Abstract

This chapter will introduce a particular graph model which we call "Syntagmatic" Graphs, applicable to both natural-language parse-structures and to modeling computer code. We explore some properties of queries over these graphs and briefly consider features of a Virtual Machine whose capabilities would include operations forming the foundation of syntagmatic-graph query-evaluation. We argue that syntagmatic-graph queries are structurally related to other software-engineering constructions, such as Aspect-Oriented Programming, and discuss implementation strategies for static code analysis and runtime-reflection, in the sense of exposing procedures to scripting and Remote-Procedure interfaces. We then introduce a schematic figure we call the "Semiotic Saltire," which informally classifies programming concerns into four engineering "aspects" supplemental to core type-implementations, namely GUI design, serialization, (database) persistence, and "procedural models," the latter of which includes details such as runtime reflection and remote-procedure call capabilities, as well as annotative procedural pre- and post-conditions. We conclude by introducing certain benefits of hypergraph-based representations for data persistence.

LH		RH
1		31
2		32
3		33
4		34
5		35
6		36
7		37
8		38
9		39
10		40
11		41
12		42
13		43
14		44
15		45
16		46
17		47
18		48
19		49
20		50
21		51
22		52
23		53
24		54
25		55
26		56
27		57
28		58
29		59
30		60

Chapter 6: Using Code Models to Instantiate Data Models

1 Introduction

In their synthesis of hypergraph categories and Conceptual Space theory that we reviewed in earlier chapters, Coecke *et al.* [9] adopted an approach for *syntax* (based on hypergraph categories) which is familiar in a computer-science setting, while favoring for *semantics* a paradigm (Conceptual Spaces) which emerged from a linguistic context. Category theory is not without precedent in linguistics, and likewise some projects have attempted to formalize Conceptual Spaces for computational applications, such as data modeling [1], [2], [45], [17], [56] and AI [16], [5], [20], [21], [31], [32], [53]. Nevertheless, Coecke *et al.*'s hypergraph-and-conceptual-space approach, rooted in Quantum NLP — Natural Language Processing carried out on quantum computers (or simulations thereof engineered for research purposes) [36], [37], [15], [33], [41], [47] — represents a hybrid paradigm integrating perspectives from both formal analyses of computations and programming languages and from natural language.

Correlations between formal and natural languages are of particular interest to NLP — because NLP by definition needs to use computers, equipped only with formal algorithmic capabilities, to understand (often ambiguous or context-sensitive) natural language. Conversely, formal programming language theory (e.g., Software Language Engineering) may draw ideas from natural language on the premise that (although first and foremost grounded on the affordances and limitations of digital artifacts such as parsers and compilers) programming languages are designed by people, where (at least for high-level languages, as compared to machine and assembly code) human readability and understandability is an important aspect of well-written code. This chapter will focus on programming languages, not natural language, but the use of hypergraph categories to structure the framing of NLP problems in terms suitable for Quantum processors (with quantum gates, qubits, and so forth) will serve as a motivating example for structures we present as a generalization on hypergraph categories.

In the case of Coecke *et al.*, the authors emphasize that syntax and semantics should mirror one another so that dynamic processes in both areas can be linked together (specifically, in Category-theoretic terms). In their words: "[T]he general programme is [to] interpret the compositional structure of the grammar category in the semantics category via a functor preserving the type reduction structure .... This functor maps type reductions in the grammar category onto algorithms for composing meanings in the semantics category" (p. 2). Adapted to Conceptual Spaces, their "programme" can be more rigorously laid out: "[W]e construct a new categorical setting for interpreting meanings which respects the important convex structure emphasized in conceptual spaces theory. We show that this category has the necessary abstract structure required by categorical compositional models. We then construct convex spaces for interpreting the types for nouns, adjective and verbs. Finally, this allows us to use the reductions of the pregroup grammar to compose meanings in conceptual spaces" (p. 2-3). A core specification here is that *type reductions in the in the grammar category* map onto *algorithms for composing meanings.*

It is obvious that "meaning is compositional" (a paradigm so entrenched, or perhaps so self-evident, that Jerry Fodor could teasingly insinuate its *a priori* resolving all questions about whether thought precedes language or vice-versa [13]). The notion of compositionality is more substantive in the context of "syntax-semantic interface," or specifically questions about how syntax and semantics respectively contribute to linguistic composition [30], [10], [25]. Coecke *et al.* make the further implicit claim that the syntax-semantics interface can be formally modeled via Category Theory: semantic composition is driven by compositions on the syntactic level with a force and causal precision that can be expressed mathematically, as if semantic constructions are transpositions or translations of syntactic constructions, akin to mapping a set onto its image under a mathematical operator.[1] We can then ask what *causes* syntactic constructions to engender semantic ones. As will be analyzed more thoroughly in Chapter 9, we take the perspective here that semantic constructions have *more information content* than their component parts, and that syntactic constructions thereby derive causative force from their encoding rules for how semantic elements can be unified with a specific emphasis on *increasing* information content.[2]

We can, in short, picture semantics as a kind of *path* which leads in a direction of more information, more detail, and more specificity, wherein the terminus of that path is a sentence's overall complete idea (e.g., the proposition which a sentence conveys, if it is expressed in illocutionary terms — as an assertion rather than a question, request, expression of desire, and so forth).[3] Bob Coecke implies a similar information-theoretic model when he suggests that syntactic relations "change" words (viz., the cognitions they tokenize; sentences "update their meanings"): "A sentence is not a state, but a process, that represents how words in it are updated by that sentence" [8, p. 18]. Coecke *et al.*'s Category-Theoretic ma-

[1] To be sure, almost every NLP method assumes that syntactic and semantic construction-patterns can be tightly integrated, because this is precisely what can make semantic tractable to computer algorithms. The *issues* at stake in this assumption — first *whether* quasi-mathematical treatments of semantic constructions being the map-image of syntactic constructions is sufficiently true for natural language, without oversimplifying linguistic nuance, and second *how* syntax and semantics are integrated in this manner — can be tightly expressed by adopting terms and perspectives from Construction Grammar [14], [38], [55]. Specifically, we can say that semantic constructions (or *paradigmatic* constructions, alluding to the paradigmatic/syntagmatic opposition) are *determined by* syntactic (likewise, or *syntagmatic*) constructions if our cognitive construal of a given paradigmatic construction, a given construction on the paradigmatic "pole" of language, is *caused by* our apply certain *rules* to synthesize syntactic elements into a syntactic construction.

[2] In effect, we can incorporate certain notions from *situational semantics* (*see*, e.g., [11], [39] or [43]) paired with *construction grammar*, with the connections between the syntax and semantics modeled through what we will call (departing from both situation and construction theories) "paths."

[3] Taking seriously Juan Uriagereka's metaphor in [54] that "syntax carves a path" which semantics "follows," though without committing to such following being automatic or possible without interpretive effort (which can be visualized by saying that a "syntagmatic path" can actually engender multiple "paradigmatic" paths, and that nuanced interpretation may be needed to disambiguate them).

chinery serves (without using these specific terms) to specify the nature of information-augmentation paths via morphisms at the syntactic level (providing "steps" toward a sentence's total information content) which engender information-bearing constructions at the semantic level. The phenomenon of semantic constructions being reflected images of syntactic ones can be accounted for, on this perspective, by treating semantic content as some body of information that is built in stages, thereby implying a path *leading* along a gradient of information-augmentation to the construction in its totality; such paths are causally grounded in the corresponding syntactic construction, and, on this theory, these paths thereby form the causative nexus of the syntax-semantics interface.

For reasons to be clarified in Chapter 9, we will use the term *syntagmatic constructions* to refer to models of syntactic construction which are explicitly based on this notion of information-content amplification. Whereas syntactic rules are static conventions that are apparent in the context of individual phrases, we will argue that "syntagmatic" models address the underlying dynamics governing the evolution, entrenchment, and cognitive internalization of syntax; notions of information content thereby serve as a hypothesis concerning the principles underlying syntagmatic dynamics. Moreover, we claim that these interlinked notions of information content, syntactic and semantic constructions, and "path" models of the interface between them, provide a foundation for grammars that have applications in programming languages and data modeling as well as (natural) linguistics. In particular, this theory points to connections between natural-language parse-graphs and formal representations of source code (apart from the obvious similarity between two models which both diagram structures derived from applying parsers to an input language): specifically, one can examine graphs in both contexts in terms of *directions* of information increase. We seek to codify the informal picture that graphs have paths along which a measure information content (in some sense) elevates from site to site.

There are additional details which may be added to this graph-theoretic picture, but the central point is that graphs meeting certain criteria (which we will briefly define here, deferring to later chapters a linguistic justification for this model) codify intuitions about cognitive and conceptual processes guiding the dynamics (the "syntagmatic" tendencies) which underlie concrete syntax. For sake of discussion, we will refer to such graphs as "Syntagmatic" graphs, a terminology which is only distantly related to the usual sentence of "syntagma," but which captures the notion that these specific graph-structures are stipulated according to desiderata arising from the syntagmatic dynamics operating within language (or so claimed by our underlying theory). In this sense, the discussion below will describe *Syntagmatic Graphs* as a specific class (or maybe Category) of graphs which are equipped with certain structural rules and features. Syntagmatic Graphs differ in some respect from the Categories examined in Coecke *et al.*, but arguably these graphs capture many of the details and intuitions informing those author's use of hypergraph categories, and in general Syntagmatic Graphs can therefore provide one system for representing structures in natural language. However, we will also argue here that Syntagmatic Graphs may be useful tools for representing *formal* (e.g., programming) languages.

Coecke *et al.*'s reasons for adopting hypergraph categories to model natural-language grammar were not, first and foremost, philosophical; instead, they were looking for representations of language which are conducive to analysis or recapitulation via operations that can be programmed on a Quantum computer. As such, the value of hypergraphs in this context lies above all in how hypergraphs organize data so that operations on data can be "compiled to," or implemented in terms of, a computational machine instantiating quantum-computing logic.

Our proposals here are analogous in that we recommend a specific variety of hypergraphs (Syntagmatic Graphs) whose properties are selected to facilitate implementation of a virtual machine to evaluate queries over data sets and data models. As (summarily) formulated here, this virtual machine would be classical (not Quantum).[4] Nevertheless, the overarching methodology here applies for both Quantum and classical settings: graphs models can be articulated with particular concern for *the nature of virtual machines which evaluate queries against the graphs* and moreover problems in multiple domains (e.g., data integration and NLP), which are not apparently graph-theoretic on the surface, can be translated to *queries against graph structures*. In combination, these principles point to criteria for optimally formulating graph metamodels: pay attention to how real-world problem-domains give rise to graph-queries and then to the design of virtual machines evaluating those queries, which are constrained by the structures of graphs encoding the queried information. This chapter will examine these dynamics in the context of (what we call) Syntagmatic Graphs.

2 Syntagmatic Graphs and Pointcut Expression Semantics

The last few paragraphs have outlined, at least in brief sketch, what amounts to a "theory of language" in which the crucial dynamic of linguistic understanding — manifest in both syntax and semantics — is a certain expansion in information content driven by an effective co-ordination between semantic and syntactic constructions. On one level, this may seem obvious (it is hardly newsworthy to claim that sentences convey information) but we intend to emphasize the idea that information is accumulated in *stages*, and that these stages thereby can be modeled via *paths* situated in both semantic and syntactic "spaces" or structures (of some kind still to be elaborated). We can then connect linguistic models (such as Coecke *et al.*'s mixture of hypergraph categories and Con-

[4] Although via parallelization tactics similar to the Gremlin virtual machine (for property-graph query evaluation) it could conceivably be optimized by factoring certain algorithms into Quantum form (perhaps using Coecke's existing quantum-computing hypergraph models as a starting point).

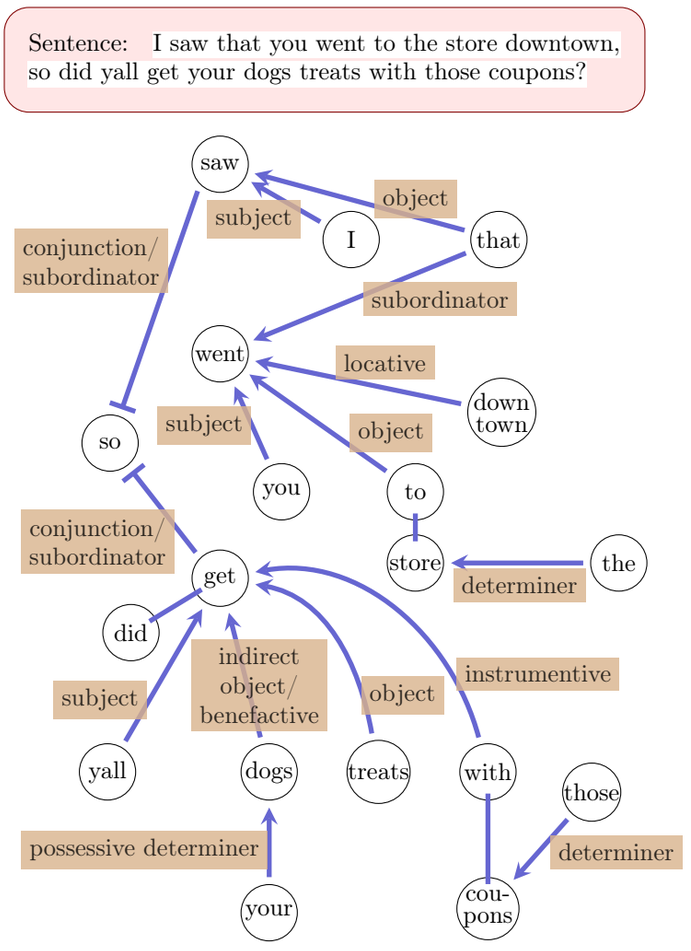
ceptual Spaces) to computational and graph-theoretic contexts where notions of *path* and *information content* are well-defined.

A full discussion of this theory in linguistic terms is outside the scope of this chapter, but what *is* relevant here is the structural format which emerges when we try to express this linguistic picture in formal models, e.g., parse-graphs. The meaning of individual linguistic units, in particular, are the details they provide — the information content they contribute — to the accretion of detail vis-à-vis an overarching semantic construction. As we will argue in Chapter 9, grammars fluidly modeling this information-theoretic perspective are generally "verb centric," insofar as semantic constructions are canonically designations of states, events, or processes which are concretized in the root-verb of a relevant clause. Parse-graphs, for example, would embody word-to-word relations where each word adds detail to a verb-profiled scenario, and where "paths" tracing word-to-word connections lead toward verb-nodes, following "contours" of greater information content. In graph-theoretic terms, this implies directed, labeled graphs (the labels being lexemes/morphemes and parts of speech on nodes, and inter-word relation kinds, as in Link Grammar [50], [12], [51], on edges) with the property that every directed edge belongs to a path leading to a verb-node and verb-nodes are, in general, *targets* (rather than *source-nodes*) of edges. Edges can be classified in terms of the *kind* of information they contribute to their construction (see Figure 1 for a visual diagram of this model).

