Chapter 21 – Virtual Machines and Hypergraph Data/Code Models: Graph-Theoretic Representations of Lambda-Style Calculi

Nathaniel Christen and Amy Neustein

Abstract

This chapter will consider Virtual Machines (VMs) in the sense of computer programs that act on intermediate representations of computer code, serving as targets for source-code compilers that are an alternative to physically-realizable "machine language." Unlike natural language, programming languages are artifacts which have to be explicitly implemented in a computing environment: apart from machine language itself, no computer can parse and properly respond to expressions in computer languages directly. High-level programming languages therefore rely on their own software components to parse, compile, and execute source code — either in lieu of compiling high-level sources all the way down to machine code or by supplementing machine-language capabilities with a runtime environment and a collection of built-in functions for low-level requirements, such as interfacing with the operating system. Many languages depend on VMs either to simulate code-execution during the compile-time phase or to simulate actual machine language during runtime. Because VMs are "virtual" and not restricted to physically realizable operations, they provide a flexible and extensible compilation-target that can promote expressive and adaptable high-level programming languages. We will examine hypergraph representations of source code, and on how hypergraph constructions can serve as a point of orientation for designing a VM's instruction set. We will sketch, semi-formally, a mathematical overview suggesting how systems similar to the lambda calculus can be represented in the context of hypergraphs, yielding graph-theoretic models of such calculi that might be adopted as alternatives to their original formulations in symbolic logic. We outline how this approach to Virtual Machine design and code-modeling may benefit software engineering in contexts such as data sharing protocols and GUI front-end programming.

1 Introduction

This chapter will examine interconnections between three of the focal subjects for Section 5 of this book: Virtual Machines, hypergraph data models and meta-models, and the common structures (amongst many dissimilarities, that were reviewed in Chapter 20) between human and computer languages. We will continue last chapter's discussion of data-model and programming language "semantics." We argued in the previous introduction that digital artifacts represent real-world objects only indirectly (we can contrast this with word-object associations in natural language). Applications "signify" empirical facts because they can be engineered to maintain a consistent (even if occasionally fallible) correlation between application-state and particular real-world propositional situations of interest (recall the case of apps showing a frequently-updated World Cup score).

To adopt terms familiar in the Philosophy of Science, we would argue that signification in this environment is (at most) an *emergent property* of an overall computational system. Whereas a photograph of a key moment in the World Cup final — assuming it is authentic — represents an event with some immediacy, a computer program that someone might consult to track the game's score can only be deemed an indicator for the relevant propositional content (something like, France and Argentina are now tied 3-3) insofar as the program fulfills its design requirements. Semantics, in this realm, is *implemented*. We have to assess the inner workings of a digital artifact to confirm that the phenomena we as users experience as "signifiers" (e.g., two numbers next to France and Argentina flags/logos, respectively) actually play that role.

Because signification is emergent in this sense, the semantics of application-state can only be rigorously defined on a holistic scale: it is a product of many parts within the application acting and interacting correctly. Consider the symbolizing device of printing a number next to the two country's flags: as users, we perceive that each displayed number correlates with the respective country's goal total, and therefore stands toward that total in something like a signifier-signified relationship. But such correlation only *happens* because many operations are completed behind the scenes: repeatedly (we may assume) connecting to an internet service (itself presumably monitored and updated by someone who witnesses the game directly) to receive a data-package including the current score, decoding/deserializing that data into live memory, mapping the fields in the resulting live objects/values to specific screen areas, and mapping the numeric scores into printable numerals whose glyphs are arranged to be viewed in those boxes on the screen. All of these operations involve data structures modeled, shared, and manipulated, and chains of procedures responsible for one step or another in the overall data-management process.

To understand the holistic semantics of computer programs as emergent systems, then, we need to model the "local" or behind-the-scenes semantics of individual data structures and the procedures which manipulate them. This is a semantics which emerges as data-type instances are initialized and then passed from one procedure to another (potentially modified along the way), and, in some cases, mapped to a visual/iconic form. It is a semantics of holistic systems which take inputs in the form of user actions and/or received data (e.g., from an internet connection) and output (mostly) visual artifacts on-screen. It is a *procedural* semantics because chains of operations are typically needed to progress from the inputs to the outputs.

To model this semantics, then, if we seek to analyze computer software by analogy to studying human language, we need to represent procedure calls: how tokens in source code map to specific functions that can be executed as subroutines, interrupting the program-flow

at the call site and then returning, and how procedures modify and manipulate data-structures according to constraints of data types and design patterns (for example, a type can never hold a value incompatible with its potential instance-set: an unsigned integer, say, can never be negative). Programming-language semantics, in short, is built up in layers, with inter-procedure conventions establishing the working substructure through which (via GUI design and responsiveness to user actions) applications take on states, and present information to users, which have emergent-significatory effects.

Formal models of data structures and procedure-calls are therefore intrinsic to the semantics of artificial computer languages, with no obvious analogues in "natural" language. In later chapters we will consider to what degree extralinguistic "cognitive processing" plays a somewhat analogous role in human language-understanding, but for the moment we can register this as a foundational contrast between these two genre of languages. Whereas the building blocks of human dialects are parts of speech, morphosyntactic markings, anaphoric/deictic resolution, and word-senses within a lexical continuum, the analogous building blocks in source- and compiled computer code are procedures, their parameters, symbol-tokens iconifying specific values (tracked and potentially modified) over time, numeric encodings of data structures (not only quantitative ones, but also text, enumerations, visual graphics, and multi-media), and text or character-string representations of values likewise (such as numerals written into source code to represent numbers).

Virtual Machines (VMs) are a useful theoretical tool for studying these "semantic" building-blocks. To be sure, VMs have many applications, and analyses of computer languages inspired by natural linguistics are probably not the most important of these. Still, a "linguistic" perspective is a useful intuitive guideline for considering how VMs relate to computer code and programming-language runtimes.

There are several different use-cases for VMs. We might recognize the following:

Implementing procedures Virtual Machines in this sense play the role of execution-engines mimicking the behavior of physical processors (see Chapter 20, footnote ?LLVM). Here VMs actually run procedures and produce values and/or side-effects as an alternative to compiling code to machine language.

Simulating procedures Instead of actually executing procedures, VMs may simulate how procedures *would* behave if they were called with specific values, so as to obtain information for compilation, diagnostics, testing, Quality Assurance, and so forth.

