype scenario involving cars stuck in traffic (along with counterposed scenario where cars flow unhindered by traffic); respectively, scenarios of persevering despite a light rain and of trying to avoid a heavy rain. Verb-choices like *spedcrawled*, or *poureddrizzled*, therefore invoke these situational prototypes, and we should treat this dynamic as their semantic essence. Nevertheless, there is still a quantifiable gradation in applicability: as cars' pace varies, the occasions when people will except *stuck in traffic* or *flowing down the highway* as reasonable glosses on the situation will ebb and flow; likewise for when either *pouring rain* or *drizzling rain* would be accepted as useful descriptions of the weather. We can, then, see quantifiable dimensions as underlaying the waxing and waning of various situational prototypes being deemed proper matches for the *current* situation, and therefore for which prototype generates the appropriate word-choices in the current context.

Implicitly, that is, we assume that both speakers and addressees share roughly compatible stocks of situational models, and that speakers select one of several possible prototypes to posit via word-choices.[3] The "meaning" of a particular word (e.g., *pour* in the context of rain) lies in how the choice of *that* word, among alternatives, evokes one situational prototype and not others. On this theory we cannot lift meanings outside this situational basis, so purely quantitative metrics cannot actual model conceptual meanings in themselves; but there often *is* a quantitative dimension to how prototype scenarios differ amongst themselves, and therefore by extension to the space of peer concepts soliciting each prototype by rejecting the alternatives.

From this angle, we can agree with Conceptual Space Theory that quantitative dimensions should be semantically modeled: how dimensions combine to form multidimensional spaces; how they relate to perceptual phenomena and physical processes; how they are mutually compatible or incompatible (e.g., which dimensions can or cannot be compared against each other). Cross-dimensional relations are more subtle in language than in contexts like science, where measurements are bound to units with strictly delimited rules. We can say, though:

- (19) His house is an hour from here on the interstate.
- (20) We got four inches of rain last night.
- (21) We picked 50 dollars worth of blueberries.

That is, we mix measures of time and distance (1), rain-intensity and height (2), or price and quantity (3) — using one magnitude as a proxy reference for the other. Again the foundation for these substitutions are situational and pragmatic: defining distance by how long it takes to travel that far; estimating rain amounts by meteorological collection; quantifying volume by how much an equivalent amount would cost were it procured commercially. These patterns are convention-driven, but they only work because there are empirical pragmas through which different magnitudes can be cross-referenced (mapping distance to travel-time, say). Dimensional "blending" is foreclosed without the reauisite pragmatic logic:

- (22) That hotel is 160 miles down the road — about four hours.
- (23) We got four hours of rain last night.
- (24) ?We got 160 miles of rain last night.

[3]Here I use "speaker" in a general way, which can include written language and various degrees of contact between speakers and addressees (the latter of whom may be unknown to the speaker, in the future, etc.).

Dimensional "proxying", that is, is not transitive (at least across situations): there is a context where time proxies distance, and another context where the duration of a rainstorm reports on its intensity; but there is no context where the two substitutions can be combined (purporting to describe the intensity of rain by a spatial distance metonymically standing for a length of time). This last counter-examples shows that a certain empircally based "dimensional analysis" is relevant to semantic models because it anticipates acceptable usages like (1) and (2) as well as nonsensical ones like (3).

Analogously, dimensions can be semantically brought together in suggestive packages:

- (25) He hit the ball with his fist.
- (26) He grazed the ball with his fist.
- (27) He shanked the ball with his fist.
- (28) He tapped the ball with his fist.
- (29) He touched the ball with his fist.

Of these, (1) suggests the hardest, most cleanly hit ball — (3) sounds as if he intended to hit the ball firmly, but miscalibrated on the direction of the strike; while (4) and maybe (5) implies that he touched the ball on a flush angle, but very softly. Or, (5) can also be read, like (2), as making only indirect contact, however forceful. The applicability of these characterizations thereby seems to depend on two dimensions — the amount of force whereby he attempted to hit the ball, and the angle of impact, determining how much force was actually imparted. If we imagine the situations associated with these different verbs, we can perceive a de facto space of concepts quantitatively ordered by these two dimensions. Here, again, empirical logic determines *how* dimensions may be unified into a multi-concept collective, but semantic convention then entrenches certain regions in the dimensionalized space according to one or another lexical anchor.

So these dimensional models can be a useful tool for investigating conceptual relations in at least *some* contexts. This does not imply that all contrasts among "peer" concepts will have ready translations into quantitative structures, however. Consider the constrast between *pour* and *spill*, whose main criterion is distinguishing deliberate acts from accidents. The verb *spill* evidently packages several components into a conceptual nexus: that which is spilled (canonically a liquid); something the liquid spills *from*; and usually a person who "causes" the spill by mishandling the container. The point of *spill* is not just that water (say) leaks out of a vessel, but that the liquid's escape occurs in a context where ordinarily someone would handle the liquid without spillage — we would be less likely to use *spill* in a case where water leaked out of a hole in a bucket, or fell down into a basement during a rain storm.

Or, at least, these prototypical uses for *spill* engender other cases which conceptually vary some of the details:

- (30) Due to heavy rain, water spilled over the banks of the lake.
- (31) Due to heavy rain, water spilled down intp the basement.
- (32) She spilled water on the floor because she was carrying a broken bucket.
- (33) She let the bathtub fill too high and water spilled over the edge.
- (34) A bunch of coins spilled out of my bag.
- (35) People spilled out of the subway.

In (5) and (6) the "substance" is not liquid, but the implied movement still evokes the visual qualities of spilled liquid; moreover (5) preserves the sense of *spill* as accidental, while (6) preserves the schema that the spilled substance escapes from some container intended to hold it in. In (3) and (4) there is a sense that she could have prevented the spill from happening by being more attentive, so the core concept is preserved in the sense that the spill was not a normal or inevitable outcome, but was facilitated by a certain negligence on someone's part. There is no comparable "blame" in (1) or (2), but these uses do preserve the core of the concept wherein the spill is abnormal. In short, concepts keyed to a central sort of occurance can give rise to variant uses which relate back to the "core" in some (potentially metaphorical or elliptical fashion). But analysis of such outliers still depends on identifying prototypical situations where a given verb (or any other kind of word) would be employed.

Quantitative dimensions are therefore only one form by which situational structures may be articulated. In general, situational schemata lay out a nexus of components which collectively designate some conceivable state of affairs, some ordinary occurance (ordinary enough to have its own semantic entrenchment). These various "elements" may be related in many ways — cause to effect, agent to patient, quality to bearer, means to ends, and so forth. Often these relations are spatiotemporal and/or conducive to summaries via quantitative dimensions, but not always; and in any case analysis of quantitative dimensions in those contexts should be understood as one mode of situational reconstruction in general, not as in itself the core of semantic theory.

I will also argue that many common concepts are flexible, and that this actually structures the pattern in how concepts translate to word-usage. Technical or narrowly defined concepts are in many cases more likely to become grounded in special-purpose vocabulary, whereas the lexicon of everyday speech is more likely to to include multi-faceted concepts which subsume a range of more specific cases. One reason is that concepts — at least via their embodiment in language — serve communicative purposes. Open-ended concepts are often more appropriate for their given conversational needs than narrower, more precise ones:

▾ (36) I was in a restaurant in Montreal when I saw some Habs players walk by!

▾ (37) Shall we go find a restauarant for dinner?

There are more specific concepts than *restaurant* — *steakhouse*, *taqueria*, *sushi bar*, etc.; but a choise that specific would potentially distract from the story in (1), or come across as presumptuous in (2). Maxims of conversational relevance or even politeness imply that we should assess the breadth of words we choose, and avoid being either evasively vague or pedantically specific. On that standard, concepts' degrees of generality or specificity is one factor in the degree of appropriateness for word-choices in given situations.

This means that broader concepts are often more useful — and so more likely to be lexically chosen — than more granular ones. On that theory, selective pressures would seem to pull everyday lexicon toward relatively broad and flexible concepts more than strict and narrow ones. In that case, everyday concepts would tend to evince an wide range of specific instances, which (among other things) allows dialogs to branch in various directions. A *restaurant* can range from a fast-food joint to a gourmet establishment; a *house* can range from a cabin to a mansion, or just about. Using these generic terms both allows our discourse to take on only the degree of precision warrented for our immediate semantic intents; and also leaves open the future course of the discussion to probe the concepts more exactly. I might say:

▾ (38) I tried a new restaurant last night.

▾ (39) I just visited my parents' new house.

Here my co-conversants have the option of responding in a way that seeks more details within the space covered by the word-choices: that I should describe the restaurant, say, or the house. The genericity of these concepts make follow-up descriptions readily available. But they also allow for the conversation to go in entirely different directions.

Moreover, broad concepts are valuable *because* they are conducive to a multitude of further descriptions. Once something is identified as a restaurant, I could add further detail by talking about the restaurants' food, or its location, or what happened during the course of a meal there. The concept provides an organizing framework for distinct subsequent discussions, which is part of its merits — referring to an establishment as a *steakhouse*, say, the more fine-grained designation, would seem to foreclose subsequent discussion outside the narrower scope of the actual food. In other words, I would presumably choose that term over generic *restaurant* because I wanted to highlight something about its food (or ambience or other commercial qualities), but this implies that I am actively seeking to present details along those lines — details of a kind where the more specific word-choice is relevant. It would be more of a stretch to analyze *steakhouse* as an "organizing framework for further discussion", or elaboration; here I instead choose a lexified concept which is already elaborated, already foregrounding one conceptual dimension at the expense of others.

A theory of how concepts get embedded in language, then, should analyze the play of generality and specificity, and how *broad* concepts open up spaces for further refinement whereas *narrow* concepts tend to add detain in particular, predetermined ways. This means that the broader concepts, as cognitive tools, play their roles in part by *evoking* spaces of further elaboration and by *covering* a spectrum of cases, so that conceptualization can proceed from the more open-ended to the more definite. Simply by using the term *restaurant* I give you relatively little detail about what the place looks like, what the food is like, and so forth. At the same time, the concept has enough structure that we know how to fill in missing details through conversation. While the concept has many subsumed cases — many kinds of businesses fall under the *restaurant* rubrick — it has a conceptual order which specifies routes toward greater and greater specificty. Anyone familiar with the concept will know that information about a particular restaurant can be filled in via more details about its food, or its location, or its appearance, and so on.

While contrasts like *house* against *cabin* or *mansion* can capture how concepts acquire specificity by marking off "peer" concepts, an equally salient aspect of conceptual reasoning — as hopefully the restaurant example shows — is how broad concepts *internally* structure our appraisals of the spectrum of their instances: how instances differ from one another (a restaurant being cheap or expensive, a house large or small, one or two story, old or new, etc.) and how language-users share a competence in leveraging dialogic patterns associated with the concept to fill in details. This cognitive utility vis-à-vis concepts' space of variation is not fully captured, I would argue, either by "prototype" theories (which would try to ground the concept *house*, say, in some hypothetical house exemplar) or by a Conceptual Space Theory which looks at concepts *compared to one another* more than to concept *instances* with their own system of differences, rationally solicited by the concept itself.

My arguments thus far raise the question of how we can take notions arising from Natural Language and apply then, as intuition primes or theoretical constructs, to *formal* semantics (or the semantics of formal languages, e.g., computer programming and data modeling languages). Gärdenfors's original theory was embraced, certainly, because it seemed as if quantitative and dimensional concept models could be naturally lifted from a humanistic context and applied in a more formal and technological setting — quantitative metrics and dimensional structures are foundational media for digital/computational environments. As I have enmeshed dimensional models in an umbrella of more cognitive-linguistic persepctives — situational prototypes, cognitive schema, dialogic maxims, lexical entrenchments — we move toward cognitive faculties whose structures are much harder to emulate on computers, or even to theorize as computational vehicles in the first place. They emanate more from our adaption to social and situational worlds than from a mechanistic computing platform somehow orchestrating human reason inside the brain. This is not to deny that microscale analyses of neurological processes — the functioning of brain cells, say — may be suitable to some computer metaphor; but the emergent patterns through which brain activity manifests as worldly consciousness seem to be structurally quite different from the macroscale structures of comptuer software or digital information spaces, so that any "mind as computer" analogy does not really help us formulate a semantics that bridges human language and digital artifacts.

Still, certain themes do arise from my comments about human language which, I will argue, are applicable to "formal" languages in some fashion. First, any notion of "conceptual spaces" should look to the internal organization of concepts' extensions, not just to inter-concept differences. Second, concepts are more often than not grounded in situational appraisals that depend on pragmatic, functionally organized situational models. Conceptual applicability is functional more than logical — that is, we tend to subsume a bearer under a concept if it plays a functional role, or participates in a functional process, specified by the concept: a *knife* as an instrument for cutting (usually food); sharpness as the attribute enabling that; to *cut*, *carve*, or *slice* being actions we can take via the functional affordances of a knife. These functional roles — more than "bundles-of-properties" (knives qua hard, sharp, oblong, etc.) or protype examples (resemblance to some imaginary "ur-knife") — lie at the core cognition/language interface where concepts become part of semantics. And, third, concepts acquire meaning through their use in dialog, and in general the communicating of ideas and information between people. Concepts are not static mental "pictures"; they are refined and reimagined by the conversational needs of people trying to syncronize their situational understandings, and collective actions.