For sake of discussion, we will stipulate that verb-nodes are always *only* target-nodes. Of course, verbs form clauses which can be incorporated into other verb-profilings (as their own details) — this is why sentences can have more than one verb. In many cases nested clauses are "packaged" via hinge-words such as "that," as in *I saw that you went to the store*: here *went* has its own cluster of details (its subject, say, is *you*), while *saw* has a different cluster (among other things, *it's* subject is *I*). The pivot which bridges these two different foci — we go from in some sense picturing or learning about the occasion of someone going to a store to learning about the fact that this event was *witnessed* directly or indirectly by the speaker — is the conjunction *that*, which we can say points in two different directions: on the one hand it connects to the inner clause which it "packages" and, on the other, to the outer clause which draws in the inner clause as a component part (usually playing a noun role; e.g. a going-to-the-store as object of *saw*). As this suggests, the paths leading *between* verbs can be defined in terms of hinge-nodes which are incident, as source nodes, to two different edges targeting two different verb-nodes. This form of structure may be introduced as a defining principle of Syntagmatic Graphs to preserve the stipulation that verbs are always target nodes rather than source nodes.

Such a convention differs from Coecke *et al.*, who approach verbs from the perspective of Category-Theoretic morphisms; in graph terms, their perspective implies that verbs have *inputs* and *outputs*, analogous to mathematical functions, which

Figure 1: Semantic Constructions as (Verb-Centered) Information-Amplification Pathways

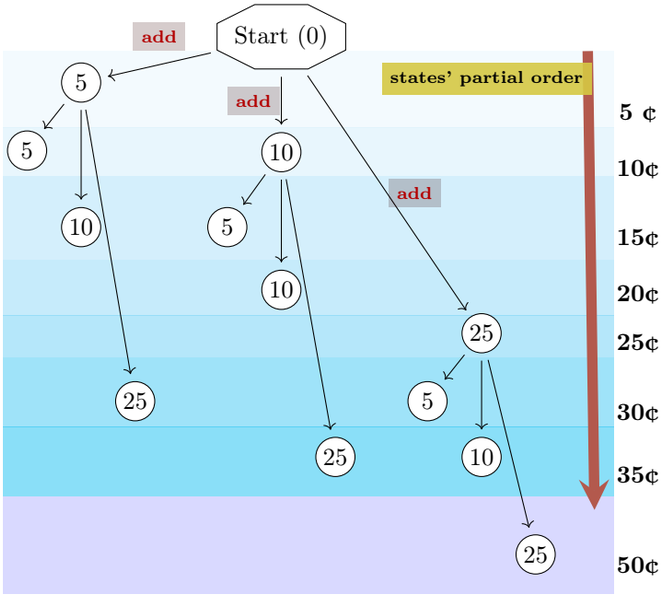


carries over to verb-nodes having *input edges* and *output edges*. However, in practice, programming language runtimes generally hold the results of procedure-calls in "temporary" values, if not lexically or globally scoped variables. As such, unfolding the execution sequence when the result of one procedure is passed to another, there is indeed an intermediary value of some sort — which may be subject to further actions, such as a type-cast or, say, in C++, call to a copy-constructor — and which for graph-representations would be roughly analogous to a linguistic conjunction-node such as *that*.

We therefore propose a modification of the conventional scheme for encoding syntactic structures in parse-graphs (whether for natural or formal languages); in the model used here directed edges always point *toward* nodes representing procedures/verbs (the distinction between "inputs" and "outputs" thereby being established by annotations on edges or their source nodes, rather than by edge-direction).

Such an inversion may seem to be a minor notational variation, but it arguably points to a larger shift in perspective, which can be considered in terms of both natural and programming languages. In the natural-language context, we have sketched a theory whose central feature construes semantics in terms of amplification in information content; in Chapter 9 we will more specifically define information content

Figure 2: Diagram of a "Vending Machine" State Chart Showing States' Partial Order (based on total coin value)



in terms of "accretion of detail" vis-à-vis verbs in particular. This means that for each verb we can identify a collection of "details" associated with the verb, which are all "fed" into some cognitive process that conceives an event/state/process which the verb profiles. All meaning-content in a sentence is therefore "pulled in" to one or another cognitive verb-profiling. At the same time, the full mass of relevant detail is brought to bear on the cognitive process; we are able to cognize the verb-profiling which the speaker intends to communicate because we mentally assemble the details relevant to the verb as a precondition for considering the verb (or its associated meanings) as a cognitive phenomenon.

Via the analogy of natural-language verbs to programming-language procedures, we can similarly see a running program as undergoing an *accretion of computational content* which results in there being sufficient data available to execute a given procedure. Procedure-calls in formal computer science are often considered in terms of "reductions," as in lambda calculus: a procedure call "reduces" to the value which the procedure returns, in the sense that this value takes the logical place of the procedure in subsequent code (sometimes an optimizing compiler will literally replace a procedure's return value for the procedure call itself if that value can be determined at compile time). However, an equally salient picture is that before the lambda "reduction" there is an *expansion* in the sense that the total data for a procedure call accumulates by determining (if necessary, evaluating nested expressions) the values for all the procedure's input parameters.

This analogy fits better with process calculi in lieu of lambda calculi, in that (in the language of process-calculi and Petri Nets) a procedure can only be called when all "bindings" or "markings" are in place (*see*, e.g., [34] or [35]). As such, a "Syntagmatic" computational model can be defined in terms of computations instantiating evolutionary processes whereby data is accreted (according to specifications registered by parameter-bindings) and must reach a threshold of completeness in order for a procedure to be called (or "fired," still borrowing Petri Net terminology — values are "fed into" a procedure much like coins into a vending machine, one of the examples often used to illustrate Petri Nets; procedure outputs are then akin to your candy bars released by the machine when it has enough coins). Once a satisfactory level of information content is reached — by resolving all symbols and nested expressions forming procedural inputs — calling the actual procedure is the next computational step. Indeed, in some distributed-object systems (such as Vission [52]) such an "accretion of content" provides the explicit form of the calling-interface: the origin code constructs values and binds them to "slots" in a remote environment, then (when all needed bindings are defined) explicitly signals that the procedure is ready for execution.

As much as vending-machine-type analogies are popular small examples of Petri Nets (or, say, Finite State Machines), an interesting detail that rarely seems to get noted in this case is how adding coins to the machine does not merely *change the state* of the system. It also *augments the amount* of money in the machine. In other words, we have a natural ordering of most or all possible states based on the total value of all coins deposited at the point when a given state obtains. This ordering affects state-transition rules, because the rationale for transitioning to an *end* state — not just awaiting more coins, but rather releasing a candy bar and resetting to a ground (no coins) state — is that the total value of coins dropped overtakes the target value of the purchased product (we represent this scenario pictorially in Figure 2).

Our analogy between vending machines and procedure-calls derives from the idea that evaluating nested expressions — or reading the value of symbols which are input as single tokens — can be modeled as adding "information content" akin to a total sum of money rising with each vending-machine coin. The idea of information content — and of ordering systems' states in terms of variant *amounts* of such content — introduces new semantic possibilities to the potential utlization of graph or hypergraph-based representations for different data structures. We think this intuitive notion has possible applications to natural language as well as formal (programming) languages. Indeed, we will sketch out a theory which finds certain overlaps between the syntax and the semantics applicable for several different domains: (1) representations of natural language; (2) representations of programming languages and source code; (3) representations of assertions and beliefs within data and "knowledge" bases; and (4) serializations of data structures.[5] We will point out how introducing *increase in information content* as a core structural feature within these diverse domains helps solidify our understanding of their overlap, which could potentially be underdetermined

[5] For this most part this book uses natural-language formations as motivating examples (on the premise that formal-language semantics can emulate linguistics proper all else being equal) and we are not focusing on NLP itself, although our discussions in the natural-language context could point to a query-architecture for text-mining and textual database engineering, potentially.

without a theoretical background: while there may be superficial connections between (say) natural and programming languages, or between source code and serialized data structures, we need a theory which can leverage such similarities for them to have any practical value.

In itself, a generic appeal to "information content" does not carry very much theoretical weight either, but it *can* serve as structural ground for more well-developed syntactic or semantic accounts. For example, information content could be leveraged in the context of natural language's distinction between "theta" and "thematic" roles (to be discussed further in Chapter 9); theta-roles derive from verb/subject or verb/object pairings, whereas thematic roles derive from syntactically more distant details (consider *I drove some students to a conference in Philadelphia*: the *conference* and *Philadelphia* are *thematic* in their grammatic relation to the verb, and likewise are more peripheral semantically than the "thetic" verb-subject *I* or object *some students*). This distinction at the level of clause-formation primes us to place different levels of emphasis on the information contributed by theta-role content versus thematic-role phrases.

In the context of programming languages, we can identify different *varieties* of information content. Some of these are (potentially) determined as a static invariant within source code, such as the types of every parameter passed in to a procedure, or such as a procedure's memory address (which is thereby associated with any source-code token naming that procedure). Some of these details may *or may not* be fully resolved at compile-time. For example, a virtual method would typically be referenced via a source-code token that maps to multiple procedures at runtime (based on the actual type of its **this** object) and argument types (assuming we distinguish supertypes from subtypes) would likewise be opaque for static analysis. Meanwhile, specifics such as the actual values of procedure arguments would (outside of unusual cases) be open-ended until the runtime procedure is actually called.

We can compare these cases through the lens of information being added on at different times — some details are fixed by the compiler, some deferred to runtime, and some occupy an intermediate status where both "early binding" or "late binding" is possible. Given such a classification for information-content vis-à-vis procedures, we can then consider how details *other* than type-attributions and value-bindings fit in to the compile-time/runtime contrast, such as argument ranges and typestates in conjunction with procedural preconditions. The net result would then be a model for computer code which theoretically buttresses the abstract idea of *information content* with a classification system structured around notions of compile-time/runtime and early/late binding.

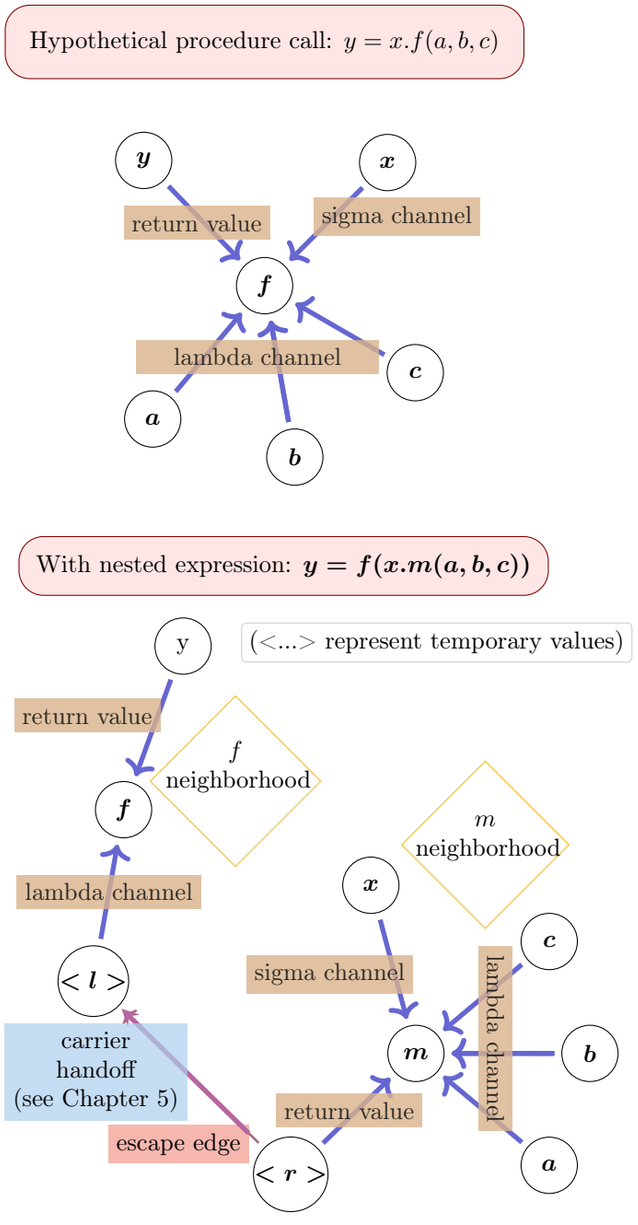
In the same way that natural-language semantics can be glossed with the picture that all meaning provides details which supply information content for a verb, we can adopt the perspective that computational processes involve declared or evaluated values supplying data for a procedure call: any time there is a concrete value which is part of a program's running state, there is a procedure toward which that value will be targeted as one requisite parameter. To fix the "meaning" of a value, then, we need to identify the procedure where that value is used. This is one rationale for adopting a convention wherein all nodes not themselves representing procedures (instead encoding literals or scoped symbols) point toward procedure-nodes. In such a graph, each procedure-node is surrounded by nodes representing data that must be concretized (such as input values that will be bound to the procedure's input parameters) as a precondition for the procedure to be runnable. Although when looking at source code we may see this as a static phenomenon (a single expression being a unit of a computation, analogous to one clause being a unit of linguistic meaning) during execution this process unfolds over multiple stages, wherein each step represents one piece of preconditional data being set in place anticipating a procedure call.

Such a picture justifies the model that *all* non-procedure nodes, even those representing output values, become source-nodes of edges directed *toward* procedures. For example, a procedure can only be called if memory is allocated to hold the procedure's return value — although the actual return value is not provided until the procedure itself runs its course (being as such a *postcondition* of the procedure) it is a *precondition* that the return value will be "held" somewhere, so the edge connecting the node representing this "carrier" (using terms from Chapter 5) can point *toward* the procedure node because the carrier (albeit not an actual value) is a precondition for the procedure running. Output and input nodes are therefore *both* procedure-preconditions; the difference is that input-nodes are preconditional in needing an actual *value* whereas output nodes need only represent the capacity to *hold* a value.[6]

In the case of natural language parse-graphs, "bridge" nodes between verbs represent parts of speech such as subordinators/conjunctions (e.g., *that*). The analogous construction in programming languages would be output nodes for one procedure becoming input nodes for a "parent" procedure (vis-à-vis expression nesting). However, the model of Syntagmatic Graphs presented here proposes a modification of this scheme in which such "pivot" nodes are essentially split

[6] As an example, the **[[nodiscard]]** attribute in C++ enforces a requirement that the result of a procedure be accounted for — bound to a variable or passed to another procedure — which means that the context for the call to proceed is not correct without additional code that uses that return value. This situation is roughly analogous to passing an argument by reference which is intended to hold the result of a computation (a common pattern in languages that can only return one sole value from functions — if multiple returns our needed, the others are handled instead by overwritable references which the called procedure initializes so as to communicate the value that would otherwise be returned). A non-optional parameter must, of course, be passed in to a procedure in order for the call to be possible. But passed-in parameters sometimes serve the *semantic* role of being initialized (or re-initialized) as if they were return values; in short, the syntactic difference between a return value (as bound to, say, *y* in **y = f(x)**) and an overwritten reference parameter is a surface-level distinction; semantically the two constructions are similar. Insofar as each (non-optional) parameter is a *precondition* for the procedure-call, then there is no reason not to consider a *return* value (or more precisely some mechanism in the calling context to use that value) as equally a precondition (which is appropriately modeled by connecting a node representing this value *to* the procedure node, rather than having the edge point in the opposite direction). After all, both a (non-optional) parameter and a (non-discarded) use of a a return value are preconditions for a procedure-call (in this analogy, the *absence* of a non-discard attribute would be equivalent to allowing an *input* parameter to be optional rather than required).

