Abstract

This chapter proposes the development of a formal (or at least semi-formal) type theory by considering types as a classification system for hypergraphs. Specifically, types are constructed as properties of hypernodes and of nodes contained within hypernodes (viz., what we term "hyponodes"); the types ascribed to hyponodes determine those of their enclosing hypernodes. One consequence of this setup is a framework for defining how much flexibility we can allow to hyponodes type-ascriptions while preserving a strong type system at both hyponode and hypernode levels, yielding a formulation we refer to as "cocyclic" types. We also present an account of "non-constructive" type theory which in essences formally characterizes types by virtue of their roles in procedure-call disambiguation rather than isomorphisms between type-instances and construction-sequences. A defining trait of non-constructive types is the possibility of computing environment being unable to determine the full set of values instantiating a particular type which may be computationally represented (in a given context), or similarly being unable to efficiently determine which constructor or sequence of constructor-calls would yield a particular value. In short, this form of type-theory assumes a large functional and conceptual gap between types and their extensions (understood as value-sets). In addition, we consider procedural sets through the lens of certain higher-scale hypergraph constructions, particularly (what we call) "channels," giving rise to a formulation of type theory we dub "NC4" (non-constructive, channelized, and cocyclic). The motivation for this type theory is to consider types in a reasonably systematic manner while also recognizing how types in programming languages model real-world concepts rather than abstract mathematical entities.

Chapter 5: Types' Internal Structure and "Non-Constructive" ("NC4") Type Theory

1 Introduction

In the context of text-based serialization languages, such as 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 be registered in fixed-size, pre-allocated segments of computer memory. 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. In this chapter we will examine the connection between type systems and *hypergraph* meta-models for data representations.

1.1 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 thereby 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).[1]

Labeled DHs accordingly have two different formations for aggregating information: first, hyponodes are grouped into hypernodes; and, second, hypernodes are interrelated via labeled connections (hyperedges). This duality allows hypergraphs to combine paradigms associated with ordinary labeled graphs with data tuples or "records" (e.g., Relational Databases). So DHs evince a step toward universal data-representation frameworks for which are structurally rigorous, but not tied down to specific 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. This chapter will focus on the first alternative.

1.2 Cocyclic Types for Hypernodes

We 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).[2] 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).[3]

In the models we propose, however, we want to focus on "Procedural" Hypergraphs, which are not necessarily static structures. 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 effect, if we allow hypernodes to cover an unconstrained list of hyponodes with no type restrictions, there ceases to be a strucutural means of sorting hypernodes by type structure. The problem is then to free up type "tupling" as far as possible while preserving a strong type system at the hypernode level.

In this context, we propose the concept of "cocyclic" types 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 $\mathbb{T}$. We will say that $\mathcal{T}$ is *cyclic* if the sequence repeats: every n**th** type is the same, for some n. We 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

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

[2] This restriction — which we call "disjointness" — can actually be weakened somewhat; see [3, p. ?] for details.

[3] 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.

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

This terminology concerning how types in a patterned sequence "cycle" (as a minimal constraint on type-patterns) is idiosyncratic, as are our later terms classifying procedures defined on types (see §2.1). Our rationale for proposing new language to describe types' internal structure is that mathematical type theory tends to focus more on the interrelationships *between* types than on the *internal organization* of types (and type-instances). This internal organization is more likely to be theoretically significant when we consider how types implemented in a programming language are simulations of real-world objects; that empirical foundation, of course, is largely outside the scope of formal type theory. In practice, a practical type-theory for software engineering needs to consider types both as formal and as empirical structures, which calls for a somewhat new vocabulary to articulate types' intermediate status as partly mathematical artifacts and partly software design patterns.

In any case, our 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. 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 $\mathbb{t}$s (meaning a list of $\mathbb{t}$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 $\mathbb{t}_1$ and the values from a $\mathbb{t}_2$ — is similarly (representable as) an endocycle alternating between $\mathbb{t}_1$ and $\mathbb{t}_2$.

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 we 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. 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, we 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, accordingly, we assume that any $\mathbb{T}$ under discussion is "cocyclic," meaning that each $\mathbb{t}$ in $\mathbb{T}$ is cocyclic.

1.3 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 $\mathbb{t}_1$ (or list of input types) and output $\mathbb{t}_2$ (or again a list of output types), the identification of a type $\mathbb{t}_1 \to \mathbb{t}_2$ representing functions from $\mathbb{t}_1$ to $\mathbb{t}_2$. 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 function 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**).

We say that a type system is *Channelized* if function-like types in $\mathbb{T}$ can be designated by describing the "channel complex," or groups of channels aggregated 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). We 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 (potentially empty) sequences of (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 procedure's "**return**" channel will hold no value, which on this theory is a valid carrier-state.

Recent work in mathematical linguistics has developed a form of "Hypergraph Category Theory" that can also be applied to programming language type systems (see Coecke *et al.* [4], an article we will return to several times in later chapters). Coecke *et al.*'s motivation for adopting hypergraphs is similar to 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.

Also, nodes may represent side-effects which can be effectuated by a software system.[4] 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.[5]

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 such as Coecke *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 (including zero). That is, Hypergraphs in this Category-Theoretic context 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 corresponding construction, in practical software, via techniques generally described as "reactive," as in "Functional Reactive Programming". A concrete example is the so-called *signal/slot* mechanism used in C++ within the Qt libraries (see [5]). 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.