These aspects of concepts do, in fact, have some analogs in the milieu of formal semantics, though I will have to circle back to this analysis to make such connections explicit. My immediate goal is instead to return to the core of Gärdenfors's original Conceptual Space Theory, but now to examine its applicability not to human language or cognition, but to the design of artificial "formal" languages and mechanisms to share and represent structured, unambiguous information aggregates.

1.2 *Formal Semantics and "Software-Centric" Data Models*

From its origins in Natural Language semantics, Conceptual Space theory quickly became adopted to scientific and technical contexts. The rationale behind this progression was effectively that scientific data — and other technical artifacts — were themselves the products of conceptual activity, and reciprocated the morphology of conceptual structures. Formats such as Conceptual Space Markup Language, for representing scientific (and othersturctured) datra, tried to establish rigorous models for scientific information by borrowing from the conceptual patterns which underlie science. It was argued that this conceptual foundation could make data representations more expressive — more true to science; more effective as practical tools for conveying information between different computing contexts in a usable and accurate fashion — as compared to data-sharing paradigms that made no reference to conceptual underpinings.

Bear in mind, then, than *data sharing* was an essential rationale for formalizing Conceptual Space Theory as a technical paradigm, or at least a framework through which technical components could be implemented. To understand the computational side of Conceptual Space Theory we therefore need to specify what are the challenges that make data sharing non-trivial — why we need sophisticated theories in order to create systems which, in the face of it, effectuate the seemingly straightforward process of moving information from one place to another — and how the Conceptual Space perspective helps address those challenges.

I'll take scientific data as a case in point. The point of encoding scientific data in digital archives is not only to warehouse it for the future; it is also to share data between distinct computing environments, which may be designed for different scientific goals or modes of analysis. Scientific data, in short, is only really meaningful in the context of scientific *software* which can reconstruct digitized data into analyzable and human-interactive forms, with visual displays and operational capabilities that human software users can leverage.

Since the rise of the World Wide Web, much of the technical focus for data modeling and sharing has been *outside* of the software context; attention has been placed especially on representation formats like XML, JSON, or RDF which abstract from the particulars of specific programming languages and software environments. In theory, general-purpose technologies are better than ones bound to a narrow group of computing environments. For this reasons, many data models (such as CSML) are formulated as concrete languages derived from XML and therefore independent of particular software development platforms, rather than as tools which could only be used in conjunction with a single programming language, or a single development environment.

The problem with this evasive paradigm — trying to avoid dependence on any software-oriented foundation — is that full-fledged data models are incomplete *without* substantial overlap with software concerns. I believe, in short, that data models are incomplete unless they express, not only the numerical or statistical details of information spaces, but how computer software should make this data available to human users. In the case of scientific data, for example, a framework for describing the structure of molecules (e.g. CHEMML, the Chemistry Markup Language) coexists with conventions for displaying 3D molecular models. Or, medical records generated by health-care providers, or during clinical trials, are associated with software ecosystems which enable researchers or clinicians to obtain information about individual patients, patient cohorts, demographic/epidemiological trends, and so forth.

The software ecosystem around scientific data is not always as rigorously articulated as the data itself, even though informal conventions and user productivity tends to inform software development — insofar as certain operational patterns, in how people use the software, become favored or expected by users, making them more productive and more committed to that software, those design patterns tend to be preserved. Informally, then, software systems can be anchored in a network of expectations that have an engineering role not dissimilar to formal specifications; but without these conventions being explicitly defined, they are harder to incorporate into rigorous data-modeling initiatives.

Given these concerns I believe in promoting a "software-centric" approach to data modeling, which tries to systematically convey the overlap between abstract data models and concrete implementational priorities, such as how to map data structures onto visual/GUI (Graphical User Interface) components, or restructure data for database persistence. This means developing data models in a software-centric context; organizing such models around technical constructions that are especially relevant in the realm of concrete software implementations, e.g., programming language type systems. The goal of multi-platform models which can be utilized in multiple development environments remains a worthy target, but this need not be achived by abstracting from software foundations entirely — one alternative is to use concrete software components as "reference implementations" which other software can use as a guide or prototype.

The upshot of this argument, in short, is that *formal semantics* can be oriented with a priority to the semantics of data models formulated in a software-centric context. This is the perspective I will take in exploring how Conceptual Space Theory can engender semantic models consistent with this kind of software-centric paradigm.

2 Types' Internal Structure and NC4 Type Theory

When data modeling is conducted in a "software-centric" milieu, notions of *type systems* and *typed values* become especially important. In the context of text-based serialization languages, like XML, data structures can be assembled without mandating conformance to predetermined schema. Assuming there is some core set of "basic" types — such as integers, floating-point numbers, and Unicode character strings (to represent names and phrases in natural languages) — data aggregates can be assembled in a "semi-structured" manner, with different types and groupings juxtaposed in no particular order. This indeterminacy is possible because each part of a data structure is embodied via a textual encoding, and text-streams can be packaged in a relatively free-form manner, with no *a priori* length or structure.

Computer software, by contrast, works with binary resources — i.e., with *binary* (rather than textual) encodings of data structures; in the general case we can see binary encodings as strings of 8-bit integers (i.e., strings of bytes, with values in the range 0-255). Binary resources have to registered in fixed-size, pre-allocated segments of computer memory. It is possible to emulate more free-form text-encoded structures via software memory, but this requires more complex implementations; it is not the underlying representational pattern which is canonical to binary data. As a result, software-centric data models should be grounded on structures consistent with the constraints imposed by binary representation at the base level, and then generalize to less restrictive aggregates via higher-level, multidimensional representations.

Whereas the building blocks of data structures in a context like XML are therefore semi-structured data complexes, the analogous foundation for software-centric data models are *typed values*, or binary resources associated with a type $\mathfrak{t}$, which in turn belongs to a *type system* $\mathbb{T}$. Data models in this sense are closely tied to the details of the relevant type systems: in a given $\mathbb{T}$, what constitutes a type in the first place, how types are combined into aggregates, and so forth.

This section will outline a model of type systems which I believe is conducive to an overall project of unifying Hypergraph Category based grammar with Conceptual Space semantics. I will build off of work I have published elsewhere, so some details will be skipped over (see [**?**] for more extensive treatment of the underyling theory).

2.1 *Cocyclic Types, Precyclic and Endocyclic Tuples*

In a hypergraph-based modeling environment, *hyperedges* may span three or more nodes (ordinary graph edge connect exactly two nodes). For *directed* hypergraphs (DHs), hyperedges have a *head set* and a *tail set*, each collections of one or more nodes. The term *hypernode* can be used to designate node-sets which are either the head or tail of a directed hyperedge; to avoid confusion the nodes inside a hypernode can then be called *hyponodes*. Directed hypgraphs which are *labeled* (generalizing ordinary labeled graphs, which are the basis of Semantic Web data, such as RDF) allow information to be associated with connections between hypernodes. Each labeled hyperedge then asserts that a certain kind of relationship exists between the entitites or sets of entities grouped on either side of the hyperedge (head or tail).[4]

Labeled DHs thereby have two different formations for aggregating information: first, how hyponodes are grouped into hypernodes; and, second, how hypernodes are interrelated via labele connections (hyperedges). This duality allows hypegraphs to combine the paradigms associated with ordinary labeled graphs and with data tuples or "records" (e.g., Relational Databases). So DHs evince a step toward a universal, expressive framework for data representation which is structurally rigorous but not tied down to simplified modeling paradigms.

Analysis of hypergraph models can bifurcate into two branches, then, depending on whether we attend to the formation of hypernodes from hypernodes or to the assertion of inter-hypernode connections, via hyperedges. I will focus on the first alternative.

[4]The relation is assumed to be intransitive, in the head-to-tail direction, thereby generalizing "Subject/Predicate/Object" triples in RDF

2.1.1 Cocyclic Types for Hypernodes

I assume we operate in a context where a type system $\mathbb{T}$ is employed in conjunction with hypegraphs, so both hypo- and hyper-nodes receive type attributions. We can then consider what sorts of types should be representable in $\mathbb{T}$ to adequately model the spectrum of hypernodes which may appear in a hypergraph. Without undue loss of generality, we can assume that nonidentical hypernodes do not overlap (i.e., no hyponode is covered by more than one hypernode).[5] Any given graph, then, seen as a static data structure, will have some fixed list of hyponodes for each hypernode (since directed hyperedges are ordered, we can assume that there is an ordering on their head and tail sets, and therefore that hyponodes have a fixed order in their covering hypernodes).[6]

In the models I propose, however, I want to focus on "Procedural" Hypergraphs, which are not necessarily static structues. Instead, each hypergraph has certain evolutionary possibilities, i.e., certain regulated operations by which it can be modified, such as (potentially) adding a hyponode to a hypernode (if that is compatible with the hypernode's type). We want, then, a more flexible type mechanism whereby hypernodes can cover a varying number of hyponodes. On the other hand, we also want these hyponodes to have types according to some fixed pattern, to preserve a usable type-attribution mechanism for hypernodes. In other words, if we allow hypernodes to cover an unconstrained list of hyponodes with any type, there ceases to be any means of sorting hypernodes into different types. The problem is then to free up type "tupling" as far as possible while preserving a strong type system at the hypernode level.

The concept of "cocyclic" types, then, is intended to convey a pattern among hyponode types which is flexible but still constrained by strong typing. Let $\mathcal{T}$ be any ordered sequence of types in a $\mathbb{T}$. I will say that $\mathcal{T}$ is *cyclic* if the sequence repeats: every n**th** type is the same, for some n. I will call a $\mathcal{T}$ *cocyclic* if it comprises a cyclic sequence preceded by a fixed-width tuple of types. A cocyclic *type* is then a product-type in $\mathbb{T}$ whose instances are hypernodes wherein their contained hyponodes have types which, listed as a sequence of $\mathbb{T}$ types, comprise a cocyclic sequence. The fixed-width tuple at the start of the hyponode-list I will call

[5]This restriction — which I call "disjointness" — can actually be weakened somewhat; see [**?**, p. ?] for details.

[6]Assume that hypernode identity is affected by hyponode order; so the same set of hyponodes cannot appear as a head-set or tail-set in two different edgers where their order would be permutated, since that would violate disjointness.

the *precyclic* part of the hypernode, while the type-tuple that repeats over the rest of the sequence I will call the *endocyclic* part.

This definition of cocyclic types can be extended outside the context of hypergraph type-attributions by considering "data fields" or other components of product-types in lieu of hyponodes. In practice, the Cocyclic model is implemented via the "PhaonGraph" library included with the dataset accompanying this paper. Note that any fixed-length product type (whose instances are fixed-length tuples of values, or, in the hypergraph context, hyponodes) is a cocyclic type with no encyclic part. Likewise, a "list" or "collections" type built on a single $\mathbb{T}$ type — a list, stack, queue, or deque of $\mathfrak{t}$s (meaning a list of $\mathfrak{t}$s which grows and/or shrinks from one or another end, or both) — is a cocyclic type with no precyclic part (although an implementation might model these instead with precyclic field tracking data such as the current length of the list). An "associative array" (a.k.a. "dictionary") using one type to index a second — where the *keys* to the dictionary come from a $\mathfrak{t}_1$ and the values from a $\mathfrak{t}_2$ — is similarly (representable as) an endocycle alternating between $\mathfrak{t}_1$ and $\mathfrak{t}_2$.

The purpose of this framework is generality — similar data structures can be emulated in a type system where tuples have to have fixed lengths, or where varying-length tuples have to contain only one single types: in these cases (what I call) "proxies" (hyponodes uniquely designating hypernodes they are not part of) can approximate the layout of cocyclic types, by analogy to programming languages using pointers or nested tuples to represent dynamically-sized collections types. However, the cocyclic type paradigm yields less indirection — less gap between the conceptual pattern represented by a data model and its software implementation — without diminishing the model's computational realizability. The "PhaonGraph" library represents one implementational strategy for modeling hypernodes via cocyclic types as an underlying data representation.[7] Of course, any individual $\mathbb{T}$ type (in the case of a hypernode with one single hyponode) can be seen as a cocyclic type with a length-one precycle.

Against this background, then, I assume that for any $\mathbb{T}$ with fixed-length types we can generalize to a related system wherein all types are cocylic. From here on then I assume that any $\mathbb{T}$ under discussion is "cocyclic", meaning each $\mathfrak{t}$ in $\mathbb{T}$ is cocyclic.

[7] In a nutshell, PhaonGraph, written in **C++**, allocates memory in fixed-sized chunks corresponding to some fixed array of hyponodes, and then maintains a separate list of pointers to the chunks thus far allocated in order. A related "PhaonLib" library uses similar techniques to provide collections values such as stacks, deques, and queues, built around a uniform list-interface providing common iterator and visitor functionality.

2.2 *Channelized Types and Channel Algebra*

In any reasonably advanced type system, some types in $\mathbb{T}$ are "function-like": they represent computations which, in some sense, take *inputs* of some type or types in $\mathbb{T}$ and produce *outputs*. Programmers sometimes talk of "pure functions" as computations that map inputs to outputs with no side effects. In most programming languages, however, the description of function-like types has to be more complex. In particular, languages can have variant sorts of input — in Object Oriented programming, for example, some (or in some languages all) functions (called "methods") have a special Object or "**this**" input which, in various technical ways, is treated differently than the method's other input parameters. Likewise, procedures have modes of "output" other than returning values — they can throw exceptions, modify input values, or have other side-effects in addition to (or in place of) returning values. Procedures implemented in most programming languages, in short, have multiple mechanisms for "communicating with the outside world" — for getting data with which to complete their given task, and for sharing the results of their operations, or otherwise effectuating some change beyond just computing a result. These alternatives generally get some representation in languages' type systems. For example, in **C++**, a method (which takes an object as a special **this** value) has a different type than a function where that same object would be passed as a normal argument.