Figure 3: Neighborhoods in Syntagmatic Graphs



in two. In general we want each node in a parse-graph to be *uniquely* adjacent to one procedure-node (nodes are uniquely adjacent if they are adjacent to only one edge whose other node has the relevant unique property — in this case that of being a procedure node). Instead of a single pivot node, therefore, with two different procedure target-nodes, we prefer a pivot "edge" connecting two distinct nodes, one representing the output of a procedure and the other the input of a different (parent-expression) procedure, with the *edge* between those non-procedure nodes representing the phenomenon of one's return value being the other's input parameter (the edge thereby representing source-code sites and program-runtime stages where details such as type-cases and temporary-object constructors/destructors come into play). We present this as a further stipulation on Syntagmatic Graphs intended to model computer code.

8

This overview does not fully exhaust all considerations needed to circumscribe the relevant class or Category of Syntagmatic Graphs. One point we have not yet mentioned is that the actual procedures called in a program might itself be evaluated at runtime, so that what appears as a procedure node in source code may in fact be a procedure which *yields* a procedure which *then* gets called — in other words, a single procedure node in this kind of case embodies *two* (or more) procedure-calls, where the first yields a value which holds the second (as a function-pointer, say), the actual procedure called to complete the expression not being known until runtime (until, that is, the first procedure returns, and then in turn other layers' if applicable). This scenario can be accommodated by allowing inputs/outputs to a procedure node to be sorted into layers, one layer representing an expression which when called yields a procedural *value*, that in turn constitutes the procedure whose parameters are bound through the second layer, and (potentially) so on iteratively. The analogous construction in natural language would be adverbs modifying verbs to form verb-phrases; it is these modified "verbs" which form the nexus of "details" informing the relevant verb-profiling.

With these further constructions at least sketched provisionally, we can clarify the specific properties of Syntagmatic Graphs that for this analysis we employ to model programming language code. Syntagmatic Graphs for these purposes are characterized by properties such as:

Procedure nodes Nodes in general are labeled with type-attributions, and by extension nodes whose types represent procedural values (those with input and/or output parameters and/our side effects, as opposed to static values) can be distinguished from other nodes;

Procedure nodes as target nodes Procedure nodes are *target nodes* but never *source nodes* for edges labeled with annotations representing computational constructions, i.e., the label semantics requisite to render directed labeled graphs as models of computational processes, embodying lambda or process calculi (this does not preclude procedure-nodes being potential source nodes from a fully distinct class of edge-labels, such as those representing sequencing between distinct source-code statements);

Procedure nodes as path targets All paths (when constructed via edges whose label semantics represent the subset of labels specific to modeling computational processes, as in the previous item) terminate in procedure nodes, and each procedure node is adjacent to a collection of non-procedure nodes, each of which are adjacent to no other procedure node — for sake of discussion, although this is technically a non-standard way of using the term, the set of a procedure node and its "surrounding" input/output nodes can be called a "neighborhood," so that a program's source-code graph can be decomposed into distinct neighborhood, each representing a distinct expression;[7]

Carrier-transfers as inter-neighborhood pivots With source-code graphs divided into neighborhoods in this

[7] To be precise, we could define a *neighborhood* in a *directed* graph (this is a term more common in undirected graph theory) as the in-neighborhood of a node with outdegree zero.

manner, a specific group of edge-kinds can be defined to represent bridges between expressions, wherein the node representing one procedure's return value is connected (in the sense of passing a value to) one representing another procedure's input (expressing "carrier-transfer" semantics as described in Chapter 5);

Channels and layers Edges incident to a common target-node can be grouped into "channels" as defined in Chapter 5, representing collections of input or output nodes with similar semantics (e.g., regular inputs are a channel distinct from object message-receivers, which have a different semantics, and regular outputs are distinct from thrown exceptions), and "layers" to accommodate where procedures return values which are themselves other procedures — that are then called with bindings derived from a different collection of nodes in the same "neighborhood."[8]

Syntagmatic Graphs meeting these criteria are *hypergraphs* because edges incident to a given procedure-node are grouped together; instead of one input and one output node we have edge-collections structured via channels and layers. Coecke *et al.* generalize graphs to hypergraph categories so as to model similar arity-variance among inputs and/or outputs (procedures may input and/or output zero, one, or more than one value), though (as outlined in the prior paragraphs) Syntagmatic Graphs have additional structural details than defined for those authors' hypergraph categories. We contend that Syntagmatic Graphs as defined here serve similar roles to their hypergraphs but, being somewhat more expressive, present more detailed modeling features.

2.1 Query-Evaluation Foundations for Syntagmatic Graphs

Our rationale in setting out schemata such as these definitions/constraints for Syntagmatic Graphs is not primarily presentational — as we have intimated, there are numerous non-isomorphic or not-fully-compatible but nonetheless similar structures which can model computational phenomena such as procedure calls; structural differences may iconify philosophical variations in how we choose to "think about" computation (or, say, formal data models), but it should not be taken for granted that subtly alternative philosophical framings are consequential from the perspective of implementing useful technologies. For our purposes, the importance of structures such as Syntagmatic Graphs lies in the project of codifying the operational and implementational environments contextualizing technologies that work on data structures such as graphs as digital resources. This involves graph query languages and the engines which evaluate such queries, as well as software for representing and serializing graphs, which in turn should be designed with consideration to how query-evaluation engines would work; the in-memory and on-disk representations of graphs provide the structures on which graph query engines operate.

Structural features defined for a particular kind of graph correspond to graph-elements which may be identified within queries, incorporated into underlying representations, and accounted for when evaluating queries against such representations. Structural features can be motivated, as such, based on whether they make queries more expressive and/or whether they support representational devices that are helpful for query-evaluation. In this sense distinct structural formats (e.g., several different graph Categories) may have noticeable differences in terms of the forms of queries they engender and the ease and flexibility with which those queries are evaluated and adapted to problem-domains based on how a graph technology is being used, in context.

For example, consider the edge marked "carrier transfer" in Figure 3 (between nodes <**r**> and <**l**>). This edge does not represent an example of the kind of program execution step (i.e., a "join-point") that is apparent on the procedural (source-code) level, but it could be defined as analogous to a join-point on a "subprocedural scale" (we'll discuss these terms below). Moreover, this edge describes a consequential point in program execution, because it is possible that operations will be performed behind-the-scenes here (e.g., type casts and class constructors/destructors). It would be beneficial, accordingly, to implement a pointcut-expression system wherein designating that specific runtime moment is possible, motivating the use of multiple nodes with "escape" edges and handoffs to represent return values being passed to outer procedures.

As a terminological aside: we employ the phrase "hypergraph queries" to encompass several kinds of expressions that could be evaluated in the context of hypergraph structures and databases (following conventional usage in the context of formal languages for manipulating databases). More specific kinds of hypergraph queries would include *traversals* (logic for moving between different graph-locations); *descriptions* (stipulating conditions on hypergraphs for a specific purpose); *site-locating*, or more concisely *siting*, referring to finding graph-locations meeting some criterion; *serialization* (encoding a hypergraph so it may be shared between different applications or computing environments), and *extraction* (pulling information out a hypergraph, by matching the structure around some *site* with a *shape* specifying how this structure encodes data that is expressed in that part of the graph).[9]

[8] To expand upon the notion of "neighborhood" introduced in Footnote 7, we'll call a neighborhood as defined there a "0-neighborhood," meaning the in-neighborhood of an outdegree-0 node (called the *center* node), and a "1-neighborhood" to be the in-neighborhood of an outdegree-1 node. A "1-neighborhood" therefore has one edge which can be called the "escape" edge, referring to the one edge incident to the neighborhood center node as *source* (or *(direct) predecessor*, for the terms *predecessor*/*successor* being common expressions in lieu of what we call *source* and *target*). By analogy, we can say that an *escape edge* is an edge which is source-incident to a node in a 0-neighborhood which is *not* the center node; a 0-neighborhood would have a *unique* escape-edge if there is only one escape edge (e.g., when modeling a procedure call in an environment where procedures may not return multiple values). An *escape channel* would be a channel formed by aggregating all escape-edges in a 0-neighborhood. A *layered neighborhood* can then be a neighborhood where non-center nodes are indexed by a number denoting a layer; we can stipulate that an escape node/channel on a neighborhood may only include nodes from the highest-index layer.

[9] In graph databases in general, a *shape constraint* designates requirements on the structures evident between graph elements — e.g., which edges exist between nodes taking into consideration edge and node labels — which then indirectly encodes how the graph structure is used to express data for that specific graph-encoding scenario. The canonical example of a shape-constraint would be using a collection of nodes to encode named data fields of a central data value: one node represents that value, and it is connected via labeled edges whose labels reproduce (or at least are uniquely associated with) data-field names, where the incident nodes of each edge are labeled with

Our general goal in this book is to reduce data-integration problems to hypergraph query problems. In other words, assuming we have a hypergraph-representation strategy which can support the various additional constructions we have been or will be setting forth (channels, layers, neighborhoods, and so on) such a paradigm lays the foundation for a collection of primitive operations on hypergraphs (thus defined) and the possibility of decomposing hypergraph-query algorithms to a set of such operations. Our claim that many data-integration problems can be reduced to hypergraph queries does not entail that a query language designed for data-integration solutions should explicitly refer to "low-level" hypergraph terms or concepts; instead, the high-level query language should employ vocabulary and grammatical formations which reflect the high-level domain relevant to each part of the language (e.g., pointcuts and pre-persistence representations, using terms to be discussed later in this chapter).

That being said, insofar as query evaluations on these various domains are "reducible" to hypergraph queries, then it should be possible to (within the requisite query-processing engine) translate or "compile" the high-level concepts to hypergraph concepts that can be addressed by the hypergraph operations just mentioned. These principles encapsulate our query-evaluation strategy; although this chapter presents only a summary insofar as we do not identify examples of these kernel operations or translations explicitly, this overview sketches out what would constitute a complete theory of the relevant hypergraph algorithms and the compiler-architecture to translate high-level queries to those algorithms.

2.2 Use-Cases for Source-Code Graphs

Our "Syntagmatic" Graph model has more structural detail than other systems for representing computer code — such as Abstract Syntax Trees or Semantic Web style RDF graphs — and one claim of this chapter is that this distinct form of graphs bears consequential differences to other code-representation strategies, rather than being just a superficially different notational convention. We need to explain, however, why the representations proposed here go beyond mere notational variations (manifest mostly at a metadiscursive level); in other words, why these variations in how source-code is *described* translate to variations in algorithms or logics for how source code is analyzed or otherwise manipulated.

To address this question, note that there are several different reasons why formal representations of computer code are important. One reason is that such representations are a precondition for translating high-level source code to machine or virtual-machine instructions, but even outside of compiler theory there are several use-cases for formal code representations. One is the fact that source-code files represent digital assets in their own right which are interactive/visual objects in the context of software programs such as Interactive Development Environments (IDEs), where users expect features such as syntax highlighting and code completion — displaying code with different colors for tokens bearing different roles (variables distinct from procedure-names and from literals, for example), and with interactive capabilities such as providing information about procedures via context-menus bound to source-code tokens bearing their names. Displaying source-code files as interactive documents in this manner is only possible when IDEs can build structural models of source code, rather than treating the files as unstructured text files.

Another use case is run-time reflection: examining source-code allows programs to make certain capabilities (such as executing a procedure by supplying its name dynamically) available at runtime (more on this kind of use-case below); or aspect-oriented behavior modification (where code is added to adjust how the program behaves at certain specific execution-points). Moreover, an additional scenario — which overlaps with aspect-orientation in this sense — is that source files need to be processed as "queryable" assets, where we can search for one (or multiple) code-locations meeting some specific criterion (an example would be applying aspect-oriented modifications at the locations thereby identified, but such code-searching equally well applies to using IDEs, or to scripts composed for code-analysis, where for instance programmers conduct code-review by locating points in code which reflect specific design patterns, or which tend to be sources of bugs and coding errors).

In short, a number of different kinds of software components — including compilers, IDEs, and code-libraries which implement various runtime-reflection and/or aspect-oriented runtime modification systems — need to construct and manipulate formal representations of computer code. Therefore, strategies for code-representation are important insofar as precise and expressive representations make such components easier to design and implement. Formal source-code models come into play at several stages of working with code-representations, including building such representations in the first place — although code-models can be instantiated by parsing source code directly, in many contexts one can construct more refined code-representations by extending a given code base with annotations either inline with or external to the code itself. Therefore, one use-case for code models is designing languages for *annotating* the code which is actually seen by a compiler.

A second use-case is designing languages for *describing* code-locations subject to further examination or processing — in aspect-oriented parlance, such location-descriptions are called "pointcut expressions." So an essential dimension of code-annotation languages is the capability to formulate pointcut expressions which can be used to select points of interest (often called *join points*) within a body of source

the corresponding field values (a **last-name** label annotates the edge pointing to a node whose label is a person's last name, for example). A shape constraint this stipulates that a given node must have at least those edges whose labels correspond to fields which must be present for the value represented by the central node to be correctly instantiated. Conversely, then, extracting data from such graph-sites — e.g., getting a person's last name — leverages the appropriate shape-constraint to determine which graph site (e.g., which node) holds or can be manipulated to obtain the desired information (shape-constraints are analyzed in e.g. [22], [42], [18]).

code.[10] A further use-case for code representation is to actually support such designations; that is, to traverse source code in order to specifically identify the pointcuts described by a given pointcut expression. Code representations can be more or less efficacious in terms of how well they support such query-guided traversals to evaluate pointcut expressions.

Note that how best to articulate the semantics of pointcut expressions is an open question — there are several options for making explicit the general idea that join-points represent "locations" in source code. That could literally mean points in source code taken as a textual document, but it can also mean some step in the runtime execution of a program — which in the latter case engenders a need to define "execution steps." Ultimately, the individual operations which collectively constitute a running computer program are determined by the computer's machine language, but it is not straightforward to map source-code-level join points to individual machine-language instructions. Amongst other problems, the same source code may compile to different kinds of machine language for different operating systems (and will also vary according to which compiler is being used), so if machine language instruction-sets were chosen to provide rigorous semantics for join-points and pointcuts these semantics would have to be defined separately for each operating system and compiler.

An alternative is to consider abstract or *virtual* machines designed at least in part to provide this kind of semantic framework for defining program execution steps (at a finer scale than high-level source code). Another alternative is to define source-code locations not in terms of actual source code but in terms of that code as compiled to some graph- or tree-like representation (such as Syntagmatic Graphs). Our approach is compatible with either of these final two options, or indeed with a combination of the two.

Moreover, as we will discuss later in this chapter, pointcut-semantics can extend over different "scales" in source code (or program execution). Consider, for example, a remote procedure call (RPC) which is carried out by encoding a message with instructions to perform some calculation exposed as a web service on a remote machine, with the request itself encoded in HTTP. This HTTP content is not itself a procedure call, and would not adhere to the conventions of an ABI (but rather to the remote service API, or Application *Programming* Interface rather than *Binary* Interface), but in terms of code design and purpose such a web request can play a *role* akin to a local function-call. Also, remote and local procedure calls have enough structural similarity (in terms of their pre- and post-conditions) that one can certainly use the same sorts of expressions to describe calls of both kinds. Given these considerations, it is reasonable that a pointcut expression language could be designed to generalize to broader phenomena such as remote procedure calls, which (we propose below) involves a notion of multiple Syntagmatic "scales." In this guise pointcut expression languages would overlap with service-description (SDL) languages and with analyses to confirm that an RPC confirms to the target SDL protocol.