There are various options for 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) model presented here (previously outlined in [3]) is different in some details than Hypergraph Categories — including by introducing channels as an extra construction on graphs — but arguably 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.

1.4 Constructors and Carrier States

Coecke *et al.*'s Hypergraph Category Theory reflects the how 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 sequence 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 modifying 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 de-

[4] 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.

[5] 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 not passed on to another procedure but instead effectuates the change in some real-world state.

rived by adding some number n to a shorter-by-one list v'. Algorithms which depend on examining 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 numbers 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 (arguably, difficult to maintain and static-analyze).

However, this recursive style of programming is only possible if specific metadata is embedded with $\mathtt{t}$ values which allows "deconstructing" $\mathtt{t}$ instances into construction *patterns* and allows $\mathtt{t}$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, such as "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 $\mathtt{t}$ 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 as 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 describing 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.

The discussion in [3] introduces 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.[6]

Co-constructors are similar to what some programmers call "factory methods" or "object factories." 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). As we will argue next chapter, reflection via function-pointers can be a foundation for code-annotation and "documentation by implementation" scenarios. 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:

[6] 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.

- 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 $\mathfrak{t}$ values from literal strings.
- Binary constructors which initialize $\mathfrak{t}$ 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 $\mathfrak{t}$ values from an aggregate data structure which may (but need not) include other $\mathfrak{t}$ 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.
- Recursive constructors allow values obtained by pattern-based constructions to be "deconstructed" and used for recursive algorithms, as outlined earlier in the case of lists.

We 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, we 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 "sentinel" values which 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, some programming languages 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 reasonable 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 year old) is probably not a reasonable default for someone's age. When data is missing, should default values be used; if not, how should the problem be represented? These decisions influ-

ence 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, respectively — and that in turn is different from an invalid value which should generally not be passed between procedures (which we'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 **null** 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.

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' "extensional space"; how there are (sometimes) patterns and groupings amongst type-values more specific than just being instances of their respective types. Arguable such extensional groupings are to some degree analogous to Conceptual Spaces, marking the "space" of types' extensions in a conceptual fashion. Of course, extensions also have *statistical* distributions which are a more precise fit for Conceptual Space theory, but both of these models (we can call space-partitions based on criteria such as null or default-initialized values *pre-statistical*) are potential forums for integrating type theory with Conceptual Spaces.

1.5 Nonconstructive Type Theory

Thus far we have suggested 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 must implement procedures which bifurcate based on which pattern matches; and they have to take a multi-part structure as a procedural input, like v' *and* e, in the place of a single carrier like v. One straightforward way to achieve this is by differentiating 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.[7]

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, we 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 we'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 we 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 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 complica-

[7] 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.

tions in conjunction with an overall type interface, which might not be immediately apparent. For instance, consider types which have (what we earlier called) "binary constructors" — e.g., a t which can initialize values by copying the bit-pattern of some other t-instance. For many $\mathbb{T}$ this option results in the theoretical possibility that ts 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 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 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 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 t, in short, may need to ensure that no 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 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 t-instance v there is then a determinate sequence of constructions which eventually produces v, and likewise an "inverse" sequence 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, t values are understood to be internally inseparable from the progression of steps which provide a recipe for constructing those ts.

Mathematically, an analogous assumption is that the space of t-values is isomorphic to the space of construction-sequences which yield ts. We can also say that the space of these sequences is a *model* for 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, we propose considering 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. We 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 that can model either non-constructive *or* types or both. Non-constructive type systems, we contend, achieve the greatest breadth in covering the diversity of possible $\mathbb{T}$s while staying within the bounds of software-centric, implementational rigor, especially in conjunction with the features of being *channelized*, and *co-cyclic*, from which we derive the "NC4" moniker (**n**on-**c**onstructive, **c**hannelized, **c**o-**c**yclic). In the following chapters we will implicitly adopt NC4 type systems as a foundation for studying the semantics of formal languages, in the hopes of deriving a theory of synthesizing Hypergraphs with Conceptual Spaces — which can be usefully paired with analogous unifications for *natural* language.

2 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 cuisine (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 alcohol. An overall "restaurant" type (say, $\mathcal{r}$) would then be a "product" of these various fields. In Conceptual Space Theory, likewise, concepts are defined by a crossing of multiple dimensions, which collectively define the space of variations which their instances can occupy. 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 between *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) continuous space which permits distances but not ordering, unless we are taking distance from one central point (e.g., someone looking for restaurants 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 $\mathcal{r}$ 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 $\mathcal{r}$ 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 $\mathcal{r}$ type reveal how all conceptual details about restaurants can be reduced to numeric patterns. Instead, quantitative structures fall out as the result of *operations* on this type — operations like ordering, querying, or measuring similarity. We 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 we 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 favorable reviews, assigning each restaurant a "grade" from, say, 0 to 100. Our $\mathcal{r}$ 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 statistically 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 we 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 acquaintances — 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. We will now examine these kinds of operations in more detail.

2.1 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. We 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 we 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, we think it is possible to make some broad claims about how data fields *usually* work in the intertype-comparison context.