We therefore should assume that $\mathbb{T}$ allows us, in principle, to differentiate function-like types on the basis of multiple *kinds* of input and output, and/or side-effects. It is not sufficient to represent $\mathbb{T}$ as permitting, say, given a single input $\mathfrak{t}_1$ (or list of input types) and output $\mathfrak{t}_2$ (or again a list of output types), the identificatio of a type $\mathfrak{t}_1 \rightarrow \mathfrak{t}_2$ representing functions from $\mathfrak{t}_1$ to $\mathfrak{t}_2$. Instead, $\mathbb{T}$ has to model multiple input and output modes. This variation does not necessarily yield distinct types — for instance, in **C++**, whether or not functions throw exceptions is not normally considered part of their type (you can assign the address of a functon which **throw**s to a function-pointer whose declared type makes no mention of exceptions). However, differences in input/output options *could* potentially herald differences in type attributions (e.g., a type system *could* stipulate that procedures which do not throw exceptions may never be deemed an instance of the same type as a procedure which *does* **throw**).

In [?] I introduced the idea of "Channel Algebra" to model different forms of input and output (an alternative outline of this framework can be found in [?], which is distributed alongside this paper). In essence, each channel is a separate mode of input and output, and procedures are assigned types based on grouping together one or more channels. I say that a type system is *Channelized* if function-like types in $\mathbb{T}$ can be described by describing the "channel complex", or sums of channels group together, specifying the kinds of inputs and outputs a given procedure recognizes (along with the types of values passed in to or returned from each procedure). I use the term *carrier* to designate the resource (e.g. a source-code token, or a hypernode in a graph used to model computer code) holding a value in the context of a procedure. Channels are then sequences of one or more (or potentially zero or more) carriers. Carriers are distinct from values (and from types) because they have states separate and apart from the values they hold: when a procedure throws an exception rather than return a value, for instance, the carrier(s) in that procedures "**return**" channel will hold no value, which on this theory is a valid carrier-state.

In [**?**] I also discuss channels and carriers, but from a more graph-oriented perspective. There I propose channels as extra structures defined on hypergraphs, grouping together multiple hyperedges as aggregate units. The two models fit together insofar as the definition of Channelized Type Systems is one application of the construction of channels on hypergraps, where procedures' types are established via "code-graphs" exemplifying the procedure's signature (channel complexes) and calls *to* the procedure (which I call channel *packages*). Hypergraphs conveying the form of channel complexes and packages thereby apply the general notion of hypergraph channels to the specific project of modeling procedure output and input kinds via code-graphs.

The rationale for introducing channels into hypergraphs, vis-à-vis models of procedures expressed in computer code, is evident similar to the hypegraph formations defined in recent work on Hypergraph Categories (providing the mathematical background to the syntactic side of the "**HCS**" synthesis). Making connections between formalisms developed in a mathematical (in this case, Category-Theoretic) context, and those based more on applied computer programming, is not fully rigorous — to some degree I would be interpreting or hypothesizing about the practical intent, or possible applications, behind the specific Hypergraph formulations chosen by the researchers behind the **HCS** idea. With that caveat, though, it certainly *seems* as if those mathematicians generalize from graphs to hypergraphs to capture some of the same procedural generalizations as motivated my "Channel Algebra" proposal.

More exactly, the Hypergraph model specifically laid out for **HCS** embodies (what we can call) procedures — the Category-Theoretic setting prefers to talk of "morphisms" instead[8] — in hypernodes, and their inputs and outputs in edges incident to those nodes. An unadorned input-versus-output distinction does not actually mandate *hypergraph* treatment: ordinary directed graphs have the structure to distinguish edges coming *in* to a node from those going *out* of a node. In my "channel" theory, I work within a hypergraph framework because we need a more fine-grained edge-classification than just input and output; as I outlined above, there are multiple *kinds* of input and output channels. Channels therefore group edges together in a fashion that introduces extra structural components outside the definition of ordinary labeled graphs.

In the case of **HCS** Hypergraph categories, the motivation for adopting hypergraphs is oriented more toward the idea that inter-procedural connections represent "information flows". In this framework, computer software can be thought of as an interconnected system whose architecture can be summarized by graphs: with procedures as nodes, an edge exists between procedures when the output of one procedure becomes an input for a second. Moreover, the theory uses nodes *also* to represent information "entering" the system — in effect, data being presented to a procedure which does not arise as the result of some other procedure, but rather as information obtained via some measurement or observation of external states.

Moreover, nodes also represent side-effects which can be effectuated by a software system.[9] Suppose a procedure concludes by formulating an instruction that a rectangle of a given size and color is to be drawn on a computer screen. The values describing that desired effect are understood to be "outputs" of the procedure, but instead of their being passed to a further procedure they are somehow translated to tangible effects in the software's external environment (in practice, this would presumably happen by calling some system kernel function, but in the abstract sense we can treat this as an output which is "absorbed" by the computer rather than passed on as an input).

Taking these two ideas together — *inputs* which are measurements of external states and *outputs* which are effects *on* external states — in the corresponding graph representations we then have directed edges which have target nodes but no source nodes (for state-inputs) or which have source nodes but no target nodes (for effect-outputs). Authors like Cocke *et. al.* then use hypergraphs to model these pattern by leveraging a generalization wherein Directed Hypergraphs' cardinality, for either head or tail, can be any quantity (includ-

[8] For technical reasons why I prefer different terminology — and to reserve *morphism* for a limited space of intertype functions — see [**?**, p. ?]

[9] Hypergraph Categories are not specifically about software, but reasoning about software behavior is cited as a possible application and used as a hypothetical case-study.

ing zero). That is, Hypegraphs in this Category-Theoretic contexts allow for hyperedges with no source or no target, as well as hyperedges with multiple sources and/or targets.

The state/effect systems thereby represented (by head- or tail-empty hyperedges) have a correspinding construction, in practical software, via techniques generally described as "reactive", as in "Functional Reactive Programming". In this paper I will point particularly to the so-called *signal/slot* mechanism used in C++ within the QT libraries. A head-empty "state" edge, on this analogy, corresponds to a *signal* which triggers a "slot" procedure; and a tail-empty "effect" edge corresponds to an operation of "emitting" a signal. I will discuss this analogy in more detail below.

There are various routes toward generalizing graphs to hypergraphs; Hypergraph Categories are only one example. Also, technical presentations of hypergraphs are not exclusively mathematical: certain software libraries and graph databases also embody formal hypergraph models, with varying features (for instance, some allow hyponodes to also be hypernodes; some support undirected hyperedges; some allow circular hyperedges wherein the head set is also the tail set). The "Channelized Hypegraph" (**CH**) setup I outlined in [**?**] is different in some details than Hypergraph Categories — including by introducing channels as an extra construction on graphs — but I believe the frameworks are sufficiently close that the Channelized Hypergraph constructions (and therefore Channelized Types) are a plausible extension of Hypergraph Categories from the viewpoint of integration with Conceptual Space Theory.

2.3 *Constructors and Carrier States*

The **HCS** version of Hypergraph Category Theory makes the trenchant point that graphs modeling values only as they pass between procedures — outputs becoming inputs — are necessarily incomplete, because values have to original from *somewhere*. In practice, some data handled by a software component comes from external sources — files, packets sent over a network, CyberPhysical devices, and so forth. Otherwise, often values come from computer code directly: all programming languages have some notion of *source code literals* which permit values (at least, those of the most basic types) to be initialized from character strings in source code. For example, the literal token "**99**" becomes the *number* **99**. One question to be addressed for an applied type theory — i.e., to specify the nature of an applied type system $\mathbb{T}$ — is how and which types can be constructed from source code literals in this manner.

Representing values passed between procedures can be seen as a "syntactic" gloss on computer code, because programming language grammars are built around how expressions in each language describe the seqeuence of function-calls specified to enact some algorithm or calculation. But understanding how different typed values interact with function-calls is also a *semantic* matter, characterizing the semantics of individual types. Given a specific instance v of type $\mathfrak{t}$, we may analyze — what sort of procedures can produce v? Is v obtained from some other $\mathfrak{t}$-value, or can it be initialized via a source-code literal? Is v a "default" value for $\mathfrak{t}$ which can be created *ab initio*, with no further input? If v is derived by modiying some alternative $\mathfrak{t}$-value, can we identify this prior value, thereby "deconstructing" the construction which yielded $\mathfrak{t}$?

Suppose, for instance, that $\mathfrak{t}$ is a list of signed 32-bit integers. The "default value" for such a type is almost always defined as an empty list. New $\mathfrak{t}$-values are created by appending a number to the end of the list. Given a non-empty list v, we can always "deconstruct" v by noting that v is derived by adding some number n to a shorter-by-one list v'. Algorithms which depend on examing the whole list — finding its largest element, or counting how many times a given number appears, or how many elements are larger than some target — can be conveniently expressed by examining the list recursively, each time stripping the last number and repeating the test on the shorter-by-one outcome. To count how many numers are positive, say, check each n at the end of the list, increase the result by one if $n > 0$, and repeat that process with the smaller-by-one list obtained by "deconstructing" the current v into v' and n. The style of recursive algorithm is endemic to Functional Programming, where it yields procedures that do not need to employ *iterators* which loop over the elements of a data collection. Recursive procedures can eliminate the loop initializations and tests that can make non-recursive, iterator-based code more cluttered and obscure.

However, this recursive style of programming is only possible if specific metadata is embedded with $\mathfrak{t}$ values which allows "deconstructing" $\mathfrak{t}$ instances into construction *patterns* and allows $\mathfrak{t}$s to be reused in a recursive fashion (e.g., repeatedly calling a procedure on shorter-by-one lists). This functionality is not available for simpler in-memory representations of data structures, like "linked lists" (a sequence of pointers to each value paired with pointers to the next value) or C-style zero-terminated arrays. In order for $\mathfrak{t}$ to support recursive algorithms in lieu of iterators, it needs to be expressly implemented in anticipation of this pattern. Conversely, types also need extra functionality (e.g. **begin()** and **end()** methods in C++) to support iterators, and extra functionality (like **operator[]** in C++) to support array-based access.

In effect, the semantics of types is much more detailed than simply describing the kind of values $\mathtt{t}$ may instantiate. A "list of numbers" may have one abstract profile, but there are a broad range of practical implementations which build, traverse, and read numbers from the list in different ways. Two types which have mathematically the same "space" of values may be distinct types with very different programming interfaces. While it is abstractly true that any non-empty list can be "deconstructed" into a single element and a shorter-by-one predecessor with that element removed, a given list implementation may not support procedures to compute that deconstruction in an efficient manner. As a result, mathematical properties of types' sets of possible values have only limited applicability to types' operational semantics.

This point also reinforces the insight that types, in applied type systems such programming languages', are not really mathematical entities — they are digital artifacts designed by an implementer to be programmatically employed in specific ways. When analyzing types we therefore have to explicate the usage patterns that are facilitated by their implementations. Type systems can expedite this process by allowing types to be described in ways that clarify their intended and expected use-cases.

The first step in descriging types' usage, moreover, is to account for their *constructors*, i.e., for the procedures which create new $\mathtt{t}$ values. Constructors are different from other procedures which output $\mathtt{t}$ values because constructors are internal to how the type is designed; they are in a sense "part of" the type. In most cases, any programmer can write procedures which return instances of $\mathtt{t}$s in $\mathbb{T}$, but most type systems have restrictions on where $\mathtt{t}$ *constructors* are implemented. Constructors are intrinsic to types in that redesigning constructros for $\mathtt{t}$ is understood to modify $\mathtt{t}$'s interface, whereas simply writing a procedure which returns a $\mathtt{t}$ is not normally seen as "changing" $\mathtt{t}$. Moreover, any "external" procedure which returns $\mathtt{t}$ is understood to call a *constructor* for $\mathtt{t}$ to obtain the value to be the external procedures outcome, or at least to call some other procedure which calls a $\mathtt{t}$-constructor, etc. — whenever a $\mathtt{t}$ is the *outcome* of a procedure, at some point, during some (maybe nested) procedure, there is a call to a $\mathtt{t}$-constructors. Constructors then become landmarks for identifying the properties of $\mathtt{t}$, because all $\mathtt{t}$-values originate from $\mathtt{t}$-constructors at some point.

In [?] I also introduce the idea of "co-constructors", which are conceptually similar to constructors but which wrap the "real" constructors in separate procedures which can present a streamlined type interface. The technical details of co-constructors vs. normal constructors are not pertinent to this paper, but suffice it to say that a type system may choose to make *co-constructors* the basic building-blocks of a type interface. On this strategy, code which is not part of the "core" implementation of $\mathtt{t}$ would not call $\mathtt{t}$'s constructors directly, but instead would call $\mathtt{t}$ co-constructors.[10]

Co-constructors are similar to what some programmers call "factory methods" or "object factories", and are similar in intent. Actual constructors have some language-limitations or peculiarities: in **C++**, for example, you cannot take a pointer to a constructor. On the other hand, insofar as co-constructors are ordinary (non-member) procedures one can take their address, e.g. for a lookup table mapping strings to co-constructor pointers; i.e., co-constructors are more amenable to "reflection" whereby programmers can dynamically invoke a procedure by supplying its name (which in turn is useful for allowing applications to be fine-tuned via scripts, or constructing objects at runtime from a database). Constructors are also sometimes "default-implemented" by compilers behind the scenes. Co-constructors or "factories", then, are in some contexts more precise representations of types' intended usage patterns than the actual constructors as recognized by compilers.