The practical consequences of models such as Coecke *et al.*'s hypergraph categories — and, we propose, variations such as Syntagmatic Graphs as in this chapter — are accordingly that these various lines of research can point to more effective code-representation strategies, which can then lead (among other results) to more detailed or powerful pointcut-expression languages and evaluators. The open question of pointcut expression semantics in Aspect-Oriented programming, as well as in Software Language Engineering in general — we should note that the notion of pointcut expressions is not limited to aspect-oriented paradigms exclusively, but also applies to debugging, static analysis, and runtime-reflection scenarios (*see* [6] and [28] for a good overview) — cuts across both source-code-representation theories and compiler/virtual-machine design. It is obvious that pointcuts are collections of "locations" in source code, but such a definition needs to be more rigorous, because source-code is a structured system (not just text) and locations represent sites within those structures. Applying models such as Hypergraph Categories or Syntagmatic Graphs allows us to ground these semantics by defining source-code structures in hypergraph terms. In effect, pointcut expressions thereby become a genre of *hypergraph queries* — designations of hypergraph sites — so that pointcut-expression languages may be treated (and implemented) as subsets of more general hypergraph-query languages. Syntagmatic Graphs, as a form of hypergraphs formulated to represent computer code, can thereby provide a semantic structure-space suitable for a general-purpose pointcut-expression semantics.

This discussion has therefore presented one specific rationale for pursuing Syntagmatic Graphs as a theoretical construct — we claim that such graphs form an effective framework for designing and implementing pointcut-expression languages, and by extension for code-modeling in general. However, our overarching goal is not code-modeling *per se*, but *data* modeling; so we will also clarify how code and data modeling can be interconnected.

3 Applying Pointcut Expressions for Data Modeling

Constructions designed for code models do not automatically translate to *data* models; however, code and data models often overlap because one of the specific purposes of computer code is to instantiate data models. This is particularly true if one adopts strongly-typed data-modeling conventions, where information spaces in general are decomposed into distinct data types. Almost all data *sets*, for example, can be structured as collections (which may be ordered or unordered sets of individual values/records/objects) whose elements are data structures which each have the same type, and accordingly have similar patterns of organization (e.g., the same set of individual data-fields). Dividing a data set into distinct

[10] Hence a *pointcut* is a set of *join points*, and *pointcut expression* is a formula for filtering join points into a pointcut.

collections-types and individual-value types yields the possibility of modeling some or all of these collections and/or "individual" values (which in general are not *one* single value but rather single data structures with some record-like structure, e.g., tuples of named attributes) via corresponding data types implemented in computer code.

Even though one could manipulate data sets and/or databases without such code-instantiations in some case (for instance via database queries), it is generally possible *at least in theory* to consider any data type internal to a data set and/or database to be convertible to data types in a programming language, with a corresponding implementation in terms of procedures such as constructors and field-accessors (some of the various procedure-roles were outlined in Chapter 5). This translatability is directly applicable to data-integration, because any integration scenario too complex to be achieved via database queries alone may be resolved by providing full-fledged programming-language implementations of any data types which need to contribute values to an integrated data model.

A complicating factor here is that different database models exhibit different levels of schematic rigor: in a JSON-based model where database objects may be arbitrary associative arrays, there is no guarantee that the key-names for an arbitrary object will match field-names for a recognized data type, so strong typing in such an environment is more difficult. However, data models can impose greater structure than required by a database engine itself; marshaling existing data to conform to a more rigorous model can then be one part of a data-integration pipeline. Whether or not concrete implementations are actually used for a given data type in a given project — and considering that integration workflows can include elevating more weakly-typed to more strongly-typed transpositions of data sources as necessary[11] — we can develop theories related to database management built around strong-typing assumptions with respect to partitioning database or data sets' contents via type-attributions, and expression of data-model constraints in type-theoretic terms.

In practice, a central tool in the data-integration arsenal is that of translating data types internal to a data set/database into implemented data types in some general-purpose programming language. Once that occurs — and in the scope of data-integration scenarios where such translation is useful — we therefore have a body of computer code whose explicit purpose is to provide computational resources manipulating data types that embody information present in a given data source (e.g., a data set or database). It is in this context that specifications in the *data model*, endemic to the data source in question, can be carried over to specifications in the *code model*, manifest in coding constructs and annotations pertaining to the procedures whereby units of data present in the original data source are manifest as typed values in the corresponding computer code. There are many ways in which data and code models in such contexts may overlap.

Strong typing also offers a productive theoretical perspective in which to discuss design patterns for data sets and data-integration strategies. Insofar as data sets are (in principle) strongly classified into types, a canonical (though not exclusive) mode of interacting with data sets is to (via queries of some sort) obtain typed values meeting certain criteria. That is, many queries against a data set will yield responses which take the form of a type-instance, or a collection of type-instances, or an iterator to step through such a collection. Type-instantiation thereby becomes an essential facet of the data set's interactions and query capabilities. In this sense, a crucial detail that we may consider at any site in a data-model instance (which we assume may be logically represented in graph/hypergraph fashion) is: *what typed value can be initialized with elements in the neighborhood of this site*? Shape-constraints provide a way to explicitly annotate graph-neighborhoods with type-initialization routines when appropriate. Let's call this the *shape-constrained initialization* (or *instantiation*) concern ("concern" in the sense of a core implementation detail for query engines).

Of course, in the typical scenario we do not happen to be at a site *a priori* and seek to populate a type-instance accordingly; instead, we want to find sites which allow instances to be constructed based on some criterion. In other words, we want to know whether a type-instance initialized from the site will have some property (some data field equaling some value, or lying in some range, etc.). Call this the *type-instance selection* problem. For a given site, we want to know both whether a type-instance can be constructed and, if so, whether it would be selected when filtered through criteria within current query or traversal specification. Note that query optimization should assume than the *selection* problem does not depend on actually constructing type-instances (otherwise we are not really providing query evaluation, only deserialization capabilities and deferring to application code the ability to select type-instances based on logical criteria). That is to say, a database and/or data set should be structured with enough information to resolve selection-problem questions without *completing* type-instantiation, though it should be determined whether instantiation is *possible.*[12]

Note that the necessary expressiveness of shape-constraints applied to selection/instantiation concerns expands according to the range of graph structures one seeks to target via a query engine. For a hybrid property/hypergraph model, for

[11] This may sound as if we are passing off as almost trivial what is often the most difficult part of data-integration (a trap lucidly and amusingly captured, in the Semantic Web context, by Clay Shirky: https://www.karmak.org/archive/2004/06/semantic_syllogism.html). However, transposing a data space to a canonical format is not any more complex a problem than integrating multiple strongly-typed spaces; so any computational environment wherein the latter is feasible should make the former feasible as well. When weakly-typed data sources are an integration hindrance, this suggests a two-step strategy wherein the sources are first normalized as strongly-typed assets to begin with, so that type-theoretic constructions are available to strategies at the second (integration) stage.

[12] In some cases, one could then use data leveraged for selection-queries to yield query-results that do not depend on type-instances; this would in keeping with conventional query paradigms such as SQL. In other words, we can return results akin to what a type instance *would* give for a data-field or some other computation if the instance were fully constructed from the relevant neighborhood. However, in query environments such as we discuss here we prefer to consider these "as if" constructions to be merely a special case of querying based on type-instantiation in conjunction with selective filters, where the step of constructing a full instance can be skipped in some cases but is still logically central to designing and implementing the query engine.

example, the process of initializing data points can be connected to type-instances through multiple pathways, including property-assertions, hypernode/subvalue relations, inter-hypernode edges, and foreign keys or similar "proxy" values that (under semantic interpretation) connect nodes to other nodes (which may not be explicitly connected via graph mechanisms). Each of these configurations have to be accounted for when providing kernel operations applicable to selection/initialization evaluations and also when specifying (and parsing) the query-language syntax (see Figure 4 for a visual illustration of these different modalities where information pertaining to a type-instance may be accumulated).

We claim in general that selection and type-instantiation problems are the most important desiderata for designing data models and the data sets that instantiate them. In the current context, note also that these are essential considerations for designing query-evaluation engines. At the core of any query-engine virtual machine should be logic for initializing type-instances from a graph-site and for testing hypothetical such instances against selective filters.

We postpone until Chapter 9 a more thorough discussion of instantiation and selection in these senses; for the time being, note merely that data-model features which render selection and instantiation problems tractable tend to propagate to code-model features in the context of code libraries that implement the corresponding data types. Thus, the structural properties of data models which are exercised in selection/instantiation query processing are also relevant to code-models, and tend to be visible in the numerous concerns where code and data models overlap. Some of these will be discussed in greater detail in this chapter.

3.1 Code Annotation with Units of Measurement

As a concrete example of data/code overlap, consider the problem of specifying units of measurement (this chapter will address dimensional annotation in terms of code models; we will revisit this issue in later chapters in the context of data models — particularly those connected to image-annotations and image feature-vectors). Units-annotations document the scales of measurement applied to data-points, which are therefore endowed with more structure than just single numbers. For example, in many applications involving computer graphics and digital documents (such as PDFs), *lengths* (on-screen or on-page distances) cam be expressed in several different units, including inches, centimeters, millimeters, and "points" (which are $1/72$nd of an inch), as well as numerous other scales (in the context of typesetting, and by extension document-viewing software, one considers also lengths determined by font-face designs, such as those based on the letters m and x, but such scale-units can be set aside for the current discussion, other than to note that a single dimension such as length may indeed give rise to a surprisingly large range of potential measurement scales).

One important feature of robust code is ensuring that procedures which operate on lengths (and other unit-denominated measures) check that input parameters match the procedure's requirements (one should not attempt to add inches and centimeters, for example). There are various strategies to enforce unit requirements. One technique is to simply recognize values measured according to different scales simply as different types. Then a procedure which expects measures in (say) points simply cannot be called with lengths in (say) inches, because such a call would not type-check (for the same reason that a procedure expecting integers could not be called with, e.g., a character-string). This forces calling code to explicitly check that local values are defined in units compatible to called procedures' expectations (converting between scales if necessary), which prevents scale-mismatch errors. (One limitation of this approach is that procedures playing the same role often then have to be defined multiple times, for various plausible measurement scales, which is facilitated by template programming but nonetheless leads to a proliferation of distinct procedure implementations, particularly if one tries to be strict about dimensional analysis — for example, the product of two inch-lengths would, scientifically at least, be an *area*, inches *squared*, and in general every multiplication-operation yields a return type different from the input types, so that there is a logically unbounded number of distinct types needed to represent arithmetic operations on a dimensionally-typechecked measurement-units system).

An alternative convention is to use a *single* type for a given *dimension* (regardless of measurement scale) but include in the type-instance a data point indicating which scale applies to the associated value. In other words, each instance has *both* a raw (scalar) value and a code indicating the scale applicable to that value (e.g., inches, points, millimeters, or centimeters). Unit-checking (and inter-scale conversions) can therefore be performed at runtime, which is more flexible (even if less foolproof then compile-type checks).

A different approach to scale-measure consistency is to simply define procedures as taking raw values (presuming that they are expressed as or converted to a given scale prior to the call) but ensuring through code analysis, annotation, and/or documentation that procedures will never be called with wrongly-scaled value. For instance, if a procedure requires that its input values be scaled in points, we can attempt to verify — by examining every point in source code where this procedure is called — that its values are guaranteed to be measured in points (rather than inches, say). In the code accompanying this book, for example, the procedure which is run when locating a sentence's extrema PDF coordinates receives a "baseline skip" value in (fractional) points, but (in this code base) that procedure is only invoked in the context of having parsed a metadata file where the relevant baseline-skip measure is deserialized. The deserialization code explicitly checks that the relevant numeric value is properly described (i.e., marked with "pt" according to LaTeX convention as indicator that points are employed as the length-scale); in other words, the deserializer only accepts strings of the form

"number-plus-units" where the unit declaration is restricted to "pt." This example demonstrates how scale-consistency may be enforced by simply exercising proper care at any point where unit-denominated values are obtained and then passed to procedures where scale-measures impinge on the procedure's outcomes.

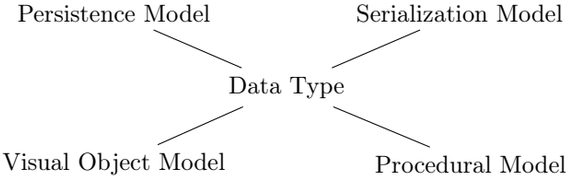
This kind of explicit code-verification can be confirmed by keeping track of code-locations where such unit-denominated values are read. Although some values are expressed directly in source code, the more common situation is that runtime values derive from one of three sources: database query results; deserializing markup which encapsulates values via binary codes and/or character strings; or GUI components where values are interactively entered by users.[13] In the GUI case, it is the component's responsibility to declare which units of measurement are assumed when translating the user-visible cues (e.g. a dial or slider's angular or orthogonal position) to a scaled measure (does a virtual thermometer model temperature in Celcius or Farenheit?). In the case of database queries, the database itself should specify units for non-scalar values. In the case of serialization, scale-units may be explicitly marked (by character strings such as "pt" adjoined to numbers or via separate notations, such as XML attributes) or else globally declared for a given markup format (e.g. via an XML DTD). In any event, the procedure which procures a value from a database, markup serialization, or GUI object can verify that it scales this value appropriately by either deferring to the origin-points documentation of its own units convention, if that is provided, or else by performing explicit checks when those are documented as necessary (e.g., number-plus-unit strings in markup).

In order to guarantee that code in this scenario is managing scale integrity properly it is therefore necessary to identify all procedures which input and output scale-delimited values, and confirm that (1) those which *input* such values specify the unit or units they can properly work on; (2) those which *output* such values either correctly assume that the values are properly-scaled within the origin-source from which these values are obtained or else explicitly check for the value's declared units, and perform scale-conversions as necessary; and (3) that every procedure in the first case (inputting scale-delimited values) is paired with a procedure in the second case (outputting scale-delimited values). That is, every time a value is passed as scale-sensitive *input* we must be able to locate where it originates as scale-verified *output*. Any of these checks are preconditioned on capabilities to identify source-code locations — that is, they all require pointcut expressions, or similar code-site designations.

The example of scale-delimited values therefore presents a use case of how data-modeling stipulations translate to code-modeling implementation-patterns. The units of measure by which particular data values or fields are expressed is a non-trivial detail of a *data* model. One way that such details become manifest as coding *requirements* is that the code which implements data types instantiating a data model includes sites where the corresponding unit-marked values are obtained and/or used, and these sites collectively represent source-code locations where specifications in the associated data-model would be accommodated and/or verified (assuming the code is implemented properly). In these scenarios, a pointcut expression language (querying for sites in *code*) becomes indirectly a modeling device for the underlying data, because the concretization of the data model in code-implementations serves to document, codify, and exhibit the requirements expressed by the data model. In short, pointcut expressions query (and their evaluators traverse) the code which *documents by implementation* (in the sense of a "reference implementation") specifications pertaining to the relevant data model. For this reason, pointcut expression languages can be considered intrinsic parts of languages for querying (or specifying/validating) *data* models, not just code models.