Encapsulating calls to specific kinds of procedures As discussed in Chapter 20, applications and code-bases often "expose" certain functionality to be called by external/third-party components. A typical case is when relatively complex steps must be taken to instantiate environments where certain kinds of functionality are available. This chapter and the next will consider examples in the context of Cyber-Physical Systems and Computer Vision. In the context of code-libraries for interfacing with Cyber-Physical devices, or image-processing algorithms, respectively, such libraries are often designed with the goal of "hiding" complications related to image/data format and mathematical models. These libraries would be engineered to encapsulate relevant functions so that they may be invoked by external code, without complex domain-specific details being directly addressed by the calling code.

Programmers could therefore invoke operations for (say) Computer Vision or Cyber-Physical networks without needing detailed understanding of these respective domains. Virtual Machines may add an extra layer of indirection by including encapsulated domain-specific procedures alongside lower-level instructions as part of their "operation set." In this way VM can play a role analogous to (or permitting the implementation) of scripting interfaces to domain-specific modules (consider, say, a Python interface to the predominant Computer Vision libary, OpenCV: many programmers find it easier to invoke OpenCV routines in Python than to call these functions directly via C++).

Synthesizing or interoperating between divergent programming languages and computing environments Because — as touched on in Chapter 20 — VMs are more flexible than machine language and physical CPUs, they can provide a neutral platform that is agnostic to language-implementation details. For example, the philosophical differences between divergent programming styles (functional vs. Object-Oriented, say) are reinforced by what is optimized by compilers and runtimes. Languages encourage code-writers to adopt certain design patterns, and in turn they are optimized so that code which adheres to such patterns runs more quickly, or consumes less memory (and other computing resources, such as CPU time), or is subject to more stringent evaluation (compile-time error detection, for example), or all of the above. Because different languages optimize in different ways, it is hard to find a neutral medium where code written in two different source-dialects might be compiled to a single byte-code, say, or where data structures created in one language can be utilized by procedures in a different language. Virtual Machines, compared with CPUs, are typically slower but more flexible: because they do not seek the most optimized implementation, they may serve as a hybrid environment through which multiple languages may interoperate.

This chapter and the next will discuss some of these potential features/applications of VMs in a little more detail. The plan is first to summarize a specific model for VMs based on procedure-call semantics; subsequent discussion will consider the role of VMs in larger ecosystems, such as those related to image-processing.

2 Virtual Machines and Hypergraph Code-Models

As a preliminary to analyses later in this section, we will make a few comments which might situate this discussion in the context of topics covered elsewhere in this book. Specifically, we intend to motivate the discussion by appeal to technologies related to industrial computing, Cyber-Physical Systems (CPS), and Computer Vision.[1]

One potential use-case for Virtual Machines is encapsulating cyber-physical networks: we assume that CPS devices are sensors and/or actuators managed remotely by more-conventional computers. In general, sensors measure physical quantities (air quality in a room, say) whereas actuators have tangible physical effects (optimizing air quality by adjusting vents, say). Neither kind of device typically has extensive computational capabilities; instead, data is routed from the devices to a central location which use software to process sensor data

[1] Throughout this book-section we speak of "Virtual Machines" specifically in the sense of digital artifacts that emulate Central Processing Units or other physically-realized computing engines. Many discussions particular in the context of industrial technology refer more broadly to Virtual Machines that simulate engineered physical artifacts — as "digital twins" [8], [3], [67], [4], [44], [52], [55], [33] — or to virtual operating systems. Discussions about VMs in the former sense, however, can carry over to this broader usage because VM execution environments are often intrinsic parts of digital twins and/or virtual operating systems.

and convey instructions to actuators. These networks might, in turn, cover multiple intermediate points (consider CPS data warehoused in the cloud) but, for exposition, we can focus on the essential end-points — on one hand the devices themselves and on the other software applications through which humans monitor and, if desired, manipulate CPS technology. Even if the actual devices do not perform calculations (they may not have the means to execute computer code), sensors and actuators *are* capable of converting some physical quantity to digital signals which computers proper can interpret. Programmers developing CPS applications will rarely interact with devices independently; instead, the "physical" links with CPS networks[2] will typically be handled by low-level driver programs, which could expose capabilities to access sensor readings through, for instance, C-language functions.

Developers might employ Virtual Machines in the CPS arena to streamline this process — instead of programmers needing to write code which interfaces with C drivers directly, the low-level function calls can be managed via virtual machines that interoperate with higher-level applications in a language-agnostic manner. Recent research in data semantics and virtual simulations of CPS networks and devices can benefit the engineering of relatively general-purpose middleware tools that route data and instructions back and forth between application-style environments and CPS hardware [65], [15], [61], [58], [64]. An analogy might be made to database systems: although there are many different data persistence engines, a handful of specific architectures (relational/SQL plus the predominant NoSQL frameworks) have emerged as templates that cover the interactions of applications with almost all databases they are likely to encounter. Interop protocols can therefore target a core group of database models, yielding results applicable to a large range of actual technologies. Analogously, consolidating the extensive variety of CPS sensors/actuators into a core set of data profiles would support generic middleware components that work against many specific CPS networks.

In this sense CPS VMs can be analogous to database query languages, in their design and rationales: just as query formats give engineers the option of communicating with a database from a diversity of application-environments, the protocols for accessing CPS data could similarly be encapsulated in VM operations. Moreover, query engines (as noted in Chapter 20) can ideally function either as standalone languages of their own (processing query-code outside an application context, e.g., on a command-line for debugging and maintenance) or via query-factories programmable from general-purpose languages. The situation is comparable for CPS, and one could pursue the same duality in how CPS network capabilities are exposed (either through host languages or special extra-application code).

If a VM is indeed devoted to specific Cyber-Physical networks (or groups thereof) then a logical first step when implementing the VM would be to articulate the collection of procedures (through driver libraries, say) to be encapsulated. More likely, a more-generic VM would be deployed in different contexts, each of which (by targeting a specific group of CPS devices) would work against particular groupings of driver code, on a case-by-case basis. The overall VM would thereby introduce capabilities to interface with some collection of kernel functions exposed by driver libraries, with the actual integration to those procedures finalized during deployment. In any case, though, the general pattern is that any particular deployment of a VM would be "seeded" with some preliminary set of functions (i.e., built-in VM operations, some perhaps added on in deployment).