In particular, implementers of a type $\mathtt{t}$ may use co-constructors to document and differentiate patterns in how $\mathtt{t}$ values are created. Constructors for a $\mathtt{t}$ can be classified into several patterns, such as:

- Default constructors which require no further inputs. Default-constructed values may be deemed conceptually valid instances of their type (e.g. 0 is a valid integer) or may also be "special" values indicating missing data (like pointers or "**NaN**").
- Literal constructors which initialize $\mathtt{t}$ values from literal strings.
- Binary constructors which initialize $\mathtt{t}$ values from binary resources holding preexisting instances; in the simplest case simply copying the bytes in a v to initialize a v'.
- Pattern-based constructors which initialize $\mathtt{t}$ values from an aggregate data structure which may (but need not) include other $\mathtt{t}$ values. This would include building a list v' from a shorter-by-one list v by appending an element to the list, if that procedure is exposed as a (co-)constructor.
- Resursive constructors allow values obtained by pattern-based constructions to be "deconstructed" and used for recursive algorithms, as outlined earlier in the case of lists.

[10]The same applies for $\mathbb{T}$ understood not as the system embraced by a *language*, but rather the norms adopted by a library or application: in **C++**, say, developers could enforce a framework of co-constructors by strategically excluding (what **C++** would call) actual constructors from classes' public interface.

I could add further details to this breakdown — (co-) constructors which validate their input, for instance — but the basic idea is that types are often designed with implicit assumptions about how they are to be used, and these assumptions become manifest in what sorts of constructors are provided. These design patterns can be made more rigorous or explicit by consciously notating and classifying what sort of use-case is envisioned; one way to achieve this is by making object factories or co-constructors the basic public interface for a type, and then supplementing co-constructors with metadata that describes the type interface in a systematic manner.

Assuming this methodology, we then have an additional set of tools for reasoning about $\mathfrak{t}$ values. Upon enumerating various *kinds* of (co-)constructors, as above, we can specify whether a $\mathfrak{t}$-value v could be the product of a co-constructor of a given kind — whether v could be default-constructed, say, or constructed from a source-code literal. Intuitively we then have an idea of "partitioning" the space of a type into regions based on the kind of construction that results in the corresponding $\mathfrak{t}$-values. This picture is hard to make fully rigorous because is not automatically given how we should think of type-instances as a "space". For some types, we can neatly list all their possible values (say, signed bytes are every number from **-127** to **127**), but in many varying-size types the actual set of values that could be represented at any moment, in any particular computing environment, will dynamically depend on factors like available memory. It is impossible to say, for instance, just how long a list can become before it requires too much memory, which in turn would result in the proposed list failing to be constructed.

In short, we need analytic methodology which does not treat types as if they were "sets of values". In the framework of channels and carriers, I try to achieve this by reasoning about types through the carriers which hold type-instances, and by defining carrier *states* (including states orthogonal to any type system, e.g. a carrier which *doesn't* hold a value). With this foundation we can talk about the "space" of type-instances in terms of *states* on carriers. Suppose a carrier $\mathfrak{c}$ holds a $\mathfrak{t}$ value produced by a co-constructor of a given kind ($\mathcal{K}$, say). We can then introduce $\mathcal{K}$ as a state on $\mathfrak{c}$: $\mathfrak{c}$ is in the state of holding a value emerging from a $\mathcal{K}$-co-constructor. This provides potentially useful information. If $\mathcal{K}$ corresponds to a default-initialized "sentinel" value — i.e., a fallback like for unavailable data — then such a state corresponds to holding a conceptually "invalid" $\mathfrak{t}$-instance. Or, if $\mathcal{K}$ corresponds to a pattern-based construction suitable for recursion, v could be used in recursive algorithm.

Note also that many types have a notion of a "fallback" or "default" value, which may or may not be *valid*. For numeric types, that value is usually zero, but the meaning of 0 can depend on context. Consider a procedure which checks a database to learn someone's age: the default 0 may be intended to mean that this information was not found or not provided. However, in (say) a medical context, 0 may also be the (real) age of an infant. Analogously, an empty string might mean that someone's middle name is not known; or that someone does not *have* a middle name.

To avoid confusion, types often are built around two "special" values, or are engineered so that a default "sentinel" value cannot be confused with a conceptually meaningful value. In computer graphics, consider the case of color: a reasonable default value (for drawing a line, say) would be the color black — which also, in **RGB** encoding, corresponds to vector of three 0s, so it is consistent with default-to-zero conventions. On the other hand, a system may need to identify situations where a color is unknown or not specified (analogous to an unknown age vs. a baby's 0 years), so a type representing colors may have some extra value meaning "no color" — which would not be confused with or deemed equal to black.

Analogously, most programming allow the (technically negative) number **-1** to be used in an *unsigned* context, as a special "unknown" value. If someone's age is given as **-1**, then, it would be clear that the intent is to report that the age is not known, with no confusion vis-à-vis a child before their first birthday. Numerically, **-1** would actually have a binary encoding (most likely) as **255**, which would never be confused with someone's real age. Similarly, types representing textual strings sometimes distinguish *empty* strings (like when someone is known not to have a middle name) from *null* strings (representing missing or unknown data).

In practice, accounting for cases of default or missing data is an essential part of designing types, qua digital artifacts. If a $\mathfrak{t}$ value is not known or not provided, should a (valid) default value be used instead? Black, say, is a reaonable default for colors (though what about color-systems with transparency: should the default be fully opaque colors, with no transparency effects, or fully transparent and therefore invisible colors)? Conversely, 0 (when it also means zero years, not yet one yeal old) is probably not reaonable default for someone's age. When data is missing, should default values be used; if not, how should the problem be represented? These decisions influence our conceptual understanding of types' spectrum of values and their expected uses. A default and conceptually valid value (like black in the realm of colors) is conceptually different than a default value which is *not* conceptually a "real-life" instance of the type (like **-1** for someone's age) — let's call these *meaningful* and *meaningless* defaults, respectivel — and that in turn is different from an invalid value which should generally not be passed between procedures (which I'll likewise call an *invalid* default). For an example of this last distinction, a sentinel "NaN" should rarely actually be passed to procedures expecting a number — on the premise that any calculation on NaN should yield NaN, so the call would be superfluous — while it is quite common to pass pointers in C++ even though their conceptual meaning is that they do *not* point to any memory address (so, conceptually, they are not "real" pointers). In short, *black* is a meaningful default, is a meaningless but not invalid default for pointers, and NaN is typically an invalid default for numbers — or at least this summarizes typical usage patterns.

Such conceptual patterns in how we think about types are, at least in part, formal analogs to the conceptual semantics of Natural Language. Here I will in fact argue that unpacking the conceptual variations between different type-instances — meaningful, meaningless and invalid defaults, for instance, or literal-intitializable values — gives rise to a sense of types' "conceptual spaces" which is to some degree analogous to Conceptual Space Theory. Before launching into that discussion, however, I will review some more details in the "theory of constructors" with an eye toward defining very general (while still formally rigorous) $\mathbb{T}$ systems.

2.4 *Nonconstructive Type Theory*

Thus far I have suggected that types' conceptual and operational profiles can be defined in part by describing the system of constructors (or co-constructors) through which their values may be created. The process by which a particular $\mathfrak{t}$-value v has been created can be a factor in how v may be used. For example, most functional programming languages allow procedures to be implemented via "pattern matching", which means splitting the procedure into different versions or different routines based on the nature of an input value, which can be determined by how it was constructed. A canonical example is procedures defined on lists: suppose v can be "deconstructed" into a smaller-by-one sublist v' and a single element e — v is v' appended by e. The right-hand side (v' with e) can be called a *construction pattern* which yields, or defines the provenance of, v. On that basis, a procedure which operates on v could equally well, at least logically or conceptually, be seem as operating on the v' and e pair. On the other hand, if v is an empty list, then algorithms need to proceed differently than for v non-empty. In combination, this yields an outline for procedures as follows: differentiate empty from non-empty v; for the latter, allow the procedure to operate on a v' and e pattern, rather than on v directly.

Programming languages which want to support these "pattern matching" features need two capabilities: they need to be able to implement procedures which bifurcate based on which pattern matches; and they need to take a multi-part structure as a procedural input, like v' *and* e, in the place of a single carrier like v. One straightfoward way to achieve this is to differentiate procedures based on the construction-patterns evinced by their arguments. For example, we can consider a procedure which *only* operates on *empty* lists, paired with a procedure which *only* operates on *non-empty* lists.

We then have to investigate how these distinctions intersect with the relevant type system. Should $\mathbb{T}$ stipulate that procedures for empty lists have a different *type* than procedures for non-empty lists? Note that in general the empty/non-empty distinction does not yield different type-attributions: a non-empty list is not a different *type* than an empty list (assuming compatibility in the type of elements declared to go in the list). There are nonetheless frameworks which would allow a *function* taking empty $\mathfrak{t}$s (for list-type $\mathfrak{t}$) being considered a different type than ones taking non-empty $\mathfrak{t}$s. For procedures taking non-empty lists, moroever, their argument can (potentially) be converted into a construction-pattern (like v' and e), so that the single input to (this verson of the) procedure actually becomes two different inputs. Enabling procedures to be split and designed in this manner — split and paired off by pattern-matching and taking compound inputs — requires $\mathbb{T}$ types to be implemented with the requisite capabilities (e.g., calculating the proper construction-pattern for a given $\mathfrak{t}$-value). Because this sense of "pattern matching" is a common idiom in functional programming languague, it certainly needs to be accommodate in a broad-based type theory attuned to the type systems of different kinds of formal languages.[11]

Conversely, though, it is just as common for types to *lack* the infrastructure for pattern-matching in this sense, so we need to these features as *possible* but not *necessary* aspects of types. For sake of discussion, I will call types amenable to pattern-matching *constructive*; and otherwise *non-constructive*. There is no need here to formalize a technical definition of the comparison, but semi-formally I'll say that a *constructive* type has (or can be provisioned with) a procedure which, given any instance v, can return a construction-pattern which reciprocates the construction of v from other values. In this context I treat the functionality to calculate patterns as an ordinary procedural interface on a type: for constructive $\mathfrak{t}$ there will be an associated "construction pattern" type (usually a type *different* from $\mathfrak{t}$) and procedures to map $\mathfrak{t}$s to their corresponding patterns. Pattern-matching can

[11]Note that there are other, unrelated uses of the phrase "pattern matching" in programming and computer science — e.g., in the context of regular expressions, which is essentially entirely different from the current context.

then be achieved, or at least emulated, by defining procedures on the construction-pattern types for $\mathfrak{t}$ rather than $\mathfrak{t}$ itself.

Implementing constructive types introduces some complications in conjunction with an overall type interface, which might not be immediately apparent. For instance, consider types which have (what I earlier called) "binary constructors" — e.g., a $\mathfrak{t}$ which can initialize values by copying the bit-pattern of some other $\mathfrak{t}$-instance. For many $\mathbb{T}$ this option results in the theoretical possibility that $\mathfrak{t}$s could be created with any bit-pattern whatsoever. In C++, for example, a pointer could potentially point to some random area of memory (after an error in neglecting to initialize the pointer, say), from which a dereferenced yields a $\mathfrak{t}$ value manifest in a random sequence of bits. Such randomness is almost always an error, but this does not preclude code having to accept the possibility that $\mathfrak{t}$-values might be "randomly" constructed. In this case, the data which could be used derive construction-patterns may well become corrupted.

Suppose a type offers an interface to return construction-patterns for all of its values, but makes assumptions about these values' binary layout when deriving those patterns. For instance, a list type might be based on a list-pointer together with fields indexing the start and end of the list. With this arrangement, one single list pointer — i.e. one list in memory — can be the basis for multiple $\mathfrak{t}$-values, by varying their start and/or endpoints. A construction pattern for a list v could then be efficiently obtained by noting the element e at the start or end of v, and producing a v'-e pair by constructing v' as a variant on v with its start or end index advanced (respectively decreased) by one. This is a plausible "deconstructing" scheme because all the intermediate list values obtained in construction-patterns, and then potentially used in recursive procedures, share the same underlying memory; there is no copying or modifying of in-memory data. A list is then considered *empty* if its start and end indices are the same. A recursive algorithm would work with a sequence of construction-patterns, with the two indices getting closer together, until the recursion would be broken by final list being empty.

The problem here is that this design only works if the start and end indices for the original v are in the correct order (the start must be less than or equal to the end). If that requirement cannot be guaranteed, then the above derivation of construction-patterns cannot be guaranteed to work properly; in particular, a recursive algorithm might loop infinitely. As this example shows, a constructive type may need to double-check all values at their point of construction to ensure that the type's various fields and internal data are configured properly to support features like pattern-matching. A constructive $\mathfrak{t}$, in short, may need to ensure that no $\mathfrak{t}$-values are constructed without certain validation checks being performed (this would be one use-case for a co-constructor). For instance, simply copying bit-patterns from one place to another may have to be disabled as a tactic for copy-constructing values, unless the newly constructed data structure is validated.