3.2 Documentation by Implementation

Insofar as code models *document by instantiating* an underlying data model, it is important to trace the features and aspects of a code body which are directly related to that data model. In general, the relevant code will be any components where associated data-instances are read from (or sent to) a data source, are manipulated and calculated upon, and are (by themselves or as transformed via calculations) shown (statically or interactively) to human users. A particular data-type may therefore be subject to several different transformations or adaptations to specific use-cases, in particular (1) persistence to a database; (2) sharing via serializations; (3) presentation via GUI components; and (4) exposed to different algorithms which extract information from the data (via statistics or any other analyses, depending on the kind of data involved). We can picture this as a diagram where a central data type is figured into four different contexts (database, serialization, GUI, and procedural capabilities implemented to support the analyses relevant to that form of data) — for sake of discussion, we'll call this sort of diagram a "Semiotic Saltire" ("semiotic" because the transforms of a given data type according to these various adaption-contexts is an example of what Joseph Goguen calls a "semiotic morphism" [19], and "saltire" because this graphic-design/iconography term denotes X-shaped patterns such as a central type with four type-morphism radii):



Pointcut expression languages extend to designations of type-transformations across different radii in this kind of "semiotic" intertype-space.

[13] This does not mean users actually *type* values; instead they may select values from a list, or by manipulating some GUI gadget; consider selecting the temperature on a virtual thermometer by turning a dial or pulling a slider "widget" where, say, clockwise or *up* signifies "warmer."

Consider, for example, the use of GUI components to interactively visualize values of a given type. For sake of discussion, assume a type correlated with its distinct GUI class expressly implemented to display values of that underlying type — in this case the relation between the central data-type and the GUI visual-object type is one of directly pairing off one type with another, because an instance of the underlying type is necessary to populate the GUI type. We can then "map" elements of one type to another — in other words, fields in the underlying type can be displayed as text/drop-down selection labels or visual gadgets in the GUI type.

Imagine an EHR form, with labels showing a patient's age, gender (maybe from a fixed selection such as Male, Female, Trans, Nonbinary, and so forth), first and last name, etc. Data-fields such as "age" may well map directly between the two types, but even with direct type-to-type correlations such inter-type mapping can be more complex in general. Suppose, for example, that a EHR form class incorporates a list of the patient's current prescribed medications. Because this list will be of varying sizes for different patients, the GUI representation would have to be not a single label (or some other single-valued visual) but rather an expandable subcontrol (such as a list or table display) where the software can add rows to list each of the medications declared by the patient's record. In the underlying data type this information is a *collection*, rather than a one-off value (recall our definition of mereotropic fields in Chapter 5).

Consequently, in the underlying-to-GUI "morphism" a *collection* field must be associated with multiple *procedures* on the GUI side, insofar as visual objects representing each element *within* the collection have to be accounted for — the graphical display for collections needs to expand and be generally modified (for instance, column-widths adjusted, in the case of table displays) to accommodate the variant-sized nature of the originating data source. In short, the underlying-to-GUI association is fully specified only by enumerating *multiple* procedures and multiple code points — ones where the GUI object is modified to accommodate dynamic variations, such as variant-sized collections, in the underlying type — such that the identification of these code locations is again an example of pointcut expressions.

In sum, pointcut expressions are intrinsic to rigorous documentation of at least two of the four points (that we have discussed so far, i.e., visual objects and procedural models) in the "Semiotic Saltire" diagramming how a single data type morphs into different structures to accommodate different coding requirements. Expressing a data type in GUI form is of course relevant for end-user applications, but this is also relevant for data integration because some data-integration scenarios may require users to expressly examine data values to confirm an integration-computation — a common scenario in cases where, for instance, provisional machine-learning outcomes are subject to confirming human reviews. 15

Moreover, pointcut expressions can also be applied to *serialization* at least in the context of (what we can call) "grounded" serialization, where serializing markup is annotated with metadata indicating how the serialization format connects with an underlying type system. A simple example of "grounding" in this sense is the annotation that a markup region serializes a single instance of a specific data type, but more complex grounding declarations can relate a serialization artifact (such as the encoding of a specific data field) to one or more implementation code-procedures (such as the accessors which collectively define the interface for a encapsulated field, which can be seen as procedural manifestations of that field as it is described in a data model, with a single data-model field mapping to multiple accessor-procedures). In this sense serialization-grounding metadata and pointcut expressions share similar procedure-denotational building blocks, aside from the possibility (e.g. via virtual properties implemented through procedure "views") that grounding would reference pointcuts directly.

This discussion therefore suggests that pointcut expressions cut across data-to-code-modeling for three of the four points of the "Semiotic Saltire." There are scenarios where pointcut expressions may apply directly to *database* logic (the fourth angle of the Saltire), but we will argue that the deeper connection between pointcut expressions and database concerns lies in the underlying hypergraph model, as will be explicated in the next full section. Prior to that discussion, however, it is relevant to point out certain additional areas of overlap between pointcuts and procedural interfaces, aside from those already alluded to above, particularly in contexts such as runtime reflection and remote procedure calls.

3.3 Annotation-Based Reflection and Procedural Binary Equivalence

We said earlier that pointcut expressions "locate" points in source code. However, the mechanisms through which such expressions are evaluated may depend on *annotations* applied to the code, so the scope of a pointcut expression language (or a hybrid code annotation and query language which includes pointcuts as one construct) encompasses annotations which provide the infrastructure for pointcut expression targets, as well as pointcut expressions themselves. Although code-annotations are potentially an intrinsic language feature — Java, C# C++ and other mainstream enterprise-level languages all have some built-in annotation (aka attribute or "decoration") mechanisms (C++ with the caveat that annotations are compiler-specific and not uniformly standardized, although that is gradually changing [26]) — this discussion will focus on the (potentially more flexible) use of *external* annotations, where a given code base is accompanied by a distinct code body (potentially in a different language, one unrelated to any standard or compiler/runtime capability associated with the original language) that describes properties, requirements, metadata, or runtime-usable documentation about the original code. A good case-study in the use for such annotations is support for dynamic method calls: providing an application (or in general a code base) the capability of invoking a procedure by expressing a textual description

(most simply just the procedure-name) of the desired procedure, along with a textual representation of the desired arguments. Exposing procedures to textual runtime-reflection in this sense has applications for unit-testing, for embedded scripting languages, for fine-tuning the behavior of a running application, and for documenting data-model requirements (because we can dynamically examine requirements declared for different code elements, such as data types, data fields, and procedures, allowing these requirements to be interactively documented as a feature of the associated software application).

To develop the theory of dynamic runtime procedure-calls, consider first that, without runtime reflection, procedures can only be *statically* invoked; in other words, a procedure runs only if there is a point in the source code where it is instructed to run. Of course, applications do not in general know exactly which procedures need to be performed in order to respond to the user's request, since one cannot know what exactly the user will do (the only exception to this principle might be behind-the-scenes programs, such as those which run when an operating system first starts up, but these programs do not typically interact with users at all, and therefore they are not technically *applications*). Most real-world programs, in short, do not follow a fixed sequence; instead, after some preliminary startup, they enter a mostly inactive state (e.g., an *event loop*) and wait for user input. They then respond on the basis of what the user does (clicks the mouse at a given screen location, types on a keypad, etc.).[14] The user's actions are typically called *signals*, and the steps taken by the application to usefully respond to those actions are *handlers*. Handlers generally call different procedures based on the nature of the user's actions, but all logic involved in translating a given action to one or more procedure-calls has to be hard-coded in the application code.

Dynamically invoking procedures via runtime reflection varies in this scenario because the "signal" which the application responds to — assuming the application is in a passive state waiting for input — is not a mouse or keypad event (or other user device) but rather an explicit description of the desired procedure (so the procedure does not need to be inferred by parsing a less-specific input event). For example, when a user clicks on a "save" button they may very well be unaware that satisfying their intended request requires calling a function called (something like) "**saveFile()**"; but a dynamic runtime-reflection invocation would explicitly designate the relevant procedure (e.g., **saveFile()**) by name. To be sure, in most cases users would not themselves type in the procedure they want to invoke (although some software, such as **emacs**, works expressly by users typing instructions). On the other hand, one design principle for adaptive user-facing software is to map user-actions and related GUI elements (such as buttons and menu items) to text descriptions in lieu of hard-coded procedure calls — a "**save**" button may be mapped to a text string (or even a text file somewhere) containing the instruction "**saveFile()**," which ends up calling that procedure. Using such a text-string as an intermediate evaluation-step would permit the application to be tweaked by modifying the text to call a different procedure, or multiple procedures, as needed — typically the user would not modify the text directly, but it could be changed during some sort of application upgrade, or to sync the application with an external cloud service, or some other post-install adaptability feature.

Even when runtime-reflection procedure-calls are not implemented for adaptive purposes along these lines, such capabilities may still be relevant for testing, documentation, and requirements engineering. For example, if a data model stipulates certain data-management requirements (such as unit-scales as discussed above), one can verify that the code base handles those requirements properly by executing a test suite which dynamically calls procedures that are affected by these specifications, verifying that they manage values (and correctly handle malformed data) properly. Aside from double-checking that the code base is properly implemented, the code supporting such testing capabilities provides documentation of the underlying model (according to the principle we earlier referred to as "documentation-through-implementation"). In general, runtime-reflection promotes documentation-through-implementation by allowing documentative descriptions or scripts to manifestly demonstrate features of an underlying data model by invoking procedures where these features are relevant. In short, one strategy for rigorously enforcing data models through code models is to provide external annotations sufficient to allow data-model-sensitive code to be tested via external invocations.

To be sure, external invocation requires more than just describing a desired procedure: the runtime-reflection code needs to parse some encoding of arguments *to* the procedure, and moreover this runtime-reflection code needs to set up the procedure call so as to *mimic* the Application Binary Interface (ABI) which collectively represents the compiler, operating system, and programming-language specifications for how procedures may be called at the machine-language level (this is less applicable to dynamic languages such as Lisp, or indeed languages such as Java or C# with built-in reflection capabilities, but it applies directly to lower-level languages such as C and C++).

We'll concentrate on C++ in particular: there are several different strategies for supporting C++ reflection, each with distinct trade-offs. One powerful solution is to use an expansive runtime-metadata framework (particularly LLVM) which allows almost any procedure to be exposed to a reflection system (and therefore invoked dynamically from an external call-description); the problem with libraries such as LLVM is that they tend to make code bases much more difficult to

[14] Neither theories emphasizing concurrency — such as Petri Nets or π-calculus — nor conventional lambda calculi seem especially well-suited for modeling program flow in GUI applications. Responses to user actions — which would typically span multiple procedures — nay run in parallel (because the user may oresent new action before a prior response has completes), but the specific modeling challenges in this context are not centered on concurrency *per se*, but rather on indeterminacy vis-à-vis *which* action the user will perform and the fact that one single overall application state can be affected by any number of possible actions. Perhaps some not-yet-formalized hybrid of (say) λ and π calculi could appropriately model the semantics of GUI applications in this sense. A good foundation might be so-called "object" Petri Nets (*see* [23] or [24], for example).

compile and install than the equivalent code base on its own, which limits the feasibility of distributing the entire code base in source-code fashion.

A different solution is typified by Qt (the most widely used cross-platform C++ Application-Development framework), which relies on preprocessing certain source-code files; this approach is limited by the necessity of that preprocessing step and also by the restriction that only procedures (and likewise data-fields) which fit certain technical criteria may be exposed to the Qt reflection system.

The AngelScript embedded scripting language is a good example of a third alternative, where the language runtime translates certain runtime logic directly to assembly language, adapted for a number of different operating systems and compilers — the limitation of such a solution is (first) that such code becomes, for all intents and purposes, completely opaque for any programmers who are not intimately familiar with multiple target platform's assembly instruction set and (second) that it demands fairly complex programmer-supplied code to describe exposed data types (as well as procedures) to the AngelScript runtime.

A fourth scenario is represented by Lisp dialects such as ECL (Embeddable Common Lisp) [3] and Clasp [48], where C++ functions can be exposed to Lisp code via a "foreign-function interface" (or, at least in Clasp, can potentially be compiled directly into the Clasp system as hidden procedures in object-file scope). These approaches require that C++ bridge code be able to work directly with Lisp constructions such as pointer-fixnum unions and cons cells, which leads to very non-standard-looking C++. Such problems are characteristic of the general approach taken by most C or C++-interoperating scripting languages, which is either a fifth alternative (when enumerating C++ reflection strategies) or a variation code pre-processing; namely the implementation (either manually coded or automatically derived via a pre-processing tool) of "wrapper" procedures which marshal data back and forth between C++ (or, likewise but less complexly, C) and the relevant scripting language (Python, Ruby, JavaScript, etc.).

In this book we are proposing (and illustrating in prototype fashion) a framework for general code annotation and pointcut-expression declarations/evaluations, which includes a system for resolving procedure-descriptions to ABI-compatible invocation structures at runtime. While any of the strategies just enumerated could potentially be applied to such a framework, we propose a system based on the notion of *procedural binary equivalence*, which eliminates almost all of the problems associated with the above-mentioned reflection paradigms while significantly reducing the programming effort (and boilerplate code) needed to provision applications with runtime-reflection capabilities. The central observation for this technique is to note that from an ABI perspective the machine-level code is not concerned with the actual types accepted by a signature, but simply with the binary profiles (the byte-lengths and, potentially, constructor/destructor/cast functions called on temporary values) of values conformant to those types. In other words, a pointer to a function of one type can safely be cast to a related function-pointer-type so long as the types constituting the latter type's binary and temporary-value aspects are the same as the actual types of the targeted procedure. Via such equivalences, the expansive range of types which enter into procedure signatures can be scoped down to a much smaller collection of "canonical" types, or (the terminology we propose) "pretypes," so that we construct equivalence classes of many different functions that are distinct on the type/signature level but binary-equivalent on the "pretype" level. Exposing a procedure to runtime-reflection thereby entails simply classifying the procedure within one of the binary equivalence classes.

This technical framework comes with some caveats. First, binary equivalence is (to some extent) compiler-specific, so adopting these techniques involves either knowing *a priori* that a code base is specifically targeting a given compiler (GCC, say) or else using runtime and/or compile-time checks. In addition to compiler variation, binary equivalence may also depend on factors endemic to the code base such as whether smart pointers are utilized, or whether certain specific data types are prominent (such as **QString** or **QVariant** for application composed via the Qt framework); the actual binary equivalence-space may accordingly vary from one code base to another.

A second caveat is that combinatorial expansion of the number of distinct binary equivalence classes needed to support relatively fully flexible procedure-exposure can potentially make compile-times too long. It is, in short, unrealistic to expect *every* procedure in a code base to be directly invoked using techniques described here; for procedures with unusually complex signatures, it may still be necessary to provide wrapper code (e.g. instead of a function which takes variant-length argument packs provide a version taking a reference-to-vector, expose this latter function to reflection and call the former function from the latter). Nevertheless, binary-equivalence techniques should make it possible to greatly reduce the number of occasions in which wrapper code needs to be written or generated. For procedures whose signatures fit within a given binary-equivalence scheme, exposing the procedure to a runtime-reflection engine would then be as simple as annotating the procedure with a single numeric code matching the procedure to an equivalence-class. The practical details of such annotations are outside the scope of this chapter, but are documented in some detail in the book's supplemental materials.

With such a reflection system in place, it is possible without significant modification to provide runtime-reflection to a code base, particularly one implementing a "documentation-through-implementation" project wherein individual procedures (and their properties declared as code meta-data) instantiate and exhibit data-model specifications. Such an approach raises the possibility of examining code in the vicinity of specific procedure-calls to verify and/or document the con-

nections between code and data models — i.e., test that the code base accurately instantiates an underlying data model and/or document the data model by expositing its implementation in the code. An example of such documentation might follow from the above discussion of unit-scale verification: the code's logic for checking scale-measure properties on input/output values and lexically-scoped variables could be tested by externally simulating calls to procedures which input scale-delimited values as well as calls to the outer procedures which call such procedures in their function body. In general, an outer procedure may go through several steps to ensure that parameters passed to a called procedure are in accord with data-model requirements. A useful design pattern is to implement this preparatory logic in anticipation of the code being verified/tested by external runtime-reflection calls; here both outer and called procedures would be exposed to the reflection engine and, moreover, a logging or documentation mechanism could be implemented for the code *surrounding* the inner procedure-call to clarify steps taken to ensure data integrity.