[2]Which of course can be physical only in an ethereal sense, e.g., passing data via wireless services

Someone might reasonably point out that interoperability between heterogeneous components — whether the differences are just in programming environments (for disparate pieces of software) or more substantial, as with connections between applications and Internet of Things endpoints — typically falls under the rubric of data-sharing protocols and APIs, rather than VMs proper. That is, CPS driver libraries could certainly expose access to devices through an API, but this would only indirectly be an issue for VM (via, say, programmers using a scripting language to facilitate API calls). While this is true in principle, we briefly argued in the previous chapter that VMs can serve as a logical extension to APIs: there are rationales for exposing functionality via connection-points handled through VMs, either to ensure interop protocols' adaptability or to present an interface that can be hooked in multiple ways — via scripts, query-like languages, application code building up requests incrementally, etc. This potential VM use-case fits well with the theoretical perspective on VMs developed in the current chapter. Here, we will consider VMs as systems organized around a collection of kernel functions, which in turn form the basis of procedures implemented through the VM that call such functions in sequence. While some kernel operations may be generic in the sense that they simply provide expected features of coding environments — declaring variables' symbolic names, types, and scopes, registering procedure inputs, binding outputs to symbols or input-positions on enclosing procedure calls, and so forth — there is no reason why kernel operations could not encapsulate access to domain-specific capabilities, such that VMs mimic features of APIs.

In this scenario, VMs furnish functionality imitating "kernel" driver or operating-system calls, initializing input arguments with values passed from applications and sending results back. The VM accordingly bridges application-level and low-level kernel functions, which involves not only encapsulating access to low-level functionality but also negotiating the discrepancy between relatively high-level and low-level programming environments. Most application programming languages, for example, recognize constructs (e.g., Object-Orientation, or functional programming via lambda closures/call-continuations, plus exceptions, mutable references, and so on) outside the scope of low-level code (even C). Effective "bridge" protocols allow applications to access CPS data (as one category of low-level resources) within familiar higher-level paradigms. For example, programmers already working in an Object-Oriented environment might reasonably find it intuitive to apply object models to CPS elements, such as individual devices. Driver code most likely would not recognize Object-Oriented calling conventions directly, so a VM could be pressed into service to translate application-level descriptions of procedure calls and their input parameters into simpler binary packages which drivers can handle.

In this sense, a VM fitting these requirements would simultaneously model "high level" calling conventions and translate procedure-call requests between higher and lower levels. We make this point in the context of CPS, but the idea would hold for many cases wherein VMs encapsulate access to some integral set of low-level functions (at least low-level relative to applications that would benefit from the requisite functionality). Consider image-analysis: procedures exposed via Computer Vision libraries are not directly comparable to low-level driver code — they may themselves be implemented in high-level languages like C++) — nevertheless, libraries such as OpenCV or ITK tend to demand a rather complex scaffolding set up to enable analytic procedures being called. We could imagine circumstances where wrapping complex image-processing pipelines in a simpler API would be convenient for code libraries managing image series. For

instance, a database exposing image-processing operations through query expressions would ideally support query-evaluation covering Computer Vision capabilities (going beyond obvious information one may want to query from an image, such as dimensions and color depth).

Indeed, database queries, image-processing, and CPS network administration each represent plausible use-cases wherein VMs can serve as adapters between application-style coding environments and procedure-collections that are too specialized or low-level to fit comfortably in application-development norms. These cases might overlap, of course; databases can warehouse CPS data such that queries monitoring real-time device state would logically coexist with queries against database content (tracking device info), or image databases can support queries dependent on Computer Vision.

The database-query perspective points to an interesting heuristic analogy, insofar as packaging procedure-calls from an application environment to a lower-level context resembles the task of packaging application-level datatypes and updates into records and fields natively recognized by a persistence engine. Similar to how live-memory data structures (objects, say, in Object-Oriented environments where the relevant data is interpreted in light of objects' polymorphic type, which could be subclasses of their declared type) are destructured for insertion into database sites, high-level calling-conventions (again, Object-Orientation provides useful examples insofar as **this**/**self** values are represented apart from other input parameters) need to be restructured into the (generally simpler) forms suitable for low-level functions (e.g., moving **this**/**self** to be an ordinary parameter, which may require truncating to base-class binary layouts, and some level of type-erasure).

Continuing this analogy, flexible database architectures bridge application-level data types with persistent data structures/records so that application code can work with back-end values according to the norms of application-context programming; equivalently, VMs can wrap low-level functions such that they fit the profile of high-level procedures called according to high-level conventions (with objects, exceptions, and so forth). A flexible VM would play this bridge role in multiple contexts, perhaps allowing native functions to be registered as kernel operations and striving to be usable from multiple host languages: that is, from one VM we can envision encapsulating access to a variety of procedure-collections (examples we've cited here include functions exposed vis-à-vis database queries, image-processing, and CPS networks) and routing descriptions of procedure-calls from a variety of programming languages (which might have object-oriented or functional characteristics, or some combination). To the degree that such generality is desired, VM should anticipate integration with diverse application-level languages preferring different calling-conventions and procedural contracts.

One maxim for VM design, then, at least in this sort of context, is to prioritize capabilities to model and carry out procedure-calls described through diverse calling-conventions, rather than narrowing in on specific calling-conventions which derive from a preferred programming model (functional or Object-Oriented, say). Object-Oriented conventions such as method overrides/polymorphism and exceptions/exception-handling might be natively expressed via the VM, but likewise functional idioms such as lazy evaluation and overloading based on type-state. To clarify the last example: procedures can potentially be given different implementations by virtue of values' type-state at the moment of call (essentially a more granular classification than type-attribution itself), often mixing type-state with value-destructuring. A canonical example would be procedures operating on list-style collections, which in one overload would accept only *non-empty* collections where the last (or first) element is passed separately, alongside a structure representing all other elements (this convention is ubiquitous in recursive algorithms, since the "tail" can then be passed to the same procedure recursively, becoming destructured by the calling mechanism into its own head-plus-tail pair; of course, procedures implemented via this strategy also need an overload taking an empty list, which serves as a halting-point).