On that basis, then, we 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 scenes 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: We use this term to represent any fields which have nominal, statistical, scalar, or in any sense quantitative qualities that can be leveraged for comparisons. We 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, we 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 ts axiatropic fields, which we propose to call ts *axiatropic structure*, defines a multi-dimensional space wherein ts are mapped to individual points. The tuple of values obtained from these fields we call an *axiatrope*, and the points to which a given t-axiatrope projects we 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, to be discussed in later chapters).

4. Flags: We 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, we 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 we 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. We 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 we would then add the portion of the interface related to constructing t-instances: constructors, co-constructors, and related functionality for testing the validity of a data structure and/or "deconstructing" values into a construction pattern. Collectively we will refer to this last aspect — which does not involve fields *per se* — ts *constructive interface* or (with apologies if the neologisms are getting a little heavy-handed) its *nomotropic interface*. The fields (textual, flags, binary) which are neither axiatropic nor collections-based we will call *endotropic*. We 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.

The *axiatropic* aspect of type-extensions may seem like a jargony designation of rather mundane statistical distributions of a set of values, since projecting data structures onto common axis-sets which permit quantitative comparisons is the bread-and-butter of data analytics. Such projects themselves certainly aren't a new theoretical posit. However, we wish to emphasize that in most types (especially ones modeling real-world objects) the process of setting up such statistical comparisons involves *filtering* fields so as to focus on statistically malleable information in each type-instance. It should be emphasized that the application of statistical metrics to value-sets constitutes a *projection* of each value onto a restricted axiated space which supports quantitative dimensions, whereas many types will *also* have qualitative data which is more difficult to accommodate numerically.

The fields which are likely relevant to comparing or querying ts are facets of t that need to be deliberately engineered. Implementers, in short, have to anticipate and provide procedures for client code — software 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 integrating 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.). We 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 we 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.

We think this perspective is also consistent with natural-language: a conceptual account of *nouns*, for instance, 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 are constituted essentially by their intellectual-functional roles. We will explore "conceptual roles" further in Chapter 9.

3 Hypergraph Ontologies

Hypergraphs — along with structures that can be modeled via hypergraphs, such as Conceptual Spaces — are an improvement on Semantic Web data formats and schema, addressing the limitations of a paradigm devoted to modeling complex information via first-order logic and non-nested graphs, with no notion of scoping or locality.[8] At the same time, a lot of effort has been extended building technologies to integrate heterogeneous data spaces via the Resource Description Framework (RDF) and RDF ontologies. In radiology, for example, attempts to better incorporate clinical and outcomes data are centered on ontologies such as RadLex, ViSion, and SeDI, which we discussed in Chapter 2. As a result, the important consideration is how to employ hypergraphs as an extension to the Semantic Web when warranted while preserving the virtues (and interoperating with) conventional RDF ontologies.

[8] These are familiar critiques, but laid out with particular thoroughness by the Conceptual Space community, as we will discuss in Chapter 7.

The idea that hypergraphs *extend* but do not *replace* RDF and the Semantic Web implies that hypergraph schema are extensions of (but not substitutes for) RDF ontologies — which in turn yields the notion of *hypergraph ontologies*. In a conventional RDF ontology, metadata is primarily associated with graph nodes and edges. In particular, nodes are referenced to Uniform Reference Identifiers (URIs), such as web addresses, and edges are labeled with concepts formally defined in one or more ontologies. Concepts which are used to annotate graph-edges, and which are given a fixed meaning in some controlled vocabulary, are often called "properties." One special "is-a" property is often used to connect nodes with concepts that classify entities into one of many categories defined in an ontology, often called "classes." As such, most RDF ontologies are primarily composed of *classes* and *properties*, each assigned a unique label. The purpose of metadata for a given graph is then to link nodes to classes (for example, specifying that one node represents a clinical trial and the second represents a patient), and furthermore to link edges with properties (for example, specifying that a patient-node is connected to a trial-node in that the patient is *enrolled in* the trial).

Hypergraph ontologies are similar to conventional RDF ontologies in that they likewise provide constraints and metadata for graphs. However, hypergraph ontologies are more complex because hypergraphs are likewise more complex than ordinary graphs. In particular, hypergraphs have different layers of structure: whereas RDF nodes are intended to represent a single concept or value (such as a number, date, personal name, or URL), a *hypernode*, within a graph, typically encompasses multiple pieces of information inside it (often called *hyponodes*, *projections*, *inner elements*, *roles*, or just *nodes*).[9] In general, when analyzing hypergraphs it is necessary to distinguish at least two "tiers" of nodes, *hypernodes* and *hyponodes*, such that hyponodes are contained within hypernodes. As a result, hypergraph ontologies need a corresponding distinction for node and edge annotations: insofar as nodes are categorized via classes, and edges via properties, it is necessary to stipulate whether these classifications apply to hypernodes, hyponodes, or some combination of the two.