3.4 Meta-Procedural, Procedural, and Sub-Procedural Syntagmatic Scales

The above discussion points to a larger topic in the overall theory of modeling computational processes: a computational step which functions logically as a single procedure-call may encompass several additional steps before or after the actual procedure-call involved, as data entering (and returning from) that procedure is assembled and/or validated. This expanding-outward phenomenon is particularly evident in the case of *remote* procedure calls, which are more complex than simply temporarily delegating execution to a different subroutine than the one currently running. A procedure which invokes a different procedure on some remote service (a cloud service, say) may be unable to simply pass raw values (there is no ABI binding remote procedures), but instead may need to encode inputs in (say) a textual markup format, and similarly deserialize data received as response. More often than not the calling procedure does not wait for the remove call to return, but rather provides an anonymous procedural value to act as a return-handler. So in a typical scenario remote procedure calls involve data serialization/deserialization and intermediate procedural values (moreover, on the remote endpoint, clients do not typically requested actual procedures directly but invoke *services*, or what may be called *meta-procedures*, which perform data-marshaling and delegate to *de facto* procedures based on the provided request). In short, remote procedure-calls involve preparatory code on both endpoints which serve as a logical framing of actual procedure calls with additional data-marshaling/handling.

As noted above in the context of Syntagmatic Graphs, a single procedure-call is built up in stages, even though from the perspective of high-level programming-language syntax a call can be expressed as one single expression-unit (the intermediate stages being largely invisible in the surface-level code but implicit in the compilation of this code to machine and/or virtual-machine representations). On the other hand, as the examples of scale-measure checking and remote-procedure calls demonstrate, sometimes the multi-stage framing of a call propagates to computational steps that *are* explicitly visible in high-level code, and sometimes (as in remote-procedures) the logic actually involves *meta*-procedures which augment the structures of procedure calls themselves with extra data-marshaling and handling logic.

Formally representating the temporal sequencing involved in performing and managing a single procedure-call can therefore concern several different levels of code "granularity," that "below" the original source-language (as in compiler-related intermediate representations) and "above" (as in networking with remote-procedure providers) as well as on the level of the source-language. This is why we propose a sufficiently expressive code-representation system (e.g., Syntagmatic Graphs) to embody coding structures at each of these levels. By extension, code-validation via runtime-reflection can test/document coding assumptions by testing implementations at each of these three levels. This is a further rationale for using code-modeling as a data-modeling device, and also points to another feature which a code-annotation language should support: the capability of expressing code structures at the three levels (from what we might call "sub-procedural" to "meta-procedural") described here. These represent different coding environments where the "procedural" aspect of the "Semiotic Saltire" would be in effect.

In effect, the general model of computation that we are operating with here — where the lambda-calculus notion of "reduction" is replaced by a process-calculus-like "binding," implying a connection between descriptions of computataton and epistemic semantics based on "information content" — this model can be manifest at several different levels or scales of computations, insofar as these are identified in computer code. We'll call these subprocedural, procedural, and meta-procedural *Syntagmatic scales* so as to have convenient terms for such levels. Again, the main goal of introducing and arguing for special terminology is to established structural features that will be useful for query languages; insofar as a point-cut expression language can be incorporated as one part of a hypergraph description/siting language, terms for different Syntagmatic scales can become terms used and recognized in the query language.

3.5 Case Study: Annotation and Image Markup

This section will conclude by outlining how themes discussed in this chapter can be demonstrated via one specific data format and code library, namely the Annotation and Image Markup project (AIM) [7], [40], [46], [49], [29]. Originally part of the cancer Biomedical Informatics Grid (caBIG), AIM standardized the encoding and sharing of image-annotations used by radiologists and other pathologist to diagnose conditions (tumors, for example) from medical images. An image-annotation in this context is typically a geometric description

or designation of a "Region of Interest" (RoI) which the examiner believes is diagnostically significant. Regions of Interest can be demarcated in simplified geometric forms (via circles, rectangles, and so forth) or isolated by creating overlays which themselves have an image-based format (the simplest example being a black-and-white picture where white pixels represent the region/foreground and black the background), so as to represent an RoI more granularly.

At present we will initiate merely a brief overview of AIM and image-annotation concerns, deferring to subsequent chapters to continue our analysis in greater depth. This book's supplemental materials will republish code within the AIM C++ library (AIMLib) with minor changes for modern compilers (the original code was developed around 2013), including parsers for consuming AIM XML as well as demo files obtained from The Cancer Imaging Archive (TCIA). More substantially, we provide a client library to simplify the process of loading annotations as runtime objects from AIM serializations. This AIM client library (published as an **aim-client** Qt project), which wraps some of the functionality exposed by AIMLib, hopefully demonstrates how specifications within data models can be translated to requirements or functionality implemented within code models. The **aim-client** code provides several forms of runtime-reflection which supplements the underlying AIMLib code base, such as dynamic procedure-invocation and also type-reflection capabilities. For example, **aim-client** packages standard C++ **typeinfo** functionality into an AIM-specific type-reflection class that manages details such as type-name demangling. A typical use-case for type-reflection, demonstrated in the book's accompanying code, involves casting an abstract **MarkupEntity** pointer (a base class that is specialized for different annotation-shapes) to its specific annotation type. Such reflection capabilities simplify the process of extracting the core annotations from the full suite of details encoded via an AIM annotation collection (specifically, an instance of the **AnnocationCollection** class). The demo archive also provides extensions to several existing 2D or 3D software applications that can furnish origin-points for images to be annotated, along with a GUI-based annotation program; coding constructs within these components serve to illustrate certain themes discussed in this chapter — such as, identifying interrelated procedure-groups (e.g., for reflection) — with these points discussed via source comments as appropriate.

Bioimage annotations intrinsically represent three different sorts of information — geometric data constituting annotations themselves; presentation data governing how annotations are made visible to human users; and bioinformatic content which provides an interpretive context for the image data (e.g., a description of the specific tissues, organs, or organ systems visible in a bioimage). One step in semantically organizing the totality of annotation information is to clarify which data points serve which roles. For example, in the **GeometricShapeEntity** class, data fields such as **LineColor** (and line thickness, opacity, style, etc.) are presentation details, whereas the annotation's vertex coordinates (a sibling class to **GeometricShapeEntity**) are underlying spatial data (logically separate from any visualization). The **aim-client** library uses several techniques to render these semantic details more explicit, relative to the underlying AIMLib code; for example, we provide stronger typing for geometric and presentation details (colors and line-graphics specifications are objects rather than C++ strings, for instance).

In the following chapters we will analyze in greater detail both AIM specifically and the data models associated with image-annotations in general. In particular, this topic provides concrete examples of how data-modeling issues often become manifest in terms of code-modeling decisions (such as procedural pre- and post-conditions) and logical relations amongst implemented procedures, which has been a central theme of the current chapter.

4 Hypergraph Representations for Data-Persistence Bridge Code

This chapter's discussion has addressed the features and affordances of hypergraph data modeling, particularly via Syntagmatic Graphs and pointcut expressions, in three of the four "radii" of the "Semiotic Saltire": inter-type connections for GUI components paired with underlying data types; "grounding" for underlying types' serialization; and Syntagmatic Graph models of procedure-call logic at meta-procedural, expression-level, and subprocedural level (fully elaborating a theory of these three levels would require additional discussion outside the scope of this chapter, so we leave the terminology not-too-rigorously defined at this point, but hopefully examples such as remote procedure calls, verifying scale-unit consistency, and bindings/temporary objects accumulating data for procedure-calls, respectively, illustrate the kinds of computational phenomena applicable to these three levels). This final section will address hypergraphs in the context of the fourth Saltire aspect (data persistence).

As earlier in this chapter, we assume that data and code models are tightly interconnected — specifically, that data models are instantiated by code libraries wherein data types that are intrinsically or schematically described in a data model become explicitly implemented in data types coded via programming languages, particularly general-purpose languages such as Java or C++. As such, data *persistence* can be defined as the process of storing typed values which are instances of types implemented in a code base so that those values may be used in the future (and/or shared with other applications employing the same code base). As intimated above, there need not be a direct isomorphism between databases and implemented types — for instance, it is not necessarily the case that every table in a relational database can be paired with a type whose fields correspond to the table's columns — but treating databases as persistence engines for a project's data types is a useful perspective from which to examine data modeling, sharing, and representation patterns.

The correspondence between databases and implemented

data-types is especially strong in the case of *object* databases, which are designed so that almost any values used by an application can be directly stored in a database, with little extra code needed to marshal data between different representations. While object databases have been explored for at least a quarter-century, they have not emerged as mainstream architectures comparable to either SQL (relational) databases or predominant NoSQL models. Perhaps the mismatch between live program memory — how data is represented while an application is running — and the structure needed to efficiently persist and then retrieve data, evinces a gap too wide for object databases to serve as viable replacements for SQL or for the most popular NoSQL architectures [4, pages 6*ff*].

Instead, the optimal database architecture probably lies in between the rigidity of the SQL model, where the structure of information within the database is significantly different from the natural structure of programming-language data types, and object databases where database structure tries to directly mimic in-memory data layouts. Successful database systems find a reasonable balance between employing structures which are conducive to efficient data storage and query-evaluation and, at the same time, minimizing the degree of difficulty apparent in the translation of application-level data types to data stored within the database itself.

Concerns about data *layout* and *query evaluation* have the consequence that application data must be structurally transformed in order to be stored in a persistent database. It is worth pointing out the specific complications introduced by layout and query-evaluation concerns in order to motivate database engineering solutions to the problem of bridging application data types to in-database structures. A database first and foremost holds data values for an indefinite amount of time, but such data persistence is only one part of its full suite of roles and requirements.

In and of itself, constructing a persistable representation of most data types is not especially difficult: most programming environments have some sort of "binary stream" encoding where numbers are directly represented in their binary form, character strings can be encoded by assigning a number to each distinct symbol in the strings' character set, and composite structure can be encoded by encoding each of their parts (e.g. date-times can be encoded as a single value, such as milliseconds-since-epoch, or as a tuple of second-minute-hour-day-month and so forth). In Qt, for instance, the **QDataStream** class can serialize most data structures in binary form by passing individual fields into the "stream" (e.g., **qds « person.name() « person.age()** and so on for a **qds** datastream and hypothetical **person** class). Using classes such as **QDataStream** (or their equivalents in other languages/environments) it is not difficult to create binary packages representing most or all of the data in a given typed value, so that this binary encoding may be stored within a database and eventually reconstructed as a duplicate of the original value, yielding data persistence without much programming effort. At this level, then, data persistence does not require a complex database architecture.

There are two main problems with this simplistic approach to persistence: first, the resulting data storage might not be *efficient*, with the consequence that individual values take up more memory-space than needed, which can be a problem if a database extends to the scale of millions (or billions) or distinct values; and, second, the problem of *finding* individual values from a large data-collection. It is one thing to have some value in application memory while running the application on one occasion; it is a different matter to re-find that value (maybe years) later amongst millions of other values.

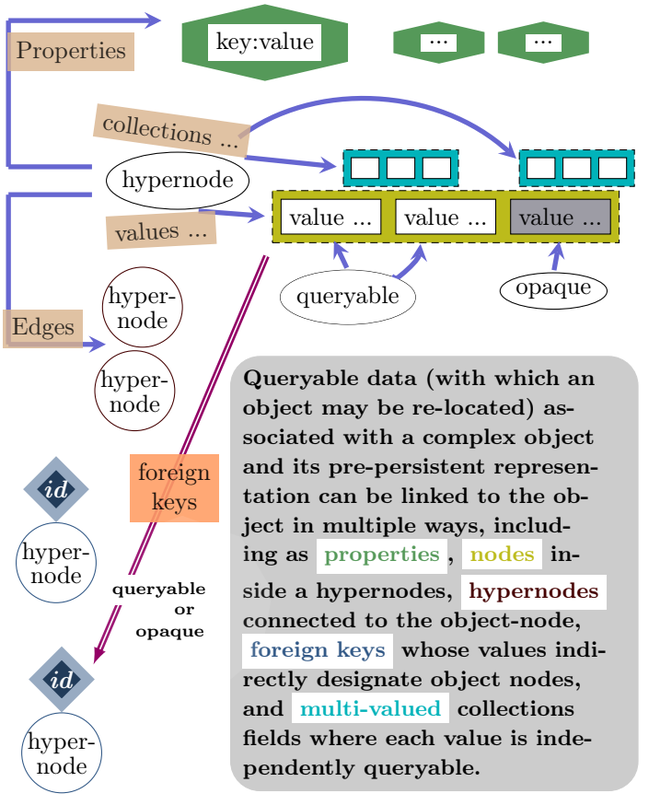
When values are binary-encoded via classes such as **QDataStream**, the data within the stream is, in general, "opaque" to the persistent database engine: the engine cannot "read" values within the stream, and from that perspective the stream is only a sequence of bytes with no specific meaning. In a working database at least some content associated with a given aggregate data-value must be exposed to the larger engine; if a value represents a *person*, say, the engine would probably need to identify the person's *name* so that one could find a person's record if one knows their name in advance. In other words, although *some* of the data in a given application-level typed value may be exported to a database simply as an opaque binary stream, at least some part of this data must be expressed in a queryable fashion, one which is tractable to the surrounding database engine.

Accordingly, with an instance of an application-level data type — or in general a type implemented within a code base associated with a corresponding data model — one desideratum for database persistence (which can be formally addressed in the data model) is how to divide the data within each specific type into "opaque" data which does not need to be queried (outside the context of a fully-instantiated instance of the data type) and *queryable* data which might be used to locate that specific value against many others. For instance, in an EHR database almost certainly a patients' first and last name would need to be queryable because it is normal to attempt to locate a person's records by providing their full name (if a patient calls their doctor they are more likely to identify themselves by name than by some in-hospital id number, for example).

Queryable data, moreover, can sometimes take the form of individual-value fields (such as first and last name) but can also take the form of collections. For instance, given an EHR database which tracks medications, it is reasonable to form queries based on the one-to-many relationship between a patient and the medicines they have been prescribed; e.g., to list all patients who are currently taking a particular pharmaceutical. Such a query can only be evaluated if the list of medicines prescribed to each patient is queryable in the system (and not opaque data that can only be read by reconstructing a whole patient-object at the application level).

In short, information stored in a database can be divided into three groupings: *opaque* (non-queryable) data,

Figure 4: Query-Aware Prepersistent Representation



non-collections queryable data, and collections (multi-valued) queryable data (which we might call "multi-queryable"); each of these forms of data have different layout and management requirements. An effective database engine will store data in a concise manner (to use as little memory as possible) without making queries inefficient (it must be possible be locate queryable data fields relatively quickly). Different database engines use different techniques to optimize memory-layout; the details of such optimizations are outside the scope of this chapter (and are not the direct concerns of application code in general), but we can point out that application data models do need to identify which data fits into which medium (opaque, queryable, and multi-queryable).