Given these considerations, it is certainly possible that some types will be non-constructive — i.e., they decline to provide procedures that return construction-patterns, and to provision the support needed to use construction-patterns — even if the logical characteristics of the types' values would seem to support a pattern-based interface. In practice, programming languages that enable pointer-based access to values, and pointer-dereferencing, tend not to natively recognize construction-patterns, and vice-versa.

The idea of proxying $\mathfrak{t}$-values via construction-patterns has mathematical foundations, emanating from "constructive" mathematics. In particular, consider "recursive" construction patterns, where a v is "deconstructable" into a v'/e pattern, and then v' is further substitutable by a further pattern based on a v'', and so on. In many cases, for any $\mathfrak{t}$-instance v there is then a determinate sequence of constructions which eventually produces v, and likewise an "inverse" seqeunce of construction patterns which "undoes" those constructions. For instance, any list of numbers can be seen as the product of numerous constructions which begins with an empty list, and produces successively larger lists by appending one number at a time, eventually arriving at an end-result v. In a language like Haskell, the notions of *list of elements* and *sequence of constructions (appending elements to a prior list)* are understood to be conceptually indistinguishable. That is to say, $\mathfrak{t}$ values are understood to be internally inseparable from the progression of steps which provide a recipe for constructing those $\mathfrak{t}$s.

Mathematically, an analogous assumption is that the space of $\mathfrak{t}$-values is isomorphic to the space of construction-sequences which yield $\mathfrak{t}$s. We can also say that the space of these sequences is a *model* for $\mathfrak{t}$. A type mathematically representing *lists of integers*, say — meaning in this context a logical specification of some mathematical space — can be said to be modeled by the space of programs which produce these lists by starting from an empty list and progressively appending numbers. This "space of programs" is a *model* for the type insofar as it satisfies the types' logical requirements. This is one example of a project for analyzing mathematical spaces in terms of *finite constructions* which yield elements of those spaces. In constructive mathematics, proofs of propositions on such "finite constructions" is considered to be easier, or more logically sounder, than proofs which engage with infinite spaces and rely on logical indirections, like the "law of the excluded middle".

Constructive types, then, inherit this mathematical backstory insofar as such types allow us to deem types' space of values as, in essence, logically indistinguishable from the space of construction-sequence that yield those values. A *constructive* type theory would be one which treats all types as constructive, perhaps on the basis of logical or philosophical reasoning: we can say in the abstract, for instance, that any list is *logically* isomorphic to a sequence of sublists building up to it. In practice, though, I consider a type constructive only if it *explicitly* provides the infrastructure needed (or if that happens automatically given the relevant implementation language) so that "constructive mathematics" intuitions can be concretely leveraged. I will say that a type system is *non-constructive* if it does not *assume* that its types are constructive; a non-constructive $\mathbb{T}$ may still have *some* constructive types.

In short, a non-constructive $\mathbb{T}$ actually *generalizes* constructive frameworks: by allowing both constructive and non-constructive types it presents a superset subsuming $\mathbb{T}$s failing to properly model non-constructive types, as well as $\mathbb{T}$s failing to properly model *constructive* types. I claim therefore that non-constructive $\mathbb{T}$s are the requisite framework to represent the spectrum of applied type systems in greatest generality.

So, in this section I have presented several features of type systems which, I believe, allow us to achieve the greatest breadth in covering the diversity of possible $\mathbb{T}$s while staying within the bounds of software-centric, implementational rigor: type systems which are *non-constructive*, *channelized*, and *co-cyclic*, from which I derive the "**NC4**" moniker. In this paper, then, I want to use **NC4** type systems as a foundation for studying the semantics of formal languages, in the hopes of deriving an **HCS** theory — a theory of unifying Hypergraphs with Conceptual Spaces — for these languages which can be usefully paired with analogous unifications for *natural* language.

3 Types as Conceptual Structures

The types encountered in a type system $\mathbb{T}$ are technical artifacts, but in many cases they also are designed to model, or track information about, concepts in the everyday world: fragments of natural language (i.e., text); people; places; colors; events; medical procedures and diagnoses; demographic and government data; scientific data and research findings; and so forth. We can accordingly consider types (implemented in software components) as *conceptual* artifacts, but with the caveat that their conceptual details have to be modeled within the constraints of software environments and computational processes.

Most real-world-based types are syntheses of multiple dimensions, or "fields": a type representing a person, say, might include their name, date of birth, gender, marital status, address, phone numbers and other contact info, etc. Or, consider how we might lay out data for restaurant listings: **GPS** coordinates to locate restaurants on a map; street address; restaurants' names, hours of operation, and phone numbers; perhaps an indication of their relative priciness; perhaps a categorization of their style of cuising (French, Italian, Chinese, Japanese ...). The type might also have certain "flags" with pieces of information in a yes/no format: do they take reservations; are they wheelchair accessible; do they accept credit cards; do they serve alchohol. An overall "restaurant" type (say, $\mathfrak{r}$) would then be a "product" of these various fields.

The idea of *concepts* as multi-field complexes is also present in Conceptual Space Theory; many concepts are defined by a crossing of multiple dimensions, which collectively define the space of variations which their instances can occupy. The concept *juice*, for instance, would seem to be constituted by dimensions of color, taste, and perhaps liquidity (some juices are more viscous than others). Conceptual analysis can then proceed by isolating individual dimensions of various before investigating how they unify to create our impressions of conceptual similarity and dissimilarity, how concepts are refined or generalized, etc.

In formal types, fields have different structural or extensional properties which influence their conceptual status. One obvious criteria derives from the statistical distinction of *nominal*, *ordinal*, *interval*, and *ratio*. In describing a restaurant, say, the field representing "kind of cuisine" is presumably nominal, in that we identify certain terms like Italian, Japanese, etc., and assign the restaurant to one or another. This field probably has no "scale" or "metric" — French is not *more* or *less* than Chinese. A field representing *cost* may similarly be broken down into discrete options (inexpensive, moderate ...) but here there *is* a notion of scale (we can rank *moderate* as between *inexpensive* and *expensive*, say; and order restaurants from cheapest to costliest, or vice-versa). On the other hand, cost may also be figured by a metric such as the average price of a typical meal, which would be a straightforward, increasing, whole-number scale; prices can be ordered and degrees of difference calculated — two restaurants are similarly expensive if their average meal costs roughly the same amount. Meanwhile, geospatial coordinates represent a two-dimensional and (up to approximation) continous space which permits distances but not ordering, unless we are taking distance from one central point (e.g., someone looking for restaurantes close to their home). Hours of operation, for their part, cannot obviously be "ordered", though we can determine if a restaurant is open at a particular time; we can also rank establishments by how late they close, or how early they open — in principle, hours are cyclic, but when defining closing times we just need to consider the window of hours during the evening and night (respectively morning and afternoon for opening times).

For a hypothetical restaurant $\mathfrak{r}$ type, then, we can analyze their various fields in terms of their propensity for *ordering* and/or *distance*. We can say, that is, that some fields allow restaurants to be ranked in increasing or decreasing measures for some fields (e.g. average cost of a meal); and some fields permit the "difference" between restaurants, within the dimension of the field, to be quantified (average cost of a meal again, or location). Other fields allow for no particular comparison except for matching against single nominal values — unless we impose some metric whereby, say, Chinese and Japanese restaurants are deemed more similar to each other than to French or Italian, the most we can say is that two Chinese (etc.) restaurants are likely to be considered similar by virtue of both serving Chinese cuisine. Still other fields allow for different kind of comparison if someone is looking for a restaurant meeting some criteria — that it accepts reservations, say, or is open at 10p.m. on a weekday.

The statistical or qualitative structure of fields, along these lines, become implementationally significant if we seek to derive algorithms to match $\mathfrak{t}$-values to *queries* (say, to find a restaurant matching some customer's preferences), or to estimate whether $\mathfrak{t}$-values will be deemed similar or dissimilar (if someone likes one restaurant, an engine might look for "similar" restaurants to recommend). These requirements, then, translate to $\mathfrak{t}$-values as a whole: can we quantify (perhaps by a single distance metric) the degree of difference, or similarity, between two values? Can we order any collection of $\mathfrak{t}$s and, if so, ranked by which dimension? Can we search a collection of $\mathfrak{t}$s and find a list of values that meet some search criteria? To put this last question differently, how can we define parameters for searching $\mathfrak{t}$ collections? Do we create a hypothetical $\mathfrak{t}$ — say, an Italian restaurant open at 10p.m. that serves wine — and use that as a template for matching concrete values in the collection? Or do we create a different data structure, with a different type, aggregating search criteria against sets of $\mathfrak{t}$s? Should we, correlated with $\mathfrak{t}$, define a distinct type for searches yielding $\mathfrak{t}$s? In the case of restaurants, such a "search" type might specify a range of price-points, maximum geospatial distance from a central location, one or more kinds of cuisine, and so forth — replacing single fields in $\mathfrak{r}$ with ranges and bounds.

In some ways, these operations of ordering, querying, and measuring difference/similarity are consistent with Conceptual Space Theory: they capture how conceptual reasoning can be bound to quantitative possibilities, using quantitative relations — distances, orderings — to reason through concepts' extensions. On the other hand, such quantitative reasoning does not constitute a full-fledged reduction of these concepts to quasi-mathematical spaces — it is not, say, that quantitative fields in a restaurant $\mathfrak{r}$ type reveal how all conceptual details about restaurants can be reduced to numeric patterns. Instead, quantitative structures fall out as the result of *operationcs* on this type — operations like ordering, querying, or measuing similarity. I would argue that, as cognitive processes, sorting and comparing are more fundamental than construing relations numerically, although numeric patterns may arise organically in the manifestation of sorting/comparing cognitions. In short, the quantitative picture of (in this example) restaurants (or analogously I would say for many concepts) is derivative upon rational operations we perform *on* sets of concept-instances, more than latent mathematizations of an underlying conceptual space.

Analogously for formal types, many numeric structures come into play, not internally within those types own fields or structures, but in terms of operations performed *on* types — and particular on *collections* of $\mathfrak{t}$-instances, collections that can be ordered, queried, clustered by degrees of similarity, and so forth.

Suppose we have a restaurant database which tracks favorble reviews, assigning each restaurant a "grade" from, say, 0 to 100. Our $\mathfrak{r}$ type thereby has another scalar field, which can be combined, say, with an average-cost-of-meal, yielding a two-dimensional space which restaurants can mapped into. From the distribution of the resulting points, we could identify "good values" which are unusually highly-reviewed for their price-point, or outliers in the opposite direction where the review scores are lower than price would suggest.

In other words, a sample-space of restaurants mapped into the price/review space gives us a quantitative distribution, and our ability to compare restaurants in this way is doubtless a facet of restaurants' concept. But these quantitative details are only really salient when it comes to *comparing* restaurants, and stistically reviewing restaurant collections. The numeric structures are less conceptually foregrounded in our cognitive appraisals of any *single* restaurant. It is true that the quantified comparisons are possible via aspects which all restaurants have because of the their conceptual "package", so to speak; so mathematizable comparisons are latent in restaurants' internal conceptualizations. But these aspects only really become *quantitative* in the context of comparisons, whether these are explicit (e.g. analyses of a database) or more mental and informal. My judgment that a certain establishment is pricey, say, or cheap, inevitably results from a comparison (maybe subconscious) with other restaurants I have visited, or at least heard about.

The acknowledgement that quantitative structures thereby arise in the course of conceptualizing *restaurants* (in this case-study) does not accordingly demonstrate that our conceptual activity representing restaurants as cognitive acquaintences — as a feature of the world we roughly understand, for which the concept serves as an orchestrative tool — is at some fundamental level essentially mathematical. Instead, numeric spaces and axes emerge from mental operations layered on top of our basic restaurant-cognitions, particularly insofar as our reasoning turns from thinking about individual restaurants to their comparisons. The analogous phenomenon in formal type theory would be that quantitative models of types' distributions come to the fore in conjunction with operations for sorting and comparing type-instances. I will now examine these kinds of operations in more detail.

3.1 *Dimensional Analysis and Axiations*

Let us assume we direct attention to types in a $\mathbb{T}$ which are characterized by numerous distinct fields, and also that we are interested in procedures for ranking and comparing $\mathfrak{t}$-instances. We can then analyze fields on the basis of how they might contribute to such comparative operations. To have a sufficiently unspecific term, I will use the phrase *intratype comparisons* to refer collectively to various ordering, comparing, querying, clusters, and measuring-dissimilarity operations.

A $\mathfrak{t}$s fields can then be classified according to how they may contribute to intratype comparisons. In the abstract, such an analysis would be provisional, because certain type-implications may have their own peculiarities. For instance, it is reasonable to say that a restaurants' *name* is not a factor in estimating similarity: two restaurants with similar names are not especially likely to be similar in other respects. However, we can envision scenarios where textual similarity *would* be taken into account (e.g., a search engine trying to accommodate spelling errors). Or, consider the question I mentioned earlier as to whether factors like Chinese/Japanese or French/Italian similarities (as styles of cuisine) should be modeled so as, in effect, to yield a distance metric on a nominal "kind of cuisine" dimension. These examples show that anticipating exactly how clustering or distance algorithms would be designed, in any concrete case, is rather speculative. Nevertheless, I think it is possible to make some broad claims about how data fields *usually* work in the intertype-comparison context.