A further complication arises because, even though hypergraphs represent nested or hierarchical structures, these hierarchies are often partial or overlapping. For example, a patient is *part of* a clinical trial, but a patient is also included in other collections as well; for instance, a patient may be enrolled in a specific health plan (for insurance coverage). One technique for modeling overlapping hierarchical data via hypergraphs is to employ "proxies," which are digital identifiers encoding a multi-faceted concept into a single value that can be part of a hypernode (proxies are similar to "foreign keys" in SQL). Therefore, each patient, represented by its own hypernode, has an identifier which can be a proxy-value for the patient;

[9] The term "roles" is used by Grakn.ai (see https://dev.grakn.ai/docs/schema/concepts); "projections" is used by HyperGraphDB (see http://www.hypergraphdb.org/?project=hypergraphdbpage=RefCustomTypes); "inner entity" is used by the biointelligence project [7], where the corresponding notion of "external entity" refers to what in other contexts might be called other hypernodes linked to a given hypernode via hyperedges.

for example, a value assigned to a hyponode becomes included in the hypernode encoding the list of patients enrolled in a clinical trial, or in the hypernode encoding the list of patients enrolled in a specific health plan. Hypernodes can then be linked to other hypernodes by virtue of proxies (e.g., the trial-to-patient connection), and also by virtue of overlap (e.g., the set of all patients both enrolled in a given clinical trial *and* enrolled in a given health plan).

In sum, compared to RDF — where there is one single sort of node-to-node relationship, based on whether or not an edge exists between nodes and how this edge is labeled — hypergraphs are more flexible/expressive because they have multiple genres of node-to-node relationships: the relation between hypernodes and their inner hyponodes; between hypernodes and one another; between hyponodes in different hypernodes; and variations on each of these relation-types wherein relations are defined indirectly through proxies. Moreover, in addition to hypernodes and hyponodes, hypergraphs afford additional levels of detail, such as *frames*, *channels*, and *axiations*. All of these details provide different "sites" where hypergraph annotations and metadata may be defined.[10]

An additional distinction within the Semantic Web is the contrast between *reference ontologies* and *application ontologies*. In general, *reference ontologies* are general-purpose schema intended to establish conventions shared by many different applications, to ensure that a large collection of data-producing software in a given domain is interoperable. By contrast, *application ontologies* are narrower in scope because they are more tightly integrated into applications that directly send and receive data. Ontologies of either variety are used by software to interoperate with other software: so long as two applications are using the same ontologies, it is possible to ensure that one application can understand the data produced by a second, and vice-versa. However, such interoperability is only potential; it is the responsibility of programmers to actually implement code which produces and/or consumes data that conforms to the relevant ontology specifications. In general, application ontologies are structured in such a way that these concrete implementations are more straightforward to produce, compared with reference ontologies. Reference ontologies offer little guidance to developers vis-à-vis how to directly support the ontology within application code. Conversely, application ontologies more rigorously describe the data structures which applications must implement in order to properly manipulate data that is structured according to the specifications of the ontology.

Within data mining and image analysis, hypergraph models are used in different ways for different algorithms. In the context of Covid-19 radiology, [6] (which we cited last chapter) describes an algorithm for assessing the probability of SARS-CoV-2 infection from chest CT scans, where hypernodes represent high-dimensional vectors (191 dimensions overall) and hyperedges represent k-nearest-neighbors; here each hypernode represents an entire image, mapped to a 191-dimensional feature-vector. In contrast, other image-analysis methods use hypernodes to designate smaller segments *within* the image, where hyperedges designate geometric adjacency and/or feature-space similarities. Whatever the algorithm, hypergraph analyses would employ a hypergraph library to store preliminary data for analysis and/or for serialization within a data set. One benefit of a Hypergraph Application Ontology is therefore that these data structures used internally to implement analytic methods can be directly expressed within the ontology, whereas RDF ontologies can only model hypergraphs indirectly.

Although it is theoretically possible to encode data directly via RDF graphs, it is far more common for applications to employ tabular and/or hierarchical formats for data sharing, such as spreadsheets, Protocol Buffers, XML, or JSON. As a result, the role of ontologies for constraining data structures (so that they adhere to common standards) is indirect. It is useful to remember that ontologies are, at their most basic level, Controlled Vocabularies; as such, ontology constraints often amount to stipulating a set of acceptable terms for a data value, column header, or annotation. For a trivial example, our calendar recognizes 12 month names and 7 day names, which constrain the set of values permissible for "month" and "day" within a calendar date. These terms are so commonplace that a "date ontology" is unnecessary, but in scientific or technical domains it becomes necessary to define vocabularies of allowable names or labels for specific data fields that representing some scientific value or measurement. For instance, the Ontology of Vaccine Adverse Events [8] provides a nomenclature for use in Adverse Events Reporting, so that researchers or clinicians can describe symptoms via canonically recognized terms rather than through informal text descriptions. In general, ontologies constrain data sets by stipulating that particular individual values within the overall data collection have names or descriptions whose associated set of possible values is prescribed *a priori* by the applicable ontology. However, the relationship between ontologies and concrete data sets must itself be documented, which is where application ontologies can become relevant — application ontologies provide a bridge between reference ontologies and the applications which use them (along with the data generated and shared by those applications).

In order to preserve the benefits of RDF ontologies — while also addressing those lacunae which make the Semantic Web "not (really) semantic" — hypergraph ontologies need to model constraint schema on hypergraph constructions (which have significantly more parameters of structuration than RDF graphs) while also connecting these schemas to the Controlled Vocabularies and logical axioms of Semantic Web (particularly OWL) ontologies. There are as such several areas of detail within hypergraphs where links to RDF ontologies may be drawn, which are outlined here:[11]

[10] This means that formats for describing hypergraph ontologies have to be more expressive than RDF ontologies, because RDF ontologies need only to classify metadata as node-annotations or edge-annotations; by contrast, hypergraph ontologies need to distribute annotations among multiple sites of graph structure.