Different database engines, aside from identifying "queryable" data (typically without actually employing this term), also have different strategies for *relating* queryable data to its associated overall data-type instance. For example, consider the process of locating a patient record using the patient's full and last name. In SQL, the two-column conjunction of first and last name might serve as a compound key uniquely identifying patient-records in a patient table. In a property-graph database, the compound first-and-last-name value may serve as a property annotating a patient-node, utilizable to locate that specific node (potentially within a large graph will many other patient nodes). In an RDF graph database or "triplestore," the combination of a fixed last-name and first-name node-value (together with "last-name" and "first-name" as edge-labels) might provide "shape constraints" which filter nodes down to a unique match yielding the desired patient-node. In short, the graph and/or tuple-structures and terminology which govern how queryable data serves as an index to retrieve records from a database differs according to the underlying database architecture.

A given application and/or code base therefore has multiple options with regard to the requirement of modeling how a data-type instance should be registered in a database in queryable fashion. Queryable data may be exposed as a *property* (uniquely identifying an overall type-instance), as a node within a hypernode (in the case of hypergraph databases), as a foreign-key or an index value (in a relational context), and so forth. The specific structures and terminology will depend on the specific database which the application uses for data-persistence. However, abstracting from the details of different architectures, we can consider an *abstract* persistence model which would recognize structures such as properties and hypernodes as generic patterns which can be translated into the architecture-specific forms of different databases as a concrete instantiation of a more general abstract model (see Figure 4).

Presuming this possibility, then, we propose the notion of a general-purpose *persistence model* which fits into our earlier "Semiotic Saltire" picture, where the persistence for an individual data type would employ properties, hypernodes, queryable and multi-queryable values, and similar formations to construct a representation of how data type-instances should be encoded when represented in persistable form. This persistence model can be thought of as a data-structure recipe targeting a "virtual" database engine which incorporates features of specific database architectures, such as property graphs and hypergraphs. By expressing data-persistence logic in this generic fashion, the data-model thereby includes as one part a persistence-model which provides an abstract picture of how types within the data model should be persisted (taking memory-layout and queryability concerns into account). Specific code libraries could then translate this abstract persistence model into concrete representations suitable for specific database engines which may be used as back-ends. (One benefit of this generic model is that the persistence logic is not tied to a specific back-end; the same abstract model can be compatible with many different back-ends, so that different back-ends may be selected as application evolves and different database features become more important; for example, over time, it might be necessary to select a new database engine which places greater emphasis on scalability, or on automated backup and replication).

Earlier in this chapter we suggested that a *pointcut expression language* can be grounded in hypergraph semantics in the sense that, insofar as source code is modeled via Syntagmatic Graphs, pointcut expressions become in effect one form of hypergraph site-location/traversal query. In the context of data persistence, according to the "abstract" model just presented, we similarly suggest that a *bridge language* for representing data type-instances in pre-persistence forms is similarly grounded in hypergraph semantics in that these bridge representations are an example of hypergraph serializations. In

this sense, such a pre-persistence language (referring to representations that constitute an intermediate step derived from in-memory data, but from which in-database structures can subsequently be constructed) can be paired with pointcut expressions and grounded-serializations as representational concerns (relating to different radii on the Semiotic Saltire) that all translate to hypergraph queries (and therefore can be implemented in terms of a common hypergraph-query parsing and evaluation engine). These three paradigms (pointcut expressions, grounded serialization, and pre-persistence bridge forms) are three aspects of the prototype hypergraph-query language which this book examines as a case-study in data-integration tools.

As just laid out, bridge pre-persistence is distinct from pointcuts and runtime reflection in general (despite their common hypergraph background). However, the data-modeling issues which should be addressed when selecting a pre-persistence scheme for individual data types does overlap with code-model concerns that would be exercised via pointcut expressions. For example, annotating data fields as queryable or multi-queryable tends to impute on those fields semantics which are also apparent in construction patterns for the type in question (e.g., a field which uniquely selects a given type-instance in a query environment where many such instances are present — one might call such an environment "crowded" — may well be a field whose value cannot be default-constructed, so that some mechanism should be in place for indicating that fully-constructed instances must initialize that field in particular). Likewise, annotating a field as multi-queryable implies that it is a collections formation with a set of state-management procedures (insertions and deletions) whose pre- and post-conditions would be specified by a rigorous code model. All told, then, in practice, design patterns for pre-persistence and pointcut-expression use-cases for their common hypergraph description/query language would tend to intersect.

4.1 Multipart Relations with Roles

Another facet of pre-persistence involves optimizing the database layout to represent relationships between relatively complex objects; that is, between objects which have enough internal structure where it is unlikely that one would usefully be encoded as a single field or property inside the other. In relational databases, this is the kind of situation addressed by foreign-key references linking two tables. In graph databases, such scenarios lead directly to labeled edges connecting two nodes (or hypernodes). In principle, any node in a graph database can be connected to any other node, so it is straightforward to connect any two data-values with however much degree of internal structure they may have, so long as a single node (or hypernode, in hypergraph contexts) encapsulates or delegates reference to each value respectively.

A familiar problem in graph-database data modeling, however, concerns *multi-part* relations, where fully asserting all details of a given instance of the relation requires naming more than two values (or objects, nodes, etc.). A popular example for examining multi-part relations is how the familiar binary relation of *marriage* can be generalized to a more complex structure if we identify details such as *whom* (officially) married the bride to the groom. Similarly, a *divorce* is a multi-part relation which (minimally) includes a husband and wife, but also requires a marriage contract of some sort which is voided by the divorce-event (and therefore such details as a marriage date, with the divorce then marking a temporal range when the marriage was in effect, followed by a time when it is annulled; plus perhaps details confirming the prior marriage's legal status, because only an "official" marriage can be superseded by a divorce). For sake of discussion, consider marriage and divorce dates, as well as identifiers for the officiating parties who certified the marriage and the later divorce, as added details marking *divorce* as a multi-part rather than binary relation.

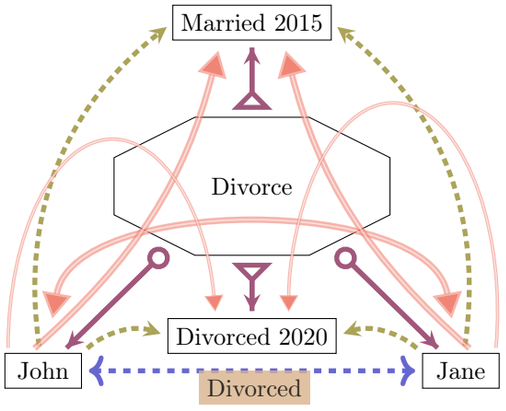
Different data-modeling strategies address multi-part relations in different ways. In general, the relation in question can either be treated as a compound whose individual components are distinguished in terms of the roles they play in the larger whole, or as a data structure which functions as a peer to the data-values connected by the relation. In a graph database, say, where nodes represent "values" or "objects" and edges represent relations, multi-part relations can be modeled as *nodes*, in which case the elements participating in the relation would be represented in accord with any other node-structure, as data fields or nested-nodes encoded via whatever structuring representations the database architecture supports (properties, hypernodes, shape-constrained neighborhoods, etc.). For example, we would have a "divorce" node whose data-fields represent the relevant husband, wife, marriage date, divorce date, certifications, and whatever related data; nodes representing the two spouses would then be connected to this divorce-node to model the spouses being divorced. The fact that divorce is also a *relation* — we can say that John *is divorced from* Jane — is then implicit in the existence of the "divorce node" to which both parties are connected: the structure does not explicitly notate this relation as a connection between, say, the *Jane* and *John* nodes.

Alternatively, the notion of multi-part *roles* — which is a prominent feature of graph databases targeted at AI applications, such as **Grakn** — tries to more directly model what we perceive to be the real-world semantics of multi-part relations, where *divorce* (say) is indeed a *relation* between two parties even as it has additional structure. We can accommodate the extra structure by noting that different parts of the relation fit into the larger whole in different ways: so the divorce has one part which is a husband, one part a wife (of course same-sex divorces are a also possible; we speak in gendered terms simply to refer conveniently to the distinct parties), other parts are dates and certificate ids, and so forth. The complete relation is semantically the totality of individuals or details playing each of these implied roles.

In effect, multi-part relations — which we will refer to for

Figure 5: Multi-Relations and Edge-Tangles

"Divorce" represents both a data structure (indicated by hooked arrows representing fields) and an edge-tangle (represented by red arcs representing cross-dependent edges).



analytic purposes as "multi-relations" — can either be modeled by *role-sorting*, which ascribes distinct roles to different participants, or *reification*, which converts a relation that would ordinarily be represented as a *connection* between two objects (an edge) and converts it to an object in its own right (taking here "object" to mean any compound data structure with its own fields to be graph-represented). The most semantically accurate representation is arguably to combine these two options. Consider the function of multi-part relations from the perspective of graph queries and traversals: suppose I know *John*, and wish to learn the name of his ex-wife (or analogously a traverser is on the John-node and wants to visit *her* node). The divorce-relation should take me between those two nodes; this expectation remains in effect even if the relation is multi-part, because a multi-relation should still serve as a fact authenticating a step between two related nodes on the basis of that relation being known as a fact.

In other words, a multi-relation should supply traversal steps no less than would a binary relation; the difference is that multi-relations enable *several* traversal paths, and we can use *roles* to select one direction or another. Getting to *Jane* from *John* traverses the *divorce* relation, but specifically the *spouse* role embedded in that relation. Invoking other roles (say, marriage-date) would cut across the multi-relation in an alternative direction. From the perspective of queries and traversals, accordingly, the multi-relation behaves functionally like a set of binary relations: divorce encompasses *ex-spouse*, *marriage date*, *divorce date*, and so forth. However, these implied binary relations are not independent graph elements; instead, the multi-relation enforces interdependencies among them. If (say) a correction were to update the *divorce date* value, this would necessarily alter the implicit relations identifying *John*'s and *Jane*'s divorce-dates, respectively.

Semantically, then, multi-relations could most accurately be described as interconnected sets of edges (or structures that logically play a role akin to edges) — i.e., from one multi-part relation there is a collection of different traversal-steps which can be taken via the data encoded in the relation as an authenticator (on the premise that knowledge-graph facts "legitimize" steps between nodes when one is seeking to obtain information constrained by known facts, e.g., to get information about an ex-spouse starting from the other former spouse). However, these implicit step-possibilities are, not "autonomous" as they would be with a disparate collection of edges instead of one multi-relation that logically implies those edges; changes to node-values related to one part of the relation may potentially propagate to other parts and participants in the relation. In this sense the multi-relation operates akin to an aggregate of "reactive values" in Functional Reactive Programming, where state-changes triggering updates in one value may (as an implementational requirement on the reactive engine) cascade to updates on the other values (*see* e.g. [44] or [27]).[15] For sake of discussion, call a collection of edges which are interdependent along these lines (in the sense that node-value changes in the neighborhood of one edge may propagate to others) an "edge tangle" (see Figure 5). A multi-relation is then in effect a data structure which merges the semantic and operational roles of *edge tangles* and of *relation-reifying* objects (in effect, the multi-relation object simultaneously reifies each edge in the edge-tangle).

This chapter has concentrated on procedure-calls more than on graph structures, but the two notions are interrelated, an idea which is especially apparent in the case of multi-relations. In graph modeling, a relation serves to guide or legitimate a traversal step; in some contexts, a relation may provide operational possibilities effectively akin to a function (e.g., a procedure): using a graph-edge to obtain the node of an ex-spouse from that of the other ex-spouse is analogous to calling a procedure which outputs the one ex-spouse when passed the other. Stepping from one node to another by following a constrained traversal strategy is, in short, not unlike calling a function which inputs the origin node and outputs the destination node.

Pursuing this analogy: a multi-relation acts like a function which can branch in different ways (e.g., by indexing branch options according to disparate roles) — in this sense multi-relations are effectively dual to single-return procedures. A procedure (in general) takes many inputs and returns one output, whereas a multi-relation (in the context of a single traversal or query) takes one starting node and presents multiple possible traversal directions.

[15] A *reactive expression* is a kind of union between a single value and a stored-expression which yields a value upon deferred evaluation, with the specific property that the value updates whenever the value of the stored expression would change due to modifications in its components (akin to a deferred expression which can be re-evaluated arbitrary many times, rather than only once, but canonically a side-effect-free expression that should only be re-evaluated when its components have changed). For example, a deferred expression in a GUI context might be the aspect-ratio of a GUI window, which gets updated when the window is resized. Reactive expressions thereby "tangle" their components (in the window case, those would be the window's width and height, and/or corner coordinates). Reactive expressions are bound to object-state (for some updatable object, e.g. a GUI components) with the idea that a reactive engine uses those expressions to maintain certain constraints (such as GUI layout constraints) even in response to external change (which the engine reacts to); declaring constraints in terms of reactive expressions obviates the need to implement procedural logic which enforces those constraints via callback handlers. Multi-relations could potentially be seen as akin to reactive expressions which provide satisfaction for constraints implicit in the coexistence of component relations in the single multi-relation; for instance, if John has divorced from Jane, then John's divorce-date is constrained to be the same as Jane's.

The fact that *roles* can be used to differentiate multiple destination-paths for a multi-relation suggests the dual idea that roles can similarly index different *input* sources for a procedure. Such an idea has not been pursued in full generality by mainstream programming languages — named (rather than positional) parameters and distinctions such as "**this**" objects from ordinary arguments (which we have modeled in terms of "channels") are rudimentary examples of parameter-role systems, but a more complete such system would support user-defined roles responding to the semantics of the target domain (rather than just the semantics of the programming language). However, parameter-roles can certainly be declared through a code-annotation system and therefore externally introduced and runtime-reflected using pointcut semantics as discussed earlier in this chapter. In this regard, parameter-roles can both add further expressiveness to external code-reflection and can leverage the idea of procedures and multi-relations being in some sense dual (and thereby potentially subject to similarly-structured metadata). We will discuss this aspect of parameter-roles further in Chapter 9, particularly in the context of Conceptual Spaces. We will argue that attempts to model procedural semantics via "blended" Conceptual Spaces, which is essential to Coecke *et al.*'s strategy for employing Conceptual Spaces as a grounding semantics for simulations of natural language, call strongly for the structural details afforded by parameter-roles.

4.2 Syntagmatic Graphs and Conceptual Spaces

This chapter has, to varying degrees of detail, examined pointcut expressions, pre-persistence bridge representations, and "grounded" serialization as three different aspects of data-and-code-modeling which can be subsumed within a common hypergraph-based description/query language. Each of these representational use-cases come into play in different coding concerns which need to be addressed when developing a robust code model that can cover the various dimensions of a full-fledged application (or code base which may be used by one or more applications), such as database persistence, serialization, and GUI integration. We suggested that these variegated coding concerns can be schematized by the visual "Saltire" device, connecting a central data type to associated types and/or representations which track how the type is restructured to accommodate different computing environments (e.g., databases, GUIs, and network-based data sharing where serialization/deserialization protocols come into effect). This picture illustrates how hypergraph-based codes and data representations can be exported to these various aspects of the overall application-development process.

In particular, pointcut expressions, pre-persistence, and serialization, according to the schemas we suggest here, all have a hypergraph basis, where query syntax and evaluation reduces to various forms of hypergraph site-locating and traversal. In other words, there is an underlying hypergraph semantics which provides the structures leveraged by the relevant hypergraph-query functionality. This semantics has not been formally set forth here, but a prototype of a semantic model adhering to the principles we have outlined is provided in the accompanying code, which can hopefully serve via demonstration-by-implementation as a useful proxy for a mathematical exposition of the underlying semantics.