In short, an ambitious VM can ideally work with a broad set of calling styles embraced by diverse programming styles, e.g. object-methods, exceptions, lazy evaluation, typestates, and parameter-destructuring. To this list we might add dependent types and functional-reactive idioms (e.g., so-called "signal/slot" conventions, reviewed in greater detail in Chapter 24).

Since not all VMs actually aspire to multi-language support to the open-ended degree implied here, theories informing VM implementation do not necessarily analyze data models or design patterns targeted specifically at language "agnosticity" (so to speak); a more common scenario is that theoretical perspectives emanate from coding paradigms which give rise to distinct flavors of programming languages, potentially at some level proselytizing for favored paradigms rather than aiming for broad generality (in the sense that VMs *for functional languages*, say, reflect a general assumption that functional methodology is in the general case a better coding style than alternatives). By contrast, we are interested here in describing theoretical VM models that remain nonjudgmental as to which conventions are better in which context (or to take the view that multiple coding styles each have their own use-case so should be supported as such). A rigorous VM "model" should have multiple dimensions (addressing types and data-encoding, for example), but of course an essential concern is modeling procedure-calls and calling-conventions, so we will pursue this topic as a starting-point for a (relatively informal) system to encapsulate VM details in a schematic fashion.

2.1 Applicative Structures and Mathematical Foundations

Theoretical (and applied) computer science often approaches procedures from the viewpoint of mathematical functions, essentially mapping that transform inputs to outputs. Codifying the principles of "functionhood" in general forms the central project of analyses that bridge math and computers (e.g., lambda calculus). In terms of mathematical "foundations," these various formulations (including lambda calculus and its derivatives and, for instance, Combinatory Logic) entail strategies for clarifying what are sometimes called "applicative structures," or generic patterns involving the *application* of a function/procedure to one or more arguments/parameters [28, e.g., Chapter 14], [57, Chapter 18] [17], [9], [25], [42], [22], [29], [12], [39], [23], [63]. Since this is such a basic phenomenon in the mathematical realm, it is understandable that some researchers in "philosophy" of mathematics and its foundations would focus on applicative structures (perhaps indirectly via, say, lambda calculus) — even though mathematical expressions are written down according to a wide variety of conventions (consider formulae for integration, for ratios, for polynomials, and so forth) we can imaging a system which translates mathematical terms to a more systematic logical representation (which would presumably resemble something like Lisp code).

The concerns evoked by applicative structures are not only orthographic, however, because there is also a *semantic* dimension in the sense of spaces of functional values qua semantic entities. The problem of semantically characterizing *a* function or procedure (e.g., as a calculational process, or a — maybe time/context-dependent — input-to-output mapping, or the like) is one matter, but whatever our foundational semantic theory in this sense it is natural to extend it via functional composition (analogous to how linguistic semantics involves the semantics of nouns and verbs, but more thoroughly also the compositional principles of verbs together with nouns yielding sentences/propositions). The set of possible applicative structures "generated" by some collection of functions (or function-symbols) is analogous to the set of discrete functional values that can be defined by the combination of multiple functions wherein the results of one function applied to its arguments becomes in turn (one of the) parameters of a different (or, recursively, the same) function.

Systematically, an "applicative system" would be a set of function-symbols alongside "variable" symbols with a rule that applicative *structures* include expressions of the form $fx_1x_2...$ where xs are variables, and that for any applicative structure the modification formed by replacing a variable-symbol with another applicative structure is also an applicative structure. Defined in terms of the closure of such substitution operations, applicative systems can be seen to follow a generative pattern very similar to labeled trees — each node is either a symbol-node or a branch with its set of child-nodes — with the added detail that labels are partitioned into two sets (function and variable symbols, respectively) such that left-most child nodes always have function-labels and other non-branch nodes always have variable labels.

The analytic purpose of applicative structures can potentially be served by systems employed in other branches of mathematics — e.g., "term algebras," in formal logic, or n-ary trees in computer science, or an extension of the "free magma" groupoid concept to (sets of) n-ary (not just binary) operations. For instance, applicative structures might be seen as term algebras if we consider only function-terms in the context of first-order logic, ignoring predicates, constants, or relations (constants can be treated as nullary functions, and predicates/relations modeled via functions whose codomain is the set of two boolean constants *true* and *false*). Applicative structures may also be defined in terms of freely generated strings in a language with balanced parentheses — essentially a "Dyck" language (after Walther von Dyck) intersected with a suitably restricted function/variable language [43], [35], [37], [32].

In effect, applicative structures can be defined by restricting the forms generated by first-order logic, or by regular languages according to the Chomsky-Schützenberger representation theorem (in formal linguistics) [66], [20], [31], [40]. Different such formulations could be preferred depending on which seems like an intuitive basis in the specific context being analyzed. However, although any language roughly equivalent to n-ary trees may be intuitively simple, it is worth defining the precise structures one is working with axiomatically to ensure that there is a strict isomorphism between representations in different contexts, as opposed to spaces which overlap but do not exactly align (for example, balanced-parentheses strings allow for nullary function applications which are not necessarily substitutable with constant values; it is unclear how to map that specific possibility to n-ary trees). In any case, our strategy here is to define applicative structures recursively in a manner that works whether we approach them syntactically or semantically, and to analyze their properties mostly through a representation based on matrices, which in turn could potentially be used to verify isomorphisms between these structures and alternative encodings (here we carry out such an analysis in a graph-theoretic context, though we propose similar methods might apply to, e.g., term algebras).

We'll make a couple of technical points about applicative structures here, not so much because the mathematics is particularly sophisticated or consequential but so as to establish a baseline of comparison for the graph-theoretic encoding of (generalizations of) applicative structures we will discuss subsequently.

Definition. *Applicative Systems*: for any fixed set of *function symbols* and *variable symbols* an *applicative structure* is any element of a set which is the closure of the set of primitive expressions (consisting of a function symbol followed by a number of variable-symbols which matches its arity, assuming we assign a specific arity to each function, or else to any number of variable-symbols) under substitution operations wherein an applicative structure is inserted in an enclosing structure taking the place of a variable. Applicative Systems are then sets of function and variable symbols (and possibly declarations of function-arity) together the full set of possible applicative structures generated on their basis. For generality, we can allow functions `f` to have dynamic range arity (multiple integers n such that `f` followed by n variables is a valid expression); in this sense systems which do not recognize arity limitations at all would implicitly allow arbitrary range arity for all function symbols.[3] Note that the generative rules allow for zero-arity functions, whose semantics would be procedures that yield results even without inputs (nested structures can be wholly comprised of one single function symbol).