[11] A full explanation of these concepts and terminology depends on an in-depth treatment of hypergraph type theory, which is outside the scope of this pro-

Hypernodes' Cocyclic Type Structure One of the central principles of hypergraph data modeling is the use of *hypernode types* to specify what sort of information is necessarily associated with a hypernode. In particular, a hypernode encompasses multiple hyponodes, each with their own type. These hyponodes represent information in some sense "contained within" or "tightly connected to" a hypernode (whereas data less canonically associated with each hypernode would, in general, be asserted via hyperedges rather than via hypernode/hyponode containment). In order to ensure that hypergraphs are predictably organized, hypernodes cannot have arbitrary collections of hyponodes, but must instead be aggregates of hyponodes which are assembled according to a schematic pattern, defined in terms of hyponode types. For maximum generality, a hypergraph type system should allow hyponode-type patterns to be as flexible as possible without introducing a need for metadata asserted at the level of individual hypernodes rather than hypernode types; this motivates the idea we proposed above of a "cocyclic" type system which is minimally constrained (but not unconstrained).[12] When translating RDF ontologies to hypergraph schema, accordingly, one consideration is whether edge-requirements are sufficiently ubiquitous in some context (e.g. with respect to some **rdf:class**) that these edges should be translated to hypernode/hyponode inclusions, and then to define a pattern of hyponode types for the corresponding hypernode type (the alternative is to retain the edges as links between *hyper*-nodes if they are *sometimes* but not *always* joined.

Roles, Projections, and Dimensional Annotations A hypernode type provides a schema defining a sequence of hyponode types; it is sometimes said that the hypernode "projects onto" that space of hyponode types. This projection is minimally characterized by hyponode types, but some hypergraph systems allow the projection to be *annotated*, introducing additional metadata that constrain (or augment the expressive power of) the enclosing hypergraph. Annotations can define scales/units/levels of measurement, probability distributions, situational roles, and other details lending semantic grounding to the data-field encapsulated by a hyponode. This metadata can then be a vehicle for translating RDF class constraints to hypergraph schema.

Semantic Nominal Dimensions The most direct translation of Controlled Vocabularies to a hypergraph context is often that of constraining the space of variation for one specific project to a set of allowable terms. In the typical case, a hyponode type encapsulates a nominal set of values (i.e., an enumeration), so any hypernode including that type as one of its projects is constrained by the labels registered in the vocabulary (a related formulation replaces non-hierarchical vocabularies with *taxonomies*, where some labels are treated as more or less granular variants of others).

posal.

[12] Becase a pattern of hyponode types is "cocyclic" if the type-sequence includes a (possibly empty) tuple of types with no requisite pattern (called the "precycle") followed by a repeatable type-tuple, cocyclically typed hypernodes can represent expandable data structures such as lists, stacks, queues, deques, and dictionaries. A typical hypernode type may then indirectly include multiple collections-types via proxies.

Dimension Aggregates, Domains, and Conceptual Spaces Conceptual Spaces can be modeled in the hypergraph context by noting that hyponode projections are sometimes interdependent: dimensions tend to aggregate into semantically related groups (like *latitude* and *longitude* as geographic markers). In Conceptual Space models, accordingly, projections are split into two levels — *dimensions* and *domains* — while other dimensional-analytic constructions (such as scales and units of measurement) are carried over (see [1]). Conceptual Space Theory also then introduces concepts of fuzzy logic or "convexivity" (according to different metrics) to simulate patterns in human conceptualization. We will discuss Conceptual Spaces in greater detail over the next few chapters.

Probabilistic, Temporal, and Overlapping Hypergraphs Other forms of metadata constrained via ontologies can be expressed in terms of annotations defining weights or probabilities on hypernodes and/or hyperedges. One example is the juxtaposition of alternative markup hierarchies, in the context of hypergraph representations for Concurrent Markup languages such as TAGML [2]. Numeric edge-annotation can represent weights (e.g., provide measures of the degree of uncertainty in the edge's relation actually obtaining), but constructions similar to weights have other sorts of applications. For instance, edge-annotations can be measures of time-spans, allowing hypergraphs to describe "entity-event models."[13]

Proxies, Inverted Proxies, and Double-Inverted-Proxy Constructions As described earlier, hypernodes can assert "containment" of other hypernodes by containing a *hyponode* which *proxies* the second hypernode. An *inverted proxy* connection is therefore the mirror-image of this assertion (which may or may not be formally recognized by the hypergraph). A *double-inverted-proxy* connection is accordingly the relation obtaining between two hypernodes which are both proxied by one third hypernode (using the earlier example of proxies, the fact that two patients are enrolled in the same clinical trial). Many graph connections identified in a Semantic Web context (e.g., by SPARQL queries) are likely to translated to double-inverted proxies in a hypergraph context.

In general, these hypergraph constructions represent sites for asserting constraints that (for RDF ontologies) would be defined on classes or properties; they are therefore a natural scaffolding for translating RDF ontologies to hypergraphs. Such a translation mechanism allows existing ontologies — which may play a valuable role in specifying protocols for workflows and data-sharing between software components — to be reused in a hypergraph modeling environment.