Since the initial form of this semantics was also presented in the context of Coecke *et al.*'s work on hypergraph categories, it is reasonable then to consider how the hypergraph semantics underlying our pointcut/pre-persistence/serialization schema can be related to the Conceptual-Space-based semantics marshaled by Coecke *et al.*'s formulation. For example, we summarily discuss paths in Syntagmatic Graphs as tracing increases in information content, or "accretion of detail," which lead toward satisfaction of preconditions for procedures (or verbs, in the natural-language context) as prerequisite for procedure calls. The Syntagmatic constructions underlying such a path-model is analogous to the syntactic constructions analyzed by Coecke *et al.*: in effect, Syntagmatic paths are analogous to morphism-chains in Hypergraph categories.

In Coecke *et al.*, however, an essential point of their analysis is that there is a tight coupling between paths in the syntactic and semantic sense: morphism-chains implicitly "carry," in the sense of compelling and serving as a map-image for, structurally resonant paths in (some form of) Conceptual Space. The analogous semantic notion on our account would be "expansions" of Information Content, but this is a less structurally detailed construction of *semantic* paths than Coecke *et al.*'s model. As such, we can consider whether (and how) to adopt Conceptual Spaces as a source for a more rigorous "path semantics" to mirror our Syntagmatic Graph syntax, by comparison to how Coecke *et al.* use Conceptual Spaces to provide an image-domain (in a sense to mirror) morphism-chains in hypergraph categories. One difference between the two models is the nature of the hypergraph categories involved — Coecke *et al.* work with straightforward monoidal structures whereas hypergraph semantics in our context involve query-processing states for hypergraph query languages, so the hypergraph basis on our end has more structure (although for reasons just outlined the path-semantics side, at least so far, has *less* structure). We will consider these comparisons in greater detail in Chapter 9.

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] _____, "Conceptual Space Markup Language (CSML): Towards the Cognitive Semantic Web". *IEEE International Conference on Semantic Computing*, 2009, Proceedings, pages 51-68. https://ieeexplore.ieee.org/document/5298627

[3] Giuseppe Attardi, "The Embeddable Common Lisp" *ACM SIGPLAN Lisp Pointers*, Volume 8, Issue 1 (1995), pages 30-41. https://common-lisp.net/project/ecl/static/files/papers/ecl-1995-attardi.pdf

[4] Sikha Bagui, "Achievements and Weaknesses of Object-Oriented Databases" *Journal of Object Technology*, 2003. http://www.jot.fm/issues/issue_2003_07/column2.pdf

[5] Lucas Bechberger and Kai-Uwe Kühnberger, "Measuring Relations between Concepts in Conceptual Spaces".*SGAI International Conference on Artificial Intelligence*, 2017. https://arxiv.org/pdf/1707.02292.pdf

[6] Walter Cazzola, *et al.*, "Semantic Join Point Models: Motivations, Notions and Requirements". *Software Engineering Properties of Languages and Aspect Technologies*, 2006, Proceedings. https://hal.inria.fr/inria-00542782/document

[7] David Channin, *et al.*, "The caBIG Annotation and Image Markup Project". *Journal of Digit Imaging*, Volume 23 (2010), pages 217-225. https://pubmed.ncbi.nlm.nih.gov/19294468

[8] Bob Coecke, "The Mathematics of Text Structure". https://arxiv.org/pdf/1904.03478.pdf

[9] 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

[10] Ralph Debusmann, *et al.*, "A Relational Syntax-Semantics Interface Based on Dependency Grammar". *20th International Conference on Computational Linguistics*, Proceedings, 2004. https://www.aclweb.org/anthology/C04-1026.pdf

[11] Keith Devlin, "Situation Theory and Situation Semantics". *Handbook of the History of Logic*, Volume 7 (2006), pages 601-664. https://web.stanford.edu/~kdevlin/Papers/HHL_SituationTheory.pdf

[12] Juneki Hong and Jason Eisner, "Deriving Multi-Headed Planar Dependency Parses from Link Grammar Parses". https://www.cs.jhu.edu/~jason/papers/hong+eisner.tlt14.paper.pdf

[13] Jerry Fodor, "Language, Thought and Compositionality". *Mind and Language*, Volume 16, Issue 1 (2001) https://onlinelibrary.wiley.com/doi/abs/10.1111/1468-0017.00153

[14] Mirjam Fried and Jan-Ola Östman, "Construction Grammar: A thumbnail sketch". In Mirjam Fried and Jan-Ola Östman, Eds., *Construction Grammar in a Cross-Language Perspective* (John Benjamins, 2004), pages 11-86. https://www.researchgate.net/publication/280291354_Construction_Grammar_A_thumbnail_sketch

[15] Ángel J. Gallego and Román Orús, "Language Design as Information Renormalization". https://arxiv.org/abs/1708.01525

[16] Peter Gärdenfors, *et al.*, "Event Boards as Tools for Holistic AI". *6th International Workshop on Artificial Intelligence and Cognition*, 2019, pages 1-10. http://ceur-ws.org/Vol-2418/paper1.pdf

[17] Peter Gärdenfors and Frank Zenker, "Theory Change as Dimensional Change: Conceptual Spaces Applied to the Dynamics of Empirical Theories". *Synthese 190*, Volume 6 (2013), pages. 1039-1058. http://lup.lub.lu.se/record/1775234

[18] Jose Emilio Labra Gayo, *et al.*, "Validating and Describing Linked Data Portals using Shapes" https://arxiv.org/pdf/1701.08924.pdf

[19] Joseph Goguen, "An Introduction to Algebraic Semiotics, with Application to User Interface Design". In Nehaniv, C. (ed.), *Computation for Metaphors, Analogy, and Agents*, Springer, 1999, pages 242-9. https://cseweb.ucsd.edu/~goguen/pps/as.pdf

[20] Suelen M. de Paula and Ricardo R. Gudwin, "Evolving Conceptual Spaces for Symbol Grounding in Language Games". *Biologically Inspired Cognitive Architectures*, Volume 14 (2015), pages 73-85. https://www.sciencedirect.com/science/article/pii/S2212683X15000493

[21] Helmar Gust and Carla Umbach, "A Qualitative Similarity Framework for the Interpretation of Natural Language Similarity Expressions". In Lucas Bechberger, *et al.* (Eds.), *Concepts in Action - Representation, Learning, and Application*, Springer 2021. http://www.carla-umbach.de/publications/Gust&Umbach_QualitativeSimilarityFramework.pdf

[22] Thomas Hartmann, "Validation Framework for RDF-based Constraint Languages". Dissertation, Karlsruhe, 2016. https://publikationen.bibliothek.kit.edu/1000056458/3865145

[23] Radek Kočí, *et al.*, "Object Oriented Petri Nets - Modelling Techniques Case Study". *Second UKSIM European Symposium on Computer Modeling and Simulation*, 2008. https://ijssst.info/Vol-10/No-3/paper4.pdf

[24] Michael Köhler and Heiko Rölke, "Properties of Object Petri Nets". *International Conference on Application and Theory of Petri Nets*, 2004, pages 278-297. https://link.springer.com/chapter/10.1007/978-3-540-27793-4_16

[25] Malka Rappaport Hovav and Beth Levin, "The Syntax-Semantics Interface". In Shalom Lappin and Chris Fox (Eds.), *The Handbook of Contemporary Semantic Theory*, Wiley 2015, chapter 19, pages 593-624. https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781118882139.ch19

[26] Corentin Jabot, "Strongly-Typed Reflection on Attributes" http://open-std.org/JTC1/SC22/WG21/docs/papers/2019/p1887r0.pdf

[27] Wolfgang Jeltsch, *Categorical Semantics for Functional Reactive Programming with Temporal Recursion and Corecursion*. https://arxiv.org/pdf/1406.2062.pdf

[28] Karl Klose and Klaus Ostermann, "A Classification Framework for Pointcut Languages in Runtime Monitoring". *International Conference on Objects, Components, Models and Patterns*, 2009, Proceedings, pages 289-307. https://link.springer.com/chapter/10.1007/978-3-642-02571-6_17

[29] Daniel Korenblum, *et al.*, "Managing Biomedical Image Metadata for Search and Retrieval of Similar Images." *Journal of Digital Imaging*, Volume 24, Number 4 (2011), pages 739-748. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3138941

[30] Yusuke Kubota and Robert Levine, "The Syntax-Semantics Interface of 'Respective' Predication: A unified analysis in Hybrid Type-Logical Categorial Grammar". *Natural Language & Linguistic Theory*, Volume 34 (2016), pages 911-973. https://www.asc.ohio-state.edu/levine.1/publications/kl-resp.pdf

[31] Jonathan Lawry and Yongchuan Tang, "Uncertainty Modelling for Vague Concepts: A prototype theory approach". *Artificial Intelligence*, Volume 173 (2009), pages 1539-1558. https://www.sciencedirect.com/science/article/pii/S0004370209000903

[32] Martha Lewis and Jonathan Lawry, "Hierarchical Conceptual Spaces for Concept Combination". *Artificial Intelligence*, Volume 237 (2016), pages 204-227. https://www.sciencedirect.com/science/article/pii/S0004370216300492

[33] Yaoyong Li and Hamish Cunningham, "Geometric and Quantum Methods for Information Retrieval". In *Association for Computing Machinery Special Interest Group on Information Retrieval*, Volume 42, Number 2, 2008, pages 22-32. https://dl.acm.org/doi/10.1145/1480506.1480510

[34] Luís Lopes, *et al.*, "A Virtual Machine for a Process Calculus". *International Conference on Principles and Practice of Declarative Programming*, 1999, pages 244-260. http://di.fc.ul.pt/~vv/papers/lopes.silva.vasconcelos_virtual-machine-tyco.pdf

[35] Fei Liu, *et al.*, "Coloured Petri Nets for Multilevel, Multiscale and Multidimensional Modelling of Biological Systems". *Briefings in Bioinformatics*, Volume 20, Issue 3, (2019), pages 877-886. https://academic.oup.com/bib/article/20/3/877/4590142

[36] Ilya Makarov, *et al.*, "Quantum Logic and Natural Language Processing". http://ceur-ws.org/Vol-1886/paper_16.pdf

[37] Konstantinos Meichanetzidis, *et al.*, "Quantum Natural Language Processing on Near-Term Quantum Computers" https://arxiv.org/abs/2005.04147

[38] Laura A. Michaelis and Knud Lambrech, "Toward a Construction-Based Theory of Language Function: The Case of Nominal Extraposition" *Language*, Volume 72, Number 2 (1996), pages 215-247. https://spot.colorado.edu/~michaeli/documents/Michaelis_Lambrecht_NE_LG.pdf

[39] Friederike Moltmann, *Parts and Wholes in Semantics*. Oxford Univ. Press, 1997. https://www.researchgate.net/publication/244487318

[40] Pattanasak Mongkolwat, *et al.*, "The National Cancer Informatics Program (NCIP) Annotation and Image Markup (AIM) Foundation Model" *Journal of Digit Imaging*, Volume 27 (2014), pages 692-701. https://europepmc.org/backend/ptpmcrender.fcgi?accid=PMC4391072&blobtype=pdf

[41] Lee J O'Riordan, *et al.*, "A Hybrid Classical-Quantum Workflow for Natural Language Processing". *Machine Learning: Science and Technology*, Volume 2 (2021). https://iopscience.iop.org/article/10.1088/2632-2153/abbd2e/pdf

[42] Paolo Pareti, *et al.*, "SHACL Constraints with Inference Rules Paolo Pareti". In C. Ghidini, *et al.* (Eds.), *International Semantic Web Conference 2019*, pages 539-557. https://www.southampton.ac.uk/~gk1e17/shacl-inference.pdf

[43] Walter B. Pedriali, "The Routes of Sense: Thought, semantic inderdeterminacy and compositionality". Dissertation, St. Andrews, 2011https://research-repository.st-andrews.ac.uk/handle/10023/3142

[44] Stefan Ramson and Robert Hirschfeld, "Active Expressions: Basic Building Blocks for Reactive Programming". https://arxiv.org/abs/1703.10859

[45] Martin Raubal, "Formalizing Conceptual Spaces". http://www.raubal.ethz.ch/Courses/288MR_Spring08_Papers/Raubal_FormalizingConceptualSpaces_FOIS04.pdf

[46] Daniel L. Rubin and Kaustubh Supekar, "Annotation and Image Markup: Accessing and Interoperating with the Semantic Content in Medical Imaging". *Intelligent Systems*, Volume 24, Number 1 (2009), pages 57-65. https://web.stanford.edu/group/rubinlab/pubs/Rubin-IEEEIntelSys-2009.pdf

[47] Vahid Salari, *et al.*, "Parametrized Quantum Circuits of Synonymous Sentences in Quantum Natural Language Processing". https://assets.researchsquare.com/files/rs-220713/v1_stamped.pdf

[48] Christian Schafmeister and Alex Wood, "Clasp Common Lisp Implementation and Optimization". *11th European Lisp Symposium*, 2018, pages 59-64. https://dl.acm.org/doi/10.5555/3323215.3323223

[49] Roger Schaer, *et al.*, "Web-Based Tools for Exploring the Potential of Quantitative Imaging Biomarkers in Radiology: Intensity and Texture Analysis on the ePAD Platform". In Adrien Depeursinge, *et al.* (Eds), *Biomedical Texture Analysis: Fundamentals, Tools, and Challenes*, Academic Press, 2017. http://bigwww.epfl.ch/publications/schaer1701.pdf

[50] Daniel D. Sleator and Davy Tamperley, "Parsing English with a Link Grammar". https://www.link.cs.cmu.edu/link/ftp-site/link-grammar/LG-IWPT93.pdf

[51] Gerold Schneider, "A Linguistic Comparison of Constituency, Dependency and Link Grammar". Zurich University, Diploma, 2008. https://files.ifi.uzh.ch/cl/gschneid/papers/FINALSgeroldschneider-latl.pdf

[52] Alexandru Telea and Jarke J. van Wijk, "VISSION: An Object Oriented Dataflow System for Simulation and Visualization". *Data Visualization*, Proceedings, 1999, pages 225-234. https://pure.rug.nl/ws/portalfiles/portal/3178139/1999ProcVisSymTelea.pdf

[53] Roman Urban and Megdalena Grzelinska, "A Potential Theory Approach to an Algorithm of Conceptual Space Partitioning". In *Cognitive Studies*, Volume 1 (2017). https://ispan.waw.pl/journals/index.php/cs-ec/article/download/cs.1310/3065

[54] Juan Uriagereka, *Syntactic Anchors: On Semantic Structuring*. Cambridge University Press, 2008. https://assets.cambridge.org/97805218/65326/frontmatter/9780521865326_frontmatter.pdf

[55] Remi van Trijp, "Cognitive vs. Generative Construction Grammar: The case of coercion and argument structure". *Cognitive Linguistics*, Volime 26, Issue 4 (2015), pages 613-632. https://www.degruyter.com/document/doi/10.1515/cog-2014-0074/html

[56] Frank Zenker, "From Features via Frames to Spaces: Modeling Scientific Conceptual Change Without Incommensurability or Aprioricity". In Thomas Gamerschlag, *et al.* (Eds.), *Frames and Concept Types: Applications in Language and Philosophy*, Springer, 2014, pages 69-89. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.901.7852&rep=rep1&type=pdf