Definition. A given symbol may appear multiple times in an applicative structure; the above definitions were formulated with the idea that "the same" symbol in different positions is, according to the generative rules, a different symbol, but for clarity we can say that one symbol can have multiple *tokens* in a given structure. Each symbol-token has a *nesting level* such that the nesting level of a function-symbol matches that of variable-symbols following it (that are not themselves part of a further nested structure) and replacing a variable with a nested structure forces the function symbol at the left of the latter structure to have a nesting level on greater than the replaced variable. By the construction/generation process, there will always be exactly one function-symbol with least nesting level, which we can stipulate to be zero. Symbol-tokens also have *positional indices* defined such that function symbols are assigned index zero and variable-symbols following them (incrementing across but skipping over nested structures) are assigned successively greater index numbers. Tokens are uniquely identified by a positional-index list whose length is determined by (viz., one greater than) the token's nesting-level, notating the tokens own positional-index and also those that would be assigned to its parent nodes (treating the structure as a tree) were they tokens rather than nested structures. Positional-index lists induce an ordering on all tokens in an applicative structure (comparing the first number in two respective lists, then the second as a tie-breaker, and so forth; akin to ordering leaf nodes on a tree where left-ward and higher nodes are prior to those below and/or to their right). Applicative structures are *non-recursive* if each function-symbol has only one token. It is helpful to further define non-*root*-recursive structures as those where the function-symbol whose token has zero nesting level appears only once.

Definition. Applicative structures can be grouped into equivalence classes wherein any structure in the class would map onto a peer structure under a permutation of the applicative system's variable symbol-set. Since we can give an ordering to the variable-set, assume each variable is assigned a number code, which in turn allows us to define a *canonical presentation* of an applicative structure by permuting its variable-symbols so that tokens

[3] A *partial* applicative structure would be one that belongs to an applicative system no subject to arity restrictions but which is excluded when arities are recognized, due to one or more function symbols lacking a sufficient number of following symbols or nested structures; partial applicative structures in this sense can be semantically interpreted as designating "meta-functions" which, when the variables are replaced by fixed values, reduce to actual functions whose parameters are the missing symbols implied by insufficient arity thresholds. Of course, this discussion assumes we also have a notion of "fixed values" being assigned to functions as parameters.

which are prior (in the ordering based on positional indices just described) are assigned symbols with lesser codes, choosing each new symbol to be the one with lowest code value not yet used. Each applicative-structure equivalence-class is thereby represented by one structure with such canonical presentation, and we can restrict attention to these structures in particular. Note that the generative rules for applicative structures have to be separated into two groups, because (in the general case) we have to avoid unrelated symbols with the same "label" colliding according to the generative step whereby a nested structure is substituted for a variable. For a nested structure with its own collection of variables, the structure may need to be rewritten (cf. lambda-calculus alpha-conversion) as an equivalent structure (but with symbol-permutation) — not necessarily (indeed not usually) one in canonical-presentation — so that each nested symbol is different from all symbols in the enclosing structure (unless the symbol-representation is explicit, i.e., the point is to model a function-application where the same input value is copied in multiple places).

Representing function-arguments via "variable" symbols (which are understood to be substituted with values from some domain in the context of *calling* functions) is consistent with lambda calculus, and with the idea that applicative structures describe valid strings in a language codifying notions of function-application. However, it is possible to develop alternative representations of the same structures, such as De Bruijn indices (due to Nicolas de Bruijn) which in effect model argument-places by numeric indices rather than symbols [5].

We propose to use a similar convention based on the idea of forming symbols which have numeric values but also "colors." Specifically, consider, first, characterizations of the set of functions which may be identified on the basis of one function-symbol — say, `f`, which assume has a fixed arity a, say, three. Then `f` itself could be notated `f`123 (using the color red to signify function-symbols and blue numbers for variable-symbols). Given `f` we implicitly also have a family of related functions which are identical to `f` but operate on different domains, taking argument-lists longer than `f`'s arity but ignoring the extra values: we propose denoting these via `f`1234, `f`12345, etc. using *gray*-color numbers for discarded extra elements. Conversely, we can also form functions with arity *less* than `f`'s by repeating numbers for the arguments; e.g., `f`121 or `f`122 where note that numbers occurring multiple times are shaded in light blue. We'll refer to this as "red-blue-gray" notation — schematically, equivalent to notation with symbols (each colored number, i.e., color/number pair, being a distinct symbol) except that the colors allow grouping into ordered sets which may be subject to further constraints. Likewise, the function-symbols themselves can be replaced by numbers encoding any ordering of the list of available function-symbols which form the core of our applicative system.

Suppose we start with some function-symbol list — for instance, as earlier in this chapter we alluded to collections of kernel functions which "seed" a Virtual Machine wrapping access to CPS networks. Intuitively, it would seem to be natural property of any computational or mathematical (or even linguistic) environment for which a notation of `f` itself would be meaningful — e.g., denoting a possible computation, or a mathematical function which yields a value when parameters are fully specified — then so too would be variants of `f` with greater or lesser arity as notated (here) through red-blue-gray strings. In other words, if we take some semantic predicate — say, *describes a computation* — then we may want to consider the full set of functions which meet this criterion that can be generated from some initial "seed" collection. Given `f` as a seed, that is, one wants to elucidate the set of functions (expressible solely via `f`) which describe a computation, for instance (or replace "describe a computation" with some other predicate). We seem to have:

Lemma 1. Assuming `f` designates a function with arity a, there is a unique list of red-blue-gray strings which each describes a function, mutually structurally distinct, enumerating all functions that can be described on the basis of `f` alone (including `f` itself). (Note that we are considering functions generated only by extending or contracting `f`'s arity; a different enumeration would involve recursive descriptions where a call involving `f` yields a value which is input to another call involving `f`.)