As illustrated by CAPTK (discussed in Chapter 2), multi-application workflows are characterized both by the data which is transferred between applications and by the operations which connect the two applications — that is, the procedures enacted by each application when they become operationally linked. As a preliminary model, we can identify two stages of

[13] See https://allegrograph.com/consulting/entity-event-knowledge-graphs/.

operational connection between an already-running application (which may be called the *primary* component) and a second application launched by the primary (which may be called the *peer* component). The first stage occurs when the primary component launches the peer component, and is characterized by two operational sequences: procedures enacted by the primary prior to this launch, and procedures enacted by the peer subsequent to the launch. A second stage occurs when the peer component has completed its actions, and sends data back to the primary component, which again involves two operational sequences: procedures enacted by the peer prior to the transfer, and procedures enacted by the primary subsequent to the transfer. Fully describing the procedural workflow therefore entails specifying four operational sequences: primary, then peer, during the launch stage; and peer, then primary, during the transfer stage. A schema describing the operations performed during these four sequences can be called a *procedural ontology*.

Consequently, rigorous models of multi-application networks should *synthesize* information about data structures (the type of information shared between application-points) with information about procedural workflows (describing operational sequences prior to the launch and transfer stages of a multi-application linkage). The synthesis of this structural and procedural information can be called a *procedural application ontology*.

One area where "procedural" Ontologies would differ from conventional Ontologies on the Semantic Web is that Ontologies in the former sense seek to characterize procedures (which by nature are digital artifacts) as well as objects (whether concrete or abstract). Of course, the Semantic Web notion of "object" is highly general, and certainly encompasses a broad range of abstract entities, including anything existing in the context of computer software and source code. So procedures as objects — or the analogous equivalence in the programming context, "functions as values" — can certainly be modeled with some degree of precision in a Semantic Web context. The difference between Semantic Web and "Hypergraph" Ontologies is not absolute, but it involves shades or degrees of emphasis. In the context of Hypergraph Ontologies, we can say that an *intrinsic* detail of computational data structures is that many data-types are collections of values which instantiate other data types — that is, containers which contain multiple instances of other types, and these containers can vary in size as values are added or removed. Different types of containers add or remove values in different ways: for instance, many containers are ordered, and restricted so that new values can only be placed at the end (or, alternatively, at the beginning) of the list. For these sorts of data structures, modeling procedures governing how the structures may change state (e.g., how lists may acquire new values) is an intrinsic dimension of modeling the semantics of the data structures themselves. This interconnectedness between data structures and procedural requirements is not entirely outside the representational scope of the Semantic Web, but nor is it the primary conceptual focus of Semantic Web Ontologies. However, the interconnectedness of data structures and the procedures which operate on them *is* an intrinsic concern of programming language type theory. A thorough foundation for reasoning about data-sharing therefore calls for some form of hybrid analysis encompassing both Semantic Web Ontologies and type theory, as we will briefly review in the next subsection.

3.1 Type Theoretic Foundations for Hypergraph-Based Data Sharing

Data communications is a trade-off between optimization for space and bandwidth (often it is important to limit the number of bytes in each unit of information to a minimum), security, and programmability (networks that aim to support many developers spread across multiple projects and institutions need to be more open-ended than those centrally controlled by a single team). Ideally, then, each data type and data structure will be amenable to serialization in multiple formats, which each elevate one priority or another; XML and JSON are more flexible in an open-ended project, for example, whereas binary serialization is more efficient. Software models should be rigorous: there should be documentation of reasonable values for data fields, and (where applicable) their units of measurement; preferably there should be some automated testing and/or type-checking to ensure that these expectations are sustained. Such specificity allows programmers to maximize resources that in some contexts may be restricted, such as internet bandwidth. On the other hand, requirements engineering should not degrade production performance, so verification should be executed in a special run mode that can be disabled for deployment, when necessary. For speed optimization, data types often fall back to types like 4-byte integers, which express can support many more values than actually are conceptually meaningful. In these situations, the proper solution is not to degrade runtime performance with smaller and/or more complex types but to model those types as predeployment checks of some sort, such as debug assertions testing the success of a type cast.

Similarly, conceptual modeling needs to anticipate the diverse ways that people express concepts. For instance, someone's age is often measured in years, but with respect to toddlers many people cite an age in weeks or months; if age is extracted from textual input it is necessary to anticipate phrases like "six weeks" or "six months"; potentially this should be reflected in the types through which age is modeled. If someone enters a child's age as six months, it might be disorienting to see this subsequently listed as, say, "zero years." This is an example of how provisional assumptions about the proper type representation of a human concept can be too simplistic, and therefore degrade User Experience. It also shows the benefit of a rich type system, where types can be crafted to better model human concepts: if ages like "six months" are recognized, then functions for comparing and binary encoding ages need to be implemented accordingly. A key role of a type system is logically organizing functions necessary for the type to work properly, which becomes more necessary as types acquire greater complexity (in the form of special flags and values) to

adapt to human concepts and/or to interfacing with speech and natural language.