On that basis, then, I propose to distinguish five rough sorts of data fields, as follows:

1. Digital/Internal fields: Computational artifacts, such as globally unique identifiers, which are employed by code managing type-instances behind the scence but do not typically embody real-world concepts related to the type, and are not typically salient in intertype comparisons.

2. Textual fields: Natural Language artifacts such as names and descriptions, which would not normally be used directly for intertype comparisons. Ordering or measuring similarity on collections of textual contents tends to be difficult, or to have little actually conceptual resonance, unless some sort of Natural Language Processing is used to extract structured data. For example, there is probably little conceptual significance attached to (dis)similarity or ordering among restaurant names, except perhaps if it is desired to list restaurants alphabetically (which in any case is a presentational matter more than a comparative one).

3. "Axiatropic" fields: I use this term to represent any fields which have nominal, statistical, scalar, or in any sense quantitative qualities that can be leveraged for comparisons. I include nominal fields (enumerations) because these are relevant to similarity — consider two restaurants both labeled "Chinese" — and also nominal dimensions can sometimes have extra comparative structure (e.g., an inexpensive-moderate-expensive scale is ordered by increasing cost). Other than enumerations, I define axiatropic fields as any dimension with numeric values where numeric properties are consequential for ordering or for measuring similarity — excluding, say, numeric id's where numbering has not structural meaning other than uniqueness, but including "locally" ordered scales like time points, as well as "globally" ordered dimensions such as integer magnitudes (e.g., prices), and spaces which have no particular ordering but which can be ordered via distances (like geospatial locations). Axiatropic fields can be seen as "axes" to which type-instances can be project, and the union of $\mathfrak{t}$s axiatropic fields, which I propose to call $\mathfrak{t}$s *axiatropic structure*, defines a multidimensional space wherein $\mathfrak{t}$s are mapped to individual points. The tuple of values obtained from these fields I call an *axiatrope*, and the points to which a given $\mathfrak{t}$-axiatrope projects I call an *axiatropic image*. If desired,

axes can be annotated with details such as valid ranges and units of measurement (consistent with, for instance, Conceptual Space Markup Language).

4. Flags: I separate out boolean values — along the lines of whether a restaurant is wheelchair accessible, or takes reservations — because these pieces of information are more likely to be used to match candidate values against criteria than for comparisons between values. This does not preclude some similarity algorithm from ranking, say, two restaurants that serve liquor, or take reservations, as a little more alike — i.e., these data points may add to a metric of similarity (or inversely subtract from a metric of difference) when they agree between instances, or contrariwise when they disagree. However, I would argue that conceptually these kinds of factors are more pertinent to building a sufficiently detailed picture of an instance than to intratype comparisons; on that premise I distinguish "flag fields" as a separate field grouping.

5. "Mereotropic" fields: These fields represent collections of values (lists, tuples, and so forth) rather than single values. In general, tuples are less conducive to direct comparison, without further analysis — say, comparing two students' grades by comparing their average; or comparing two lists by counting the elements they have in common. Similarity metrics, then, can employ collections fields, but the calculations are more involved than just projecting type instances onto points in a mathematical space. I leave open the possibility that collections may have elements that are themselves collections, so that mereotropic fields can give rise to "mereological", or part-whole, hierarchical structures.

These different genre of fields are reflection in different compartments to a type's interface; to this list I would then add the portion of the interface related to constructing t-instances: constructors, co-consructors, and related functionality for testing the validity of a data structure and/or "deconstructing" values into a construction pattern. Collectively I will refer to this last aspect — which does not involve fields *per se* — ts *constructive interface* or (with apologies if my neologisms are getting a little heavy-handed) its *nomotropic interface*. The fields (textual, flags, binary) which are neither axiatropic nor collections-based I will call *endotropic*. I will then call procedures related to restructuring or re-presenting t-values for analysis, **GUI** or visual rendering, serialization and deserialization, or database persistence, as collectively a *morphotropic* interface. We then have a partition of types' interface into five facets: nomotropic, axiatopic, morphotropic, endotrophic, mereotropic.

With respect to the portions of an interface which *do* involve data fields, certain elements of this overall model are consistent with and informed by Conceptual Space theory, but its practical applications are oriented more toward computational operations on multiple values. Specifically, this aspect of types' implementations is focused on operations wherein t-instances are compared to one another. The idea is not to use a "theory of fields" to uncover underlying conceptual patterns linking formal types t to real-world phenomena. The concept *field* itself may imply an operational manipulation of type-instances: a computation may be described or utilized as a field even if it is not technically provided among the tuple of values via which a *v* is registered in memory. For instance, the average value of a collections field can be considered a data point on t even if it has to be dynamically computed.

In effect, then, t data points which are relevant to comparing or querying ts are facets of t that need to be deliberately engineered. Implementers choose which t details to make accessible to external code; they may also choose to provide dynamic calculations that provide information about t values (sometimes called *views*), such as the average of a student's grades, even if this data is not internally tracked by the ts binary layout. Implementers, in short, have to anticipate and provide procedures for client code — sofware using the type implementation — to interact with ts fields: iterating over collections, dynamically calculating views, querying against a prototype, ordering on one or another field, etc. These considerations inform the process of designing an *interface* for t Overall, a *ty* interface covers various tasks, such as constructing t instances in the first place, or itegrating ts with other kinds of software components (serializing and deserializing ts for data sharing; sending ts to a database; showing ts in a **GUI**; etc.). I will use the term *paratropic interface* to that portion of a ts interface which concerns sorting, comparing, and querying ts (or sets of ts).

In practice, the field-classification I proposed earlier would most likely be applicable to a type's "paratropic interface": it would help implementers reason about what data points to expose for a t and how precisely to set up the logistics for obtaining this data from t values. Taken in conjunction with the principle that formal types are (often) modeling artifacts which approximate real-world concepts, this discussion then suggests that such conceptual goals are mostly operative in the interface *to* types. That is, when considering how to use formal artifacts to proxy real-world, human concepts, the point is not only to assemble a list of particular details to keep track of (for a restaurant: name, location, cost, etc.). A type's effectiveness in representing human concepts is equally dependent on a programming interface which allows digital operations to be performed; operations which reflect how we conceptualize real-world phenomena, including by actions of ranking and comparing instances of the same concept.

To the degree that Conceptual Space Theory has an intuitive role to play type semantics, then, I would argue that this role is chiefly manifest in the realm of types' *interface design*, in the procedures defined *for* types, rather than in the internal mathematical/computational structure of type-instances qua digital constructions.

I think this perspective is also consistent with my earlier analysis from a natural-language perspective: a conceptual account of *nouns*, in particular, should be oriented in the pragmatic employments of concepts as cognitive tools: the idea that we mentally *do things with* concepts. Perhaps there is a certain correlation between the idea that formal types' are conceptually defined by their *interface* and that, cognitively, concepts (noun concepts in particular) are constituted essentially by their intellectual-functional roles (or, at least, so I will now consider).

3.2 *Concepts as Functional Tools*

Earlier I argued that judgments of conceptual appropriateness are typically driven by pragmatic concerns — deeming rain as *pouring*, for instance, is a useful characterization if it implies a response consistent with how most people would behave in pouring rain (as opposed to, say, drizzling rain). In some cases — as with drizzle/pour — we might find a quantitative dimension which models the peaks and nadirs of concepts' applicability, but the quantitative picture is most relevant when it implies a *qualitative* difference. Along the scale of rain intensity, conceptualizing rain as *pouring* or *drizzling* will have qualitatively different implications.

I centered my earlier discussion on verbs and prepositions: using the verb *pour* or *drizzle*, or say *on* vs. *all over* for *dishes on the table*, reflect different situational consturals which are appropriate to the degree that they are useful, in organizing our response to situations or anticipating that of others. Choosing the phrase, e.g., *dishes all over the table* signals the speaker's framing the circumstance, a semantic cue which will be deemed appropriate if it engenders valid follow-up reasoning; e.g. if it implies that it will take some effort to clear the dishes, and such is indeed the case. Word-choices like *pour*/*drizzle* or *on*/*all over* imply a conceptualization which people will tend to judge *appropriate* if they are practically *useful* glosses on states of affairs.

Presenting this thesis in the context of verb and prepositions, I would argue that a similar analysis can be given for nouns: subsuming an entity under a given noun-concept is admissable if doing so gleans useful hints as to the entity's properties or dispositions. Calling something a restaurant is useful if the practical implications — that someone can go there to order and consume food, etc. — are appropriate. By choosing the word *restaurant*, we hear the speaker as implying certain practical details appertaning to the thereby categorized establishment, and we are wont to deem the concept-attribution accurate or not depending on how well those practical expectations are borne out. Calling a food court stall *restaurant* is dubious because it raises expectations that are unwarranted.

In the case of verbs, concept-attributions are canonically assessed in the context of specific situations, because verbs typically profile discrete events. If I say *we ran for the bus*, the word-choice *ran* packages several components expected to be part of the relevant circumstance, e.g. that some thing or person/people were *doing* the running, as a conscious choice, were running *to* some destination, and were moving with non-ordinary speed. The characterization invites various addenda: why were the rushing? Where were they running to? Who was running? If these questions are not sensible in the described situations' context, the *run* choice of words was probably ill-conceived (and the corresponding conceptualizing of the situation would be deemed flawed). Or, citing *pouring* rain would imply that people would or should seriously try to avoid being exposed to that weather, that it could potentially be dangerous, and so forth: if in fact the rain is not strong enough to elicit that response from other people, then *pour* is not a useful concept to apply to the situation; it does not have realistic predicative implications.

In the case of nouns, estimations of applicability may have a similar overall structure but are less oriented to individual events, and more toward *potential* or *typical* events that we can anticipate vis-à-vis the target of a conceptualization. Announcing that a restaurant is conceptualized as such commits us to predicating the pattern of various kinds of events that would commonly occur in conjunction with the restaurant: people being seated at tables, ordering food, having the food served to them, etc. If these are not the events evidenced in how typical situations unfold — e.g. a food court stall would instead characteristically have people find their own tables and carry their orders themselves — then the proposed concept is of limited applicability.

The worthiness of a concept-attribution, in short, can be measured by how accurately the concept leads us on to further strategies for learning about, or responding to, the things, events, or situations covered by the concept. If we are conceptualizing an event — that someone ran, that it rained — the concept we apply (signaled by the verb we choose if talking about it) unlocks a passage to uncovering more details.

In our "inner", or "subconscius" thoughts, conceptualization mentally gear ourselves for where to then turn our attention, if we seek to glean more information, or how to respond pragmatically. If concepts map to words, and we hear or speak of conceptualizations interpersonally, recognizing the concepts others apply cues us to how *they* are disposed to act, and also opens a shared information space where both parties anticipate how to query each others' conceptualizations.

For natural language semantics, concepts' intersubjective dimension should be foregrounded, even though it is based on each persons' internal, probably partly subconscious ideations. Accordingly, then, we should emphasize how conceptualizations, signfified by language, "open up" a dialogic space. When conceptual attitudes profile an event, their linguistic expression puts conversants on the threshold of co-responsive process where parties tease out details of other's situational framings. The dynamics of this process depend on context. If one person knows the most about a situation — maybe they are telling a story — the interpersonal epistemics would be oriented to those learning of the situation at one remove steering the former party to provide information the other find relevant. If multiple parties were all equally proximate to a situation, the dialog could be more about syncing their respective interpretations — the how, why, and so-what which profile scenarios toward finding optimal pragmatic responses.

Noun-concepts add the complication that they are not necessarily employed in conjunction with framings of specific situations. Of course, negotiations can apply to nouns also:

- (40) We were delayed in traffic because we got stuck behind a truck.
- (41) Well, it wasn't really a truck, it was more like a van, but it kept slowing down at the intersections.
- (42) Well, we didn't get stuck for long, but we pulled off onto a side street so it took longer to get here.

In this case the speakers are debating with each other over which conceptual presentation best captures the situational details; a dialectic that can be evidenced in negotiations over noun-acceptability (as in (2)) as well as verb-acceptability (3). Analogous collaborative framing-choices could play out in immediate, pending situations where conversants have to act in consort:

- (43) Should we stop at that restaurant? The kids are starting to act up.
- (44) That's not really a restaurant — I'd rather wait until we can sit down for lunch.
- (45) They're not really acting up — I think they'll be OK for another hour or so.

Counter to these "in the moment" examples, though, noun-concepts can be negotiated outside any immediate situation:

- (46) The best place to eat in this neighborhood isn't actually a restaurant; it's a health-food store where they have a seating area in the basement.
- (47) Experts are predicting a recession next year, which could affect the elections.
- (48) They're not really predicting a recession, just a slowdown.

These hypothetical discourses could likely occur in situations where the speakers do not have an immediate need to act, but are sharing "background knowlege". We use language not only to coordinate responses to pressing circumstances, but to asssess and enrich our collective knowledge by sharing beliefs and ideas — it is useful if friends, neighbors, coworkers, etc., share a robust stock of propositional attitudes so as to facilitate collective action when it *does* become necessary.

But whether oriented to immediate or to generic, hypothetical, or future situations, conceptualizations are linguistically and cognitively salient largely because of how they open the space for new rational activity. In language, my grasping how another language-user conceptually frames a given event or entity provides a data point from which I can then respond, pursuing ever refined mutual understanding and co-ordination: do their conceptualizations agree with mine? Do they have knowledge that I should access? Do their conceptualizations spur a propensity to act in ways I can anticipate and/or reciprocate? Are there differences in beliefs or opinions that we can reconcile?