Proof Consider strings with exactly a (not necessarily distinct) variable-symbols. Without losing generality, we can assume that symbols (blue-colored numbers) with lesser numeric value appear to the left in their first token. Let $a' < a$ be the number of *distinct* variable-symbols in a given string; i.e., the arity of the function being described as a variant of `f` (as compared to the arity of `f` itself). Then any a-length permutation of a' numbers (in the range $1 \ldots a'$) — restricted to cases where the first occurrence of smaller numbers is always before the first occurrence of any larger number — describes a structurally unique variant of `f`. Analogously (distinguishing blue and light-blue) the darker-blue-only substrings are increasing by one starting at one, and light-blue symbols can be freely interspersed except that any light-blue n has to follow the corresponding darker n. For each of these strings, consider the strings themselves and then modifications which add the gray-color numbers starting with $a' + 1$, and so forth, where the gray-colored numbers can only occur in an arithmetically (-by-one) increasingly list. Each string generated in this fashion represents a structurally distinct variation on `f`, suggesting that the full set of strings generated accordingly embodies all `f` variations, at least those which could be notated by a language that encodes function-calls through symbols representing input parameters.

Definition. We will call a red-blue-gray string "lambda feasible" if it satisfies the restrictions employed above vis-à-vis darker-blue and gray numbers.

Generalizing this analysis to something like lambda calculus requires notating function-composition, by substituting nested applicative structures for variable-symbols. For this context we propose extending "red-blue-gray" notation to include "black" numbers that stand in for nested structures. Call a "red-blue-gray-black matrix" a set of rows formed from red, blue, gray, or black-colored numbers (unlike normal mathematical matrices the rows need not have equal size, though if desired we can always introduce a "dummy" symbol, e.g., a black zero, to pad shorter rows on the left, yielding something that looks like a normal matrix; likewise, colored numbers can always be mapped onto disjoint integer sub-sets, so we can if desired treat red-blue-gray-black matrices as isomorphic to normal integer-matrices).

Lemma 2. For any fixed function-symbol and (ordered) variable-symbol set there is exactly one (countably infinite) set of finite applicative structures in canonical presentation that can be generated from those symbols.

Proof We'll proceed by representing each applicative structure via a red-blue-gray-black matrix. The simplest applicative structures have no nesting, just a function-symbol followed by a list of variable-symbols. Since we are attending strictly to canonical presentation, we can permute the latter list so that lower-numbered symbols appear to the left, and distinct symbol-numbers are allocated incrementally. Symbol-numbers in this context function similarly to "De Bruijn indices" (in a presentation of lambda calculus formalized through de Bruijn's nonstandard notation) mentioned earlier. Lemma 1 argued that we can describe all variations on an `f` *without* composition as the set of "lambda-feasible" red-blue-gray strings with `f` (which can be denoted by a red number as well as a symbol) as the sole function notated. According to the construction rules generating applicative structures, we then have to consider substituting nested structures for variable symbols. Using red-blue-gray-black matrices, encode such nesting by inserting a black-colored number instead of a blue one, selecting black

numbers according to the rule that lower numbers occur before (to the left of and above) higher ones, and that the black-number values map the index of subsequent rows in the matrix; each row, then, is its own red-blue-gray-black string. Re-encode blue numbers in nested structures by adding to their value the smallest offset possible without conflicting with blue numbers in earlier rows. Call red-blue-gray-black matrices subject to these restrictions (on the black numbers as well as lambda-feasible restrictions for each row) "lambda-feasible" matrices. Based on their construction, we claim the set of such feasible matrices is isomorphic to the set of applicative structures, so that the lambda-feasible red-blue-gray-black matrices offer a systematic enumeration of all applicative structures.

There is a certain amount of mathematical bookkeeping implicit in the above presentation, which might obscure the fact that applicative structure are very basic; they should indeed by seen as rudimentary to the point of being rather uninteresting in themselves. More involved systems however emerge by relaxing certain conditions; for example, we might consider allowing self-referential applicative structures that express infinite recursion. The "red-blue-gray-black" matrix form points to one plausible avenue for modeling this process, because negative matrix entries could be allowed to "refer back" to prior rows, notating the idea of instruction-sequences in a computing machine looping back to prior instructions. Other classical developments (often phrased within lambda calculus rather than applicative structures *per se*) involves encoding natural numbers via repeated iterations of a single "successor" function (one being the successor to zero, etc.), such that — again appealing to matrix-notion — there is a sequence of matrices encoding the sequence of natural numbers. Any application of functions *to* natural numbers can then be encoded alternately as a *substitution* of these special matrices for the relevant variables.

Additional applications of applicative-structure theory turn on the notion that applicative structures model function-composition semantics. To the degree that we can (with suitable semantics) treat functions as *values* — as points in an ambient function "space" — then functions may be composed in various ways to generate new such values. In the simplest case (functions of arity one), at least without recursion, composition can be analogous to an algebraic operation — from `f` and `g` get $f \circ g$ (and $g \circ f$, which is generally different). Once one or more functions has arity two or greater, however, we have a set of multiple composition-options — $fx(gx)$, $fx(gy)$, $f(gx)y)$, $f(gx)(gx)$, $f(gx)(gy)$, for example, are all potential compositions of a two-arity `f` with unary `g`, listing only non-root-recursive structures where `f` takes priority over `g` (the full list as such is larger, and, if we allow unrestricted recursion, infinite). We can therefore describe each form of composition via applicative structures, yielding a more complete description of function values' semantic terrain than is designated via individual function-symbols themselves (this discussion leaves unaddressed the question of whether there are function values that can *not* be encoded via applicative structures).

Depending on one's perspective, applicative structures can be seen as either essentially semantic or syntactic phenomena. Syntactically, we can treat these structures as characterizing valid strings in a language comprised of function and variable symbols, and one single grouping construct (via nested structures, which syntactically take the form of sub-terms that can be grouped into quasi-atomic units, substituting for the actual atoms, viz., variables). Certain syntactic formations, however, also seem to have semantic interpretations: as already observed, the $f \circ g$ (and $g \circ f$) compositions correspond to what are implicitly semantic relations, that is, the composition of two functions to yield a new function. Moreover, mappings *between* applicative structures sometimes appear to express semantic relations: $g \circ f$ is the compositional *inverse to* $f \circ g$, and the inverse of one (say, binary) function fxy could be notated as fyx — in short, we can extend applicative systems to treat certain non-canonical structures as notations for variational forms of functions derived from their "base" forms by substituting which argument is placed in which position, the simplest example being $fxy \rightarrow fyx$ (note the similar point mentioned earlier, that partial applicative structures can be read as designating "meta"-functions). The fact that some applicative structures thereby have semantic interpretations — even if we consider applicative systems as essential syntactic constructions (enumerations of valid expressions in a certain simple class of formal languages) — has led some researchers to consider applicative structures (particularly in the guise of Combinatory Logic) within fields as diverse as linguistics and psychology. Combinatory Logic essentially uses a set of combinator symbols (external to both function and variable symbols) and string-reduction rules to enumerate all applicative structures generated by a system's intrinsic symbol-lists [18, page 5], [27], [14], [56]: for any structure formed from symbols $f_1, ..., x_1, ...$ (fs and xs being functions and variables respectively) there is a combinator $\mathbf{C}$ such that the string $\mathbf{C}f_1 \cdots x_1 \cdots$ reduces to the relevant structure (where $\mathbf{C}$ can itself be a string of other, more primitive combinators). In this context combinators play an enumerative role analogous to (what we are calling) red-blue-gray-black matrices (although we personally find combinators' reduction rules feel more ad-hoc than the state-machine-like interpretation one can give to a red-blue-gray-black matrix; we'll leave the details to a footnote).[4]