The above examples show why semantic modeling can be important to software development: semantic models are guidelines that improve applications' robustness and correctness by simultaneously codifying developers' and stakeholders' expectations and anticipating concepts which end users will bring to bear on their interactions with deployed software. Unfortunately, there are several different notions of semantic models, with roots in distinct academic fields, that do not precisely overlap, making it hard to integrate multiple semantic models into coherent, multi-paradigms information systems. Among semantic modeling paradigms, we can perhaps identify three which are particularly influential: grammars and lexical specifications for Natural Language Processing; formal Ontologies described in formats such as OWL (Web Ontology Language) and providing semantics for data formats, query systems, and persistence mechanisms such as RDF, SPARQL, and graph databases or "triple-stores"; and the formal semantics at the foundation of type systems and implementations of concrete programming languages (implementations of interpreters, compilers, and/or runtimes).

In many modern-day applications, all three of these paradigms will likely play a role. For example, in a medical setting, bioinformatic semantics and communications are often modeled by some variation of OWL and RDF; patients' natural language (e.g., in the form of clinical narratives) is an essential part thorough health records; and application-specific data types need to encapsulate aggregates of medical information as communicated between applications via protocols such as DICOM (Digital Imaging and Communications in Medicine) and HL7 (Health Level Seven International). It is also important to observe that document-level criteria (like restrictions on XML documents) do not provide enough behavioral specification to address operational concerns, especially in domains requiring complex calculations and native code libraries (medical imaging, robotics, 3D graphics, image analysis, video analysis, or simulations of physical systems). Robust technology in these environments demands that a given real-world concept is modeled by behaviorally equivalent types at different points on a network, and that a serialization format (like XML) serves merely as a conduit for typed values between network points. The components which send and receive typed values should be designed and tested for conformance to behavioral expectations once they have a fully-formed instance of the relevant type in running memory; this is a more rigorous test than can be applied by checking an XML document against a Document Type Definition, for example. It seems fair to say the internet technology often fails this standard, with a frequent use of XML or JSON as a raw data stream that is processed as an untyped data structure on the receiving end, rather than packaged into a type or class whose overall behavioral properties are well-documented.

Expressing operational constraints through languages' type systems and implementational mechanisms can be difficult. Consider the example of using a non-constant reference parameter passed to a function so as to initialize a value. This design requires that the function body always assign a proper value for that parameter and never attempt to read its value prior to assignment. This rather straightforward requirement is hard to express in many languages; in a pure-functional context the use of non-constant reference parameters is not directly allowed in the first place (forcing a more complex syntax at the call site), whereas in an Object-Oriented or procedural language there may be no distinct type to capture a guarantee of initialization (analogous to how a assertion guarantees that a value is "not" changed). Type systems are often designed around optimizations which may be useful in many contexts but which are not always realistic, and the attempt to guide programmers toward writing optimizable code can unnecessarily obscure code which needs to be structured differently — compare the syntax for monads and strict evaluation in Haskell against the (simpler and more transparent) syntax for lazy evaluation, which is paradigmatically preferred. Conversely, expressing lazy evaluation in an Object-Oriented language like C++ is convoluted and requires library support. These trends illustrate how type systems emerge from (sometimes too abstract) mathematical theories and need to be supplemented by semantic models that recognize conceptual as well as mathematical formalisms.

The consequence of this discussion in the present context concerns the problem of meta-modeling: the preferred frameworks wherein specific concrete data models may be developed. Various data structures — such as hierarchical trees or Semantic Web style graphs — have been proposed as universal meta-modeles, with the idea that such representations are sufficiently abstract and flexible that they can accommodate any data-modeling requirements in a computational/digital environment. This section, however, has attempted to identify reasons why these general-purpose data structures are suboptimal for data modeling at the level of Requirements Engineering and detailed procedural analysis. In this book we claim that Hypergraphs, augmented by notions taken from Conceptual Space Theory, can provide a better meta-modeling foundation.

3.2 Hypergraphs as a Meta-Model for Data Sharing

Publishers, in recent years, have begun to encourage authors to share their research data, and to incorporate "Data Availability" or "Supplemental Materials" sections as an integral part of their journal and book publications. Initiatives such FAIRsharing or the Bill and Melinda Gates Foundation Guidelines for Authors (which is part of FAIRsharing) provide recommendations to help authors produce data sets that are Findable, Accessible, Interoperable, and Reusable (FAIR). Such initiatives represent a step toward realizing the goal of engineering a broad ecosystem of open-access scientific data — an ecosystem that interconnects and interoperates with both scientific publications and scientific-computing software applications. However, this goal is hard to attain because, in reality, scientific

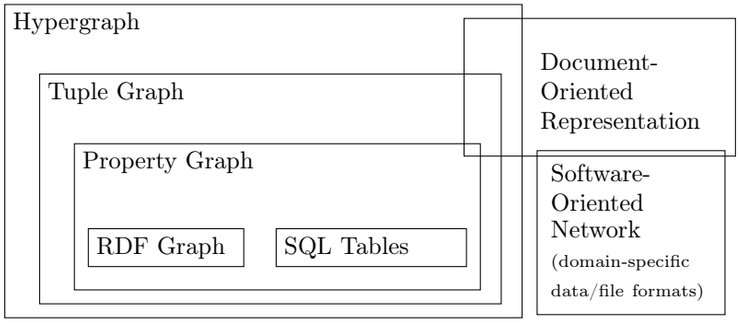

Figure 1: Relationships Between Different Kinds of Data Models

applications, publishing software, and data-hosting technology remain largely siloed from one another.