Conversation, of course, cycles through a large number of conceptualizations over the course of several turns of dialog, or even of one sentence. Each concept may play a role in the subsequent course of that dynamic, because how we conceptualize something is a window, for others, onto our thought-processes and dispositions. Concepts as cognitive tools, in short, play out largely in this intersubjective, co-intellectual dynamic, marked by our "theory of other minds". Consequently, we should view theories of concepts which are too "solipsistic", or treat concepts only in the confines of one person's cognition, as at best incomplete. This would include theories that model concepts around prototypes or exemplars — a paragon "restaurant idea" that grounds the concept by offering a point of comparison for actual restaurants. It would also cover theories that treat concepts as instramental logical conjuncts, a semantic frame of qualities to check off: x is a restaurant if it serves food, as a commercial activity, with table service, with a menu, with a fixed location, etc. In particular, uncomplicated encodings of conceptual structures within logicomathematical frameworks have to be taken with a grain of salt.

Assuming the gist of this analysis, how then *should* an "intersubjective" theory of concepts affect our appraisal of formal types and digital artifacts being, in part, technical representations of concepts as they operate in human affairs and natural language?

3.3 *Hypergraph Grammar for Natural and Formal Languages*

I have argued that conceptualizations in (and expressed via) natural language are cognitively significant because they spur subsequent discourse and reasoning: mentally deciding (or hearing from someone else) that something or some situations falls under some conceptual rubrick is a stepping stone to further mental or practical activity. This may be purposeful behaviors enacted as the situation unfolds, or less tangible actions in the discursive space of a dialog, or intramental operations wherein new beliefs are formed, or choices are made about where to apply attentional focus. But whether in behavior, language, or cognition, concept-attributions acquire their full meaning only through subsequent operations which become available or reasonable by virtue of it being accepted, or posited, that some event or entity is attributable by some corresponding concept.

The formal analog to this theory is clearly that types' are semantically operationalized, as I argued earlier in this section, by procedures defined on them rather than by their precise layout of data fields. The purpose of a type, qua digital artifact, is not just to collate a tuple of values. It is rather to package multiple values into an operationally beneficial whole. Each t's utility derives from procedural resources that come into play via t's inplementation, not merely from the values t-instances carries with them.

This means that types' formal semantics derives in part from procedures defined on them — i.e., procedures which input and/or output t values. Such a proposition may appear to make types' semantics open-ended, because most programming environments do not restrict where and when typical procedures involve each t may be defined. It would not make much sense to say that types' semantics are so closely bound to any and all procedures such that simply defining any function that takes or returns a t converts t to a different type; that would render notions of type-identity almost meaningless.

However, it *is* reasonable to isolate a group of procedures as an *interface* to t, which are then constitutive of t's semantics: modifying the interface — by adding, or changing the signature of, an interface procedure — *does* make t into a new type. Types' identities are therefore bound to the specific collection of procedures which are declared to be operationally "internal" to t; which provide the basic functionality through which other sorts of actions involving t can be defined.

Some coding styles offer a construction to transparently demarcate these core, interface procedures from the others. The obvious example is Object-Orientation: in most Object-Oriented languages, the *interface* to a *class* (a *type* whose values are Objects) is constituted by those procedures ("methods") which take an instance as a privileged "**this**" value, symantically separated out from and treatedly differently than ordinary function parameters — in the context of "channels" I say that methods have a separate channel for "message receiver" values (using Object-Oriented terminology), or a "sigma" channel (using terms from "Sigma Calculus", an Object-Oriented extension to "Lambda Calculus").

For a general model of type-systems, however, we cannot say for sure that some semantic or channel-based mechanism will force procedures to be classified as within a type interface or not; at most we can stipulate that interface procedures may be explicitly declared as such. Accordingly, assume that any $\mathbb{T}$ can denote, for each t it covers, that a procedure which inputs and/or outputs a t is somehow "core" and part of ts interface. Conceptually, such procedures provide the foundation for digitally or computationally manipulating t-instances, so they reveal a conceptual model of t's purpose — the modeling agenda behind t being implemented as such can be revealed through the choice of procedures included in t's interface.

The semantics of types, then, is essentially revealed through the semantics of their interface procedures. This points to the larger observation that the semantics of computer code in general is predominantly the semantics of procedures: almost all code, if not literally all code, in mainstream programming languages, is code describing (or effectuating) the implementation of procedures. Moreover, almost all code in the body of a procedure describes call to *other* procedures. As such, the most important criteria for defining programming languages' syntax and semantics is to describe procedure bodies (to demarcate sequences of statements or operations enacting each procedure) and procedure calls (how one procedure achieves its rationale by calling to other procedures in some structured fashion).

Modeling inter-procedure calls is especially amenable to a graph-based representation because individual nodes can represent external procedures to invoke, during the execution of the current procedure. If we process code via graphs — i.e., "code graphs" — then each procedure's implementation can encompass its own code graph; some nodes in that graph then represent other procedures, with their own graphs. There is not necessarily a one-to-one relationship between nodes in one procedure and other procedures the first one calls: sometimes a source-code token, for instance, provides a name for a function or method which may actually be mapped to several different procedures at runtime. Subclasses can override the methods of the class they inherit from; also generic code can be reused across multiple types. As a result, a single code-graph node may not embody one specific procedure, but rather be a placeholder for a variety of possible procedures for which the proper target is dynamically selected when the enclosing procedure executes, given a specific set of input parameters.

What *is* fixed within a code-graph, however, is how the nodes standing for external procedures connect to surrounding nodes which supply onputs to those procedures, or hold their outputs. Almost all the structure of code-graphs derives from their marking how some nodes carry values which become inputs to inter-procedure calls, with edges linking those nodes to the central node representing that external procedure itself. In the "Channel Algebra" terms I proposed in [**?**], code-graphs modeling procedure implementations comprise (in general) multiple *channel package*, each package being a graph structure centered on one "ground" node which designates an external procedure, and other nodes linked to it which represent how the channels carrying data between the two procedures are to be populated with values from the original node's current environment. Code-graphs on this approach are hypergraphs because channels group multiple edges (specifically those linking "carriers" to ground nodes in a channel package), representing an extra layer of graph structure (as well as the fact that graph-nodes represent typed values which may internally contain a tuple of further values; i.e. they are hypernodes spanning a tuple of hyponodes).

Programming language grammars, in turn, can then be analyzed as systems for mapping surface-level computer code to "channelized", hypergraph-based code-graph representations. Grammars need to document how source code indicates which tokens represent "ground" nodes in a channel package (i.e., represent procedures to call); which tokens represent input or output values and their corresponding ground node; which channels the carrier-to-ground links belong to; and how carriers in the context of one procedure-call pass values to carriers in the context of subsequent procedure calls. In short, an adequate grammar for a programming language has to engender compilers that transform source code to code-graphs comprised of *channel packages*. Insofar as the underlying "channelized" graph model is based on hypergraphs, this criteria effectively states that programming languages need hypergraph grammars.[12]

Note that my form for hypegraph structures in this analysis is divergent from Hypergraph Categories as originally proposed for integration with Conceptual Space semantics. Instead of hyperedges linking two procedures, I situate graphs primarily within the implementation of each single procedure. Hyperedges then, most commonly, represent links between *carriers* and *ground* nodes; they represent the relation whereby a procedure call draws from a surrounding value to pass as a parameter to an external procedure. In practice, however, the two notations can coincide insofar as (according to a common pattern) one procedure call is provisional to a second; so the output acquired from a former call becomes mapped — internally in the calling procedure's environment — to input for the subsequent call.

The mashup of hypergraphs and Conceptual Spaces — what I earlier dubbed "**HCS**" — aims to web a hypergraph-based grammar to a Conceptual Space semantics. Here I have provided at least a provisional outline of a Hypergraph Grammar framework for programming languages; also ideas on formal type semantics vis-à-vis programming language type systems, noting some similarities and contrasts with Conceptual Space theory. Plus I outlined my case for which directions best capture the theoretical strengths of Conceptual Space semantics in natural language.

Against this backdrop, I have largely neglected what would be the fourth leg of a double-hybrid framework combining hypegraph grammars and conceptual space within both formal and natural language — I have not discussed whether my just-outlined "Channelized Hypegraph" model, formulated in the programming-language context, has any applicability to natural language. I will offer some comments on that subject next.

3.3.1 Hypergraph Grammars and Natural Language

So far I have argued that programming language grammars should be rooted in the foundationally procedural nature of computer code — the building-blocks of any self-contained stretch of source code are constructions of calls to one procedure from the context of the second. In graph representations, once a node is marked as identifying the procedure to call, the essential structure of the surrounding

[12] The point is not that all languages' formal specifications are hypergraph-based — or that all compilers should or do use hypergraph code representations — but rather that all languages's code *can* be modeled via Channelized Hypergraphs and that all languages thereby *can* be modeled via hypergraph grammars.

code is how other nodes supply values to pass to the marked procedure as inputs. Inter-procedure calls are therefore the prime semantic constituents of program behaviors, so that programming languages' syntactic rules are (directly or indirectly) governed by the need for inter-procedure calls to be unambiguously constructed. The syntactic forms which these languages exhibit, that is, can be explained by our need to use these languages to implement procedures by calling other procedures, identifying the procedures to call and the symbols carrying their inputs and outputs, in a transparent fashion which computers can mechanically react to.

So the basic building blocks of programming languages' "discourse" are "statements" which involve one procedure calling another, possibly using results obtain from calling prior procedures as intermediate values. Statements are structural analogs in this context to sentences in natural language, although the analogy may not seem very substantive — after all, it is hard to see in what sense a *sentence* could be deemed a "procedure call". Still, I *will* pursue this analogy to some extent, not necessarily on the actual sentence level but on the level of sentences' smaller constituents.

Here I have endorsed a generally cognitive-linguistis methodology more than a reductively "logical" model of language, one which discounts (natural) language nuance and contextuality. I disavowed what might be called "logistical" models of concepts, where concepts could be treated as artifacts of logic, and entertaining concepts — or deeming concepts attributable to their tokens — could proceed as an essentially logical operation: how close is a candidate bearer to a conceptual prototype; how many "boxes" does the candidate "check off". I believe instead that conceptualization is more *interpretive*: to ascribe a concept to a bearer is to offer (internally in one's mind, or externally through language) an interpretation of that (object, situation, or event) which is conceptualized.

On this perspective, when we communicate conceptualizations via language we are also signifying *interpretations*; and the syntactic and semantic strata of language can be studied through the lens of how elements of language contribute to this process. Each word in a sentence, for example, adds some detail to interpretation conveyed by the whole. Such a principle has ramifications for both semantics and grammar. Semantically, the meanings of words can be understood in the context of the interpretive additions or alterations they contibute to other words in their context. Syntactically, the structures of language can be seen as vehicles to mark which words are, most directly, interpretively linked to other words.

When we say, for instance:

- (49) A friendly dog was behind the gate barking and wagging her tail.
- (50) My neighbor walks that dog every morning.
- (51) The store around the corner is closed, so we have to walk a few blocks out of the way.
- (52) We drove along the scenic drive by the lakeshore.
- (53) We drove the scenic lakeshore drive.
- (54) We drove to the lake shore.

we are using implicitly connected pairs or groups of words to paint various interpretive pictures, with one word adding interpretive shading to the other(s). The obvious case is in (1), interpreting the dog as *friendly*. But describing the dog's action (and the tail's movement) as *wagging* is also an interpretation. Likewise characterizing a store as *closed* (which could mean either closed for the night or shut down permanently) is an interpretation of its current state.

Those hearing sentences like (say) (1)-(5) likewise have to glean the speaker's interpretative attitudes (I'm deliberately playing off the philosophical *proposiontal attitudes*) — which is itself an interpretation. Moreover, the addressee's interpretations are driven by a combination of linguistic and extra-linguistic, or situational, givens. The verb *walk*, e.g., calls for a different interpretation in (2) than in (3): in the former the walk is not a fixed path to a destination, whereas the latter profiles the endpoint of the activity more than the process. A similar dynamic defines the contrast between (4)-(5) and (6).

However, these interpretive variations are not fully specified by the language itself; we cannot very well account for the different kinds of situations represented in these sentences by examining words by themselves. A lot of this interpretation instead comes from background knowledge: that people routinely walk dogs (typically with no set destination) for the dogs to get exercise and relieve themseves; that people like to drive on scenic roads so as to appreciate the scenery, separate and apart from their desire to get to destination; that people in urban areas use the term "block" as an indicator of distance (as well as for profiling a spatial region as in *around the block* or *down the block*). These *usages* are not really explicated by considering just word-senses, or discrete lexical entries; it is more that words become entrenched in specific roles for discussing specific kinds of situations.

So the point is not that *walk* as in *walk the dog* is a different (or, indeed, the same) lexified concept as in *walk to the store*. Instead, conventional usage primes us to certain discursive habits which transcend notions of sameness or difference among word-senses. Both *walk the dog* and *walk to the store* are conceptual packages which involve more than the *walk* verb alone. The enunciation of "walk" *in its specific* *context* calls forth the entirety of those conceptual tableux: so when we hear *walk* in *walk the dog* we instinctively intuit a pre-formed cognitive framing; a kind of mental "script" that spells out the prototypical situation of someone walking a dog, why they do so and their typical behaviors in the process. Likewise for *walk to the store*. Likewise also, the concept of driving *to a destination* brings up a cognitive script somewhat different than driving to observe the scenery.