The linguistics-based interest in combinators evidently reflects the idea that certain applicative structures (and intra-structure morphisms) codify compositions or modifications which express semantic operators, and, in general, certain applicative structure embody syntactic constructions that are sufficiently common or entrenched as to emerge as semantic conventions, not just syntactic forms (the underlying principle, articulated for example in Construction Grammar, being that semantic constructions originate at least in some cases from recurring syntactic formations, such that *syntax per se* — novel phrases, say, which rely on grammar rather than idiomatics to signal the intended composition — represent forms that are *not* sufficiently entrenched as to have "automatic" meanings, e.g., those of single words, but instead need to be parsed) [51], [26], [41], [1], [13], [19], [50]. In this context combinators are at least intuitive emblems suggesting how syntactic entrenchment yields semantic conventions [6, page 6], [7, Chapter 9], [16], [49], [59]. More generally, we can also observe in the linguistic context that the overall space of applicative structures (over all words in a sentence, say) can be partitioned into subspaces (semantic entrenchment being one example, but we can also analyze different applicative structures as having more or less linguistic coherence, based on criteria such as verb-to-noun relations).

[4] An intuitive way to picture Combinatory Logic is as follows: disregard the interpretation of function/variable symbols as functions and arguments, respectively, and consider merely symbol-strings as expressions in a formal language. Extend this language with combinator-symbols that are associated with reduction rules that impose a partial order on the set of permissible strings (such strings are differentiated by a grouping operation, potentially nested, as well as symbol-lists). A canonical example is the $\mathbf{S}$ combinator such that $\mathbf{S}fgx \rangle\!\!- fx(gx)$ (where $\rangle\!\!-$ notates reducing to) or the $\mathbf{B}$ that appears to connote composition (seen as semantic): $\mathbf{B}fgx \rangle\!\!- f(gx)$ (or the famous "Y" combinator, which can be described by the rule $\mathbf{Y}g \rangle\!\!- g\mathbf{Y}g$). Note that "reduction" here, however, at least in the mathematical presentation of Combinatory Logic, does not have an explicit semantic interpretation, but merely notes that some strings come before others in the $\rangle\!\!-$ partial order; moreover, strings in the language contain free-form admixtures of combinators, functions, and variables, which have no apparent semantic interpretation (the general premise being that only "maximally" reduced strings should be approached semantically). We find these details to render Combinatory Logic proper somewhat counter-intuitive, at least on its own terms (to be fair, though, although some papers systematize Combinatory Logic in isolation, it is more common to adopt combinators as merely notational conveniences for certain constructions in lambda calculus).

Apart from *syntactic* interpretations, applicative structure can also be seen from semantic angles in more narrowly mathematical contexts, for instance when we consider properties *of* functions, such as those derived from algorithm-theory. As an example, if `f` denotes a function which can be evaluated through a calculation implemented with guaranteed termination, then composing `f` with another function with the same property yields (or describes) a different function which also by guarantee terminates (simply allow `g`, say, to terminate, then feed its value to `f`). More generally, we can take the applicative system over a set of functions sharing properties such as guaranteed-termination to be a larger set with the same property. In this sense we can regard applicative structures as part of the backbone for analyzing different kinds of function-spaces according to algorithm-theoretic properties or profiles (e.g., computability, termination, different complexity classes, and the like). These potential applications, alongside those mentioned above in the context of recursive function theory and infinite recursion, as well as encoding number theory/arithmetic, constitute some of the extensions to underlying applicative-structure theory which have some mathematical significance.

Our concerns here are less mathematical, so we will extend applicative constructions in a different direction, grounded in graph theory. We will suggest that applicative structures have natural correlates among (directed) graphs, and that this setting is in some ways more intuitive and less cumbersome than the above presentation appealing to notions like "substitution" and function/variable "symbols."

Outside of mathematics proper, there is still some value in enumerating the full collection of applicative structures (generated by a preliminary function-list). If we treat these structures as syntactic phenomena, then intuitively a plausible *semantics* for notions of function-application would consider formally distinct applicative structures as semantically distinct entities. The mechanisms for enumerating distinct structures may be less important than the mere fact of having a well-defined algorithm for producing and screening for all valid constructions. We use (what we call) red-blue-gray-black matrices to provide these criteria: intuitively, a reasonable semantics for applicative structures would recognize every formation described by a distinct such matrix as semantically distinct, and recognize the set of lambda-feasible red-blue-gray-black matrices as a complete listing of all valid strings in a language that describes functions (for some meaning of "function") derived from kernel "seed" functions by arity-expansion (via unused arguments), projection (arity-reduction through repeated parameters), and composition. Later in the section we will discuss more specific semantics, but the enumeration with red-blue-gray-black matrices yields an initial picture of the space over which such a semantics should quantify.

2.2 Hypergraph Models of Calling Conventions

The definition of function-application modeled via applicative structures (and similarly the lambda calculus) should be deemed insufficiently expressive for analyses related to modern programming languages. To be sure, in the above discussion we were evasive about functions' "semantics," such that an incomplete account of "application" *per se* is arguably warranted by generality. When turning to programming languages in particular, however, we have more detailed semantic notions to work with; for instance, *functions* can be considered as computational *processes* which follow some *procedure* so as to (in general) derive an output in the presence of given inputs. Such an overview is still mostly intuitive and informal, but it begins concretizing the notion of "function" in *procedures* which, presumably, in the general case execute series of smaller steps in sequence (by contrast, say, to regarding functions set-theoretically as mathematical relations between inputs and outputs, each function being a subset of the total space of mappings conceivable between its domain and codomain).