The FAIRsharing standards also include several dozen domain-specific guidelines collectively referred to as MIBBI (Minimum Information for Biological and Biomedical Investigations). The MIBBI specifications serve as a checklist for authors preparing data sets to ensure that they properly, and in sufficient detail, document their research and/or laboratory protocols (there are also research-documentation formats associated with specific publishing platforms, such as *Springer Protocols*, part of Springer Nature). As these examples demonstrate, researchers have numerous options for how they may document their research data. However, none of these documentation format options are closely integrated either with publishing software — in particular, with software used to compose books/articles for publication — nor with scientific software applications used to acquire, analyze, and visualize research data.

Data-sharing in the publishing context is often based on the Semantic Web. For example, "Research Objects" — a standard model for composing reusable data sets — relies on the canonical Semantic Web data format, RDF (Resource Description Framework), to encode research meta-data. Semantic Web data is often criticized, however, for being "flat" — that is, RDF does not directly recognize any information structures which involve multiple levels or organization, or contexts: data collections (such as unordered sets or ordered lists), clusters of interrelated pieces of information, contexts where certain connections between information units have elevated significance, and so forth. To redress these limitations, multi-tier models such as Property Graphs and Hypergraphs have been proposed as alternative vehicles for Semantic Web data. While Property Graph and Hypergraph Database Engines are increasingly popular in business IT, these technologies have not been comparably adopted within the publishing industry.

The interrelationships between several different data-modeling paradigms — and hybrid models that seek to unify multiple paradigms — are sketched out in Figure 1 above. "Document-oriented" architecture in this context refers to formats such as XML, which allow arbitrarily nested content; these formats can model information with multiple levels of organization, but they are often difficult to work with in text-mining and data-mining contexts. Property graphs augment the flat-graph model by allowing sets of properties to be defined on edges and vertices. Property graphs, in turn, may be generalized to "tuple" models (where nodes could be collections of other nodes) and then subsequently to hypergraphs proper, which can be seen as tuple-graphs augmented with extra details characterizing the information encompassed by individual nodes. As one proceeds from more restricted to more flexible data models, we achieve capabilities to represent larger classes of data structures, so that general-purpose data-sharing initiatives should in principle embrace broader frameworks (such as property or tuple graphs or hypergraphs proper) in lieu of narrower ones (such as SQL or RDF). However, it is a challenging problem to implement programming environments for the more complex graph models — in particular, encoding, querying, and validating complex graphs — which may be one reason why such technology has not been embraced by publishers, despite there being promising code libraries available that could serve as a foundation for technologies targeted to authors and publishers.

Overall, in conclusion, hypergraphs represent a logical evolution of different meta-modeling paradigms, and they subsume many other data structures as special types or cases. Accordingly, we propose intuitively that hypergraphs incorporate the desired structural features of numerous other paradigms, while also introducing added structural details that can make hypergraphs a preferred foundation for database engineering and data-sharing. Of course, this intuition derives from a rather cursory overview of competing data-modeling paradigms. We will develop a more substantial theory to back up these ideas in subsequent chapters.

References

[1] Benjamin Adams and Martin Raubal, "A Metric Conceptual Space Algebra". *International Conference on Spatial Information Theory*, 2009, Proceedings, pages 51-68. https://pdfs.semanticscholar.org/521a/cbab9658df27acd9f40bba2b9445f75d681c.pdf

[2] Elli Bleeker, *et al.*, "Between Flexibility and Universality: Combining TAGML and XML to Enhance the Modeling of Cultural Heritage Text". http://ceur-ws.org/Vol-2723/short39.pdf

[3] Nathaniel Christen, "Hypergraph Type Theory for Specifications-Conformant Code and Generalized Lambda Calculus". In Amy Neustein (Ed.), *Advances in Ubiquitous Computing: Cyber-Physical Systems, Smart Cities, and Ecological Monitoring* (Elsevier 2019).

[4] Bob Coecke, *et al.*, "Interacting Conceptual Spaces I: Grammatical Composition of Concepts". Extended version of *Proceedings of the 2016 Workshop on Semantic Spaces at the Intersection of NLP, Physics and Cognitive Science*, pages 11-19. https://arxiv.org/pdf/1703.08314.pdf

[5] Ivan Čukić, "Functional and Imperative Reactive Programming Based on a Generalization of the Continuation Monad in the C++ Programming Language". Dissertation, University of Belgrade, 2018. http://www.math.rs/files/ivan-cukic-phd.pdf

[6] Donglin Di, *et al.*, "Hypergraph Learning for Identification of COVID-19 with CT Imaging". *Medical Image Analysis*, Volume 68 (2021). https://pubmed.ncbi.nlm.nih.gov/33285483

[7] Toni Farley, *et al.*, "The BioIntelligence Framework: A new computational platform for biomedical knowledge computing". *Journal of the American Medical Informatics Association (JAMIA)*, Volume 20, Number 1 (2013), pages 128-133. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3555311

[8] Erica Marcos, *et al.*, "The Ontology of Vaccine Adverse Events (OVAE) and its Usage in Representing and Analyzing Adverse Events Associated with US-Licensed Human Vaccines". *Journal of Biomedical Semantics*, Volume 4 (2013). https://jbiomedsem.biomedcentral.com/articles/10.1186/2041-1480-4-40