Words, on this theory, do not usually signify logical constructs; rather, they are proxies for situational gestalts, selected to bring forth in the addressee's mind an interpretive framing consistent with the speakers' own. As a competent participant in dialog — and a rational member of a relevant language community — our fellow conversants assume that we have a stock of many situational "scripts" and that they can trigger the appropriate construal in our minds by their choice of words, and of phrase-structures. Such scripts, moreover, are (at least to some degree) extralinguistic — they emerge from our background knowledge and overall ability to craft purposeful behavior in the face of a present situation, rather (in general) than from purely linguistic competence with syntax or semantics.

Properly understanding language is therefore a matter of parsing sentences so that, in response to both semantic (e.g., word-choice) and syntactic (e.g., phrase-structural) data, we can call forth the proper interpretive script for each received linguistic performances (the scripts which best recapitulate speakers' own interpretations). I believe that this theory does, now, actually begin to resonate with my earlier gloss on programming-language grammars. In that context I argued that the key responsibilities of grammars were to build packages of values (or "carriers") and "ground nodes" marking the call to a procedure, in the context of a different procedure. In the current, natural language setting, I would say that the role of *natural* language grammar is to package words — together with their interword links and larger phrasal units, if applicable — so as to map from linguistic givens to interpretive "scripts", selected from a vast warehouse of situational prototypes and interpretive dispositions we are assumed to command, as competent language-users. Interpretive scripts here play a role analogous to computer languages' procedures.

Interpretations are called forth on a smaller scale, not just a whole sentence — the predication *friendly dog*, say, does more than just attribute the quality of friendliness to a canine. It is more even than a prototypical "friendly dog" which anchors the script. Instead it is something like a concept that is narrativized, that is cognitively entrenched not just as a mental picture or observed pattern but as a recipe for pragmatic action: conceptualizing the dog as *friendly* primes me to behave toward her in certain ways, including bearing certain expectations of her own behavior.

Moreover, language does not have sufficient logical transparency to allow an independently meaningful semantic substratum independent of cognitive and pragmatic concerns ("pragmatics" here in the linguistic sense). The fact that *friendly dog* is not really just a predicate structure is one example. A related example comes from one of the introductory presentations for **CSML**, where the notions of a "contrast class" is demonstrated through the case of *large chihuahua*, which would still be a *small* dog [**?**, page 3]. Contrast classes are introduced as a semantic device to redress the supposed anomaly wherein *chihuahuas are dogs* does not entail to *large chihuahuas are large dogs* (to be sure, there is much literature on these "contrastive semantics" problems in linguistics). But this is only a problem if we expect that people receive language segments as if they were propositional forms, where rules of logical deduction or substitution apply.

In truth, language is much more illogical than that. Even simple adjective-noun pairings are received as condensates of some conventualized subconcept or situational template more than a straightforward logical predication. We have a "large dog" script no less than a "friendly dog" script. Dogs we would conceptualize as *large* are not even, necessarily, larger than the median or mean; the conventionalized usage has *large* essentially meaning any breed other than truly diminutive pets bred over generations for that tiny size. There is probably also a social dynamic in play; dog owners appear to classify themselves (and one another) as "small dog" or "large dog" lovers. A 40 pound pit bull and 120 pound mastif would both be called *large* in part because the pit bull's owner is more likely to think of herself as the kind of person that is drawn to "large" dogs.

Obviously, linguists (much less logicians) do not do sociological surveys of dog owners, so these usage pattern do not really manifest as logical rationales for linguistic predication. That's the point; our instinct as languages users is to search for entrenched conventions rather than to parse phrases as novel logical combinations, so we decode *large dog* through whatever mental script best captures the entrenched usage, whether or not these are actually logically accurate. Dog owners would not start calling pit bulls "small" even if it were statistically demonstrated that the average pit bull were actually on the small end of the dog-size scale.

Nor is this actually being stubbornly irrational: the reason why we are interested in a dog's size is usually because it indirectly implies how the dog might behave. A pit bull should be deemed large insofar as it is more likely to act similarly to an actual large dog than to a chihuahua. Likewise a large chihuahua will still act like a small dog, which is more important in most practical context than the fact that it *is* small in a statistical sense. This is the same pattern as I analyzed vis-à-vis *friendly dog*: we do not intrinsically read this as a logical unit, but as a compound signification targeting some script most people have sketching out how friendly dogs behave and how most people then respond.

These anticipations are not intrinsic to the notion of friendliness, so they are not simply etched out in the word's lexical resonances. We can, for example, expect that a dog's friendliness will be manifest by their propensity to lick our ankles or sniff around our waist. Those are not behaviors we would associate with friendliness in other environments — say, when we describe a shopkeeper as "friendly", or a politician as friendly to advocates of a political cause. Each use of "friendly" bears a distinct implication as to what the predicate implies for the bearer's behavior *in that context*.

The upshot is that interpretive scripts need to be selected for — and are solicited by — even small-scale linguistic units, like a word-pair (e.g., adjective-to-noun). These smaller interpretive stages get absorbed into larger interpretive processes; see how *friendly dog* becomes a unit in a spatial and descriptive scene in (1). So — insofar as natural language grammar describes how syntax (and word morphology) connects words together (as adjective to noun, verb to subject, subject to object, and so forth) — we can see grammar as stating the rules by which we identify some words as interpretive modifiers to others, their fact of being thereby linked spurring a cognitive sifting for the optimal interpretive procedure reciprocating speakers' intent.

This is still a rudimentary analysis, and elsewhere (in a paper accompanying this one in the software demo) I have developed these ideas more substantially, through a theory I described as "Cognitive Transform Grammar". The minimal intuition of that theory is that grammar describes how words should be paired up (or more preceisely grammar describes syntactic formations that convey intended pairings), and that any pairing profiles a cognitive, or interpretive, process; whereby one word (a "modifier") somehow alters our constural of (the signified or referent of) its partner (a "ground"). Moreover, indirectly, the modifier transforms our overall appraisal of a situation by staging a focused modification, or revised interpretation, of the ground — passing from *dog* to *friendly dog* both refines our conceptualization of the dog in question *and* adds an important shading to our understanding of the overall circumstance spanned by our interaction with the dog.

I also think that grammar formalisms based on hypergraphs — and something like a theory of channels — can apply in the natural language context. One reason is that interpretive juxtapositions between words can draw in further removed syntactic contexts, refining the "local" interpretive process. Consider the difference between:

- (55) A few times we passed friendly dogs who barked at us.
- (56) We passed friendly dogs who barked at us a few times.

Here (1) is more likely heard as identifying *different* groups of dogs, with a "friendly dogs barking" scenario repeated for several occasions; while (2) more likely reads as only naming *one* such occasion. The phrase *friendly dogs* is interpreted differently depending on how we read the scope of the *a few times* quantifier, and whether we the scenario is understood as specific or as generic and replicated. I would argue that this constitutes contextual data that gets pulled in alongside the interpretive effect of *friendly* on *dogs*, as if following a distinct channel for interpretive nuance alongside the "local channel" of co-modifying word pairs. These extra "non-local" channels can be modeled as an overlay structure, i.e. a hypergraph, above word-pair graphs which would be the primary syntactic representatums for graph-based grammars, e.g., Dependency Grammar.

But the key point of my analysis is that linguistic formations have to trigger interpretive processes which are, at least in part, extralinguistic. This influences how we should see the role of natural language grammar, because the point of syntax is not to describe phrase constructions which, by their internal logical pattern — e.g. by appending the property *friendly* to *a dog* — fully explicate an intended signification. Rather, syntax is ordered to guide addressees, within their response to linguistic content, toward extralinguistic cognitions — the *friendly dogs* phrase being semantically complete not by logically predicating "friendliness" but by being received as in instruction to load up the mental frame, the cognitive anticipation-package, of *friendly dogs*, within suitably typical ambient situations.

In short, this theory builds off the idea that semantis is, at root, cognitively procedural; and that syntax thereby is a process of forming aggregates whose pattern triggers the right procedure, rather than a logically self-contained signification. This, I would argue, is quite consistent with formal grammars organized around the idea that formal languages — or at least prorgramming languages — are, themselves, at a semantic and pragmatic level, intrinsically procedural; although the notion of "procedure" in a computational context is obviously different from cognitive or interpretive procedures. Modulo the diversity in senses of "procedure", we can still say that procedure-oriented semantics retroactively compels procedure-oriented grammar, and that this trend cuts across both formal and natural language grammars.

Here then is a case for unifying hypergraph grammars across the formal/natural delineation, to complement the projection of a Conceptual Space Theory similarly straddling that line. I have addressed, then, some points of similarity and difference between grammar and semantics in both formal and natural-language contexts, from the perspective of a Hypergraph/Conceptual Space unification. To what degree do these individual comparisons add up to a plausible framework for combining Hypergraphs and Conceptual Space Theory in both formal and natural contexts?

3.3.2 Limitations of Conceptual Space Semantics

Because a lot of the substance of natural language semantics, I would argue, is *extra*-linguistic, any semantic formalism has only limited value against languages' complexities and nuances. I think this applies to the core Gärdenfors Conceptual Space Theory, but also to any other *a priori* semantic methodology. The only non-reductive option for truly unpack semantic structures is to take each sentence, each language artifact, on a case-by-case basis, explaining the interpretive fiats and contextual idiosyncracies of every usage. General theories can still be guiding framework for such case-by-case analysis, however, and Conceptual Spaces are certainly one of the devices which belong in the linguists' and philosopher of langage's toolkit.

For formal types, I have argued that Conceptual Spaces are most relevant when it comes to interface design, and in particular to that portion of a types' interface I called *axiatropic*, where intra-type comparisons can be ordered and/or measured. The notion of partioning a types' "space" of potential instances is also relevant, I argued, to reason about types' constructors and construction patterns — to their "constructive interface". More generally, the notion of types' *interface* being internally organized, with different parts exposing different facets of types' operational and conceptual rationales, is itself perhaps an embodiment of types being understood in some sense as "conceptual spaces". Not every interface facet may involve quantitative dimensions, but interface design does reveal the workings of human conceptualization crafting the type as an organized space, a technical artifact which encompasses and renders computationally tractable a spectrum of values.

I would argue, then, that a systematic theory of interface design would be a reasonable formal application of Conceptual Space theory. To some degree this is already hinted at in existing sofware-oriented tools based on Conceptual Spaces, such as **CSML**; my "**NEMAM**" model of type-interface facets can legitimately be presented as a superset of the **CSML** semantics.

So I think we have strong theoretical grounds for pursuing a combined grammar/semantics paradigm in the *formal* context which covers (channelized) hypegraphs on the syntactic end with **NC4** type systems and the "**NEMAM**" type-interface model. How this might look in practice is suggested perhaps by the code base accompanying this paper (and others distributed with the data set), which essentially operationalizes a Channelized Hypergraph/**NC4** combination. I also believe a strong case can be made for connecting Chanelized Hypergraph grammars in the programming language context to natural language syntax, e.g. what I have analyzed as "Cognitive Transform Grammar".

Conceptual Space semantics for natural language remains rather an odd man out for this unification, perhaps. I have presented reasons why this may be the case, concerning the extra-linguistic reaches of any comprehensive natural semantics, which would seem to inhibit *any* substative analogy between formal languages (or formal type theories, formal semantics, etc.) and human language. The question though is whether this would *practically* limit the applicability of a Hypergraph/Conceptual Space unification for linguistics (in a non-reductive vein) and humanities in general. This in turn depends on the proposed ends to which a **HCS** model might be applied in a humanities context, which I'll comment on in conclusion.

4 Conclusion

When Conceptual Space Theory migrated from a natural-language and philosophical environment to a more technical and scientific foundation — as a basis for modeling scientific data and analyzing scientific theories and theory-formation — it also picked up certain evident practical applications. For example, **CSML** was a concrete proposal for technical data modeling whose exlplicit goal was to be more conceptually expressive and scientifically rigorous than conventional — or even "Semantic Web" — data sharing tactics. So one obvious domain for concrete applying Conceptual Space Theory lies in the communicating and annotating of scientific (and other technical research) data. This use-case could certainly benefit from the added structure of Hypergraph syntactic models (which can engender hypergraph serialization formats) and hypergraph-based type theories.

So a Hypergraph/Conceptual Space hybrid can readily be imagined as a kind of next-generation extension of the Semantic Web or reincarnation of **CSML**, with an emphasis on sharing scientific data in a format conducive to capturing the theoretical context within which data is generated. This is still removed from the *natural language* origins of Conceptual Spaces, but it would mark a further step in the evolution of Gärdenfors's theory from a linguistic to a metascientific paradigm.

But going even a step further, a data-sharing framework emerging in the scientific context may retroactively be utilized in a more humanistic context as well; so an HCS hybrid may find applications in the conveying of *humanities* data — natural language structures (parse trees or graphs, lexicons, and so forth), sociological/demographic data sets, digitized artifacts (art, archaeology, museum science), etc. In this scenario Conceptual Spaces might be relevant to, say, Cognitive Linguistics on two level — a practical, software-oriented tool for linguistic research in its digital practice, alongside a paradigm for natural language semantic at the theoretical level. These two modes of application may not have fully aligned theoretical commitments, but they would reveal a core Conceptual Space theory diversify, branching, and adapting to different practical and theoretical requirements.

References