Given the functions-as-procedures paradigm, a first step is to broaden the notion of applicative structure appropriately to accommodate how programming languages (via which procedures are implemented) recognize multiple *forms* of inputs and outputs (objects vs. ordinary parameters, say, or thrown exceptions as opposed to ordinary return values). To designate these variations in forms of inputs/outputs, we will describe procedure parameters as being grouped into "channels," which can be interpreted as gathering a node's adjacency set into (potentially) two or more partitions. We will introduce some nonstandard terminology, hopefully justified by the Virtual Machine model which "drops out" of the theory we're sketching here, as a practical use-case. The central underlying notion is to designate certain graphs as "syntagmatic" insofar as they model procedure calls with (in general) multiple parameters grouped into variegated "channels" (an explanation for the choice of the terminology *syntagm* and *syntagmatic*, usually encountered in linguistics, is offered in Chapter 6 of [45]).

Definition. A *channelized in-neighborhood* (we can drop the "in-" when it is clear that out-neighborhoods are not relevant) of a directed graph is the in-neighborhood of one vertex `v` where the edges incident to `v` are ordered and grouped into (typically) one or more *channels*. A "channel" in this context can be considered, most generally, as any collection of edges (focusing on the edges themselves, not their incident nodes; in the case of labeled graphs channels would then be, in effect, sets of label-tokens) but define a *syntagmatic channel* as a channel all of whose directed edges share the same target node; by default *channel* and *syntagmatic channel* can be used interchangeably. The central vertex `v` might be labeled with a string taken from a prior collection of names (intuitively, name of procedures insofar as graphs diagram procedure-calls). Channels may also have "descriptive" labels (which serve to associate channels with channel "kinds").

Definition. A *channel system* on a directed graph is a set of criteria constraining the legal channelized neighborhoods around any vertex. Example restrictions would be stipulations that vertices cannot have multiple channels with the same kind, or (in some context) restricted to specific possible channel-kinds, or that channels need some fixed number or range of edges (e.g., a channel embodying procedural *outputs* may be restricted to have at most one edge). More detailed restrictions can apply to multiple channels in interaction. Consider modeling the alternative between normal procedure *outputs* and *exceptions*: throwing an exception precludes returning from a procedure normally, and vice-versa. In terms of channels this means that a non-empty channel representing (normal) outputs precludes a non-empty channel representing exceptions, and vice-versa (more precisely, as clarified below, the two channels cannot simultaneously have edges out-incident to nodes with *non-void state*).

Definition. A "lambda-restricted" channel system is one specific system which it is convenient to name ahead of time, intended to mimic a minimal lambda calculus. See the following lemma.

Instead of an open-ended notion of "applicative structures" we can speak of applicative systems *constructed relative to* channel systems. This more general sense of applicative structures builds up incrementally from channelized neighborhoods conformant to the relevant channel system. By way of illustration, consider a graph-theoretic encoding of "classical" applicative structures (the form embodied in lambda calculus) as defined earlier.

Lemma 3. Any "classical" applicative structure can be modeled via channel systems with the following characteristics: there are two channel kinds — inputs and outputs — and output channels must have exactly one edge. Call channel systems subject to these limitations "lambda-restricted" as anticipated above.

Proof Proceeding by induction on the generative rules for (the kind of) applicative structures defined above. The simplest case is one function-symbol followed by some number of variable-symbols. Express this via a neighborhood with one "procedure" node (standing for a procedure to be called) and a collection of "argument-nodes" with edges pointing in to the procedure-node (all these edges grouped into one channel). Next, we expand outward to encode nesting (substituting structures for single variables). Consider two channelized neighborhoods N_1 and N_2 restricted to the channel system described in the lemma statement (inputs plus exactly one output). We can form a connection from N_2 to N_1 by inserting an edge between the vertex in N_2 which is adjacent (in the outgoing sense) to the single output-channel edge in N_2 and one of the input-channel edges in N_1. That is, the out-adjacent vertex in N_2 will now have two out-edges, one targeting the function node in N_2 and one targeting an argument node in N_1. We can see by induction an equivalence between this graph-theoretic model and the original closure-definition for applicative structures. Indeed, introducing these connecting edges is structurally analogous to "substituting" nested structures for variables. We'll clarify that claim with a further definition.

Definition. The *syntagmatic neighborhood-set* of a vertex is an expansion of channelized in-neighborhoods by connecting a different neighborhood to an initial neighborhood via an edge between a "peripheral" node in each neighborhood (i.e., excluding the vertex in-adjacent to other nodes in the neighborhood). Because each channelized neighborhood could have its own syntagmatic neighborhood-set these structures can model nesting. Stipulate that any syntagmatic neighborhood-set proper has exactly one channelized neighborhood which does *not* have a connection to another neighborhood; i.e., all of its vertices have at most one out-edge. Call the central node of this neighborhood central for the neighborhood-set overall, and stipulate that syntagmatic neighborhood-sets must be *connected* in the sense that the central node of each component channelized neighborhood must be connected to the central node (via paths allowing either in- or out-adjacency).

Definition. An *input ring* around a syntagmatic neighborhood-set is a collection of labeled nodes (each with distinct labels) that are not otherwise parts of the construction. Consider the ring nodes to be connected with (non-procedural) nodes via a "supplication" edge (intuitively, to represent the idea that the node is associated with a value based on its supplier-node's label); two nodes adjacent to the same ring node via such edges have "shared supplication."

Lemma 4. Syntagmatic neighborhood-sets within a channel-system restricted to output/input kinds (and max-one output channels) can be unambiguously encoded via matrices equivalent to "red-blue-gray-black" matrices represented earlier for applicative structures in canonical presentation.

Proof Label procedure-nodes (i.e., center-nodes) with *red* numbers based on their string labels (assigning like numbers to like strings). Edges in channelized neighborhoods are ordered, so form matrix rows by notating the procedure-number followed by *blue* numbers assigned to argument-nodes; if an argument-node shares supplication with a different node already numerically labeled, adopt that number, otherwise adopt the least available numeric label. This applies only to argument-nodes which are not connected to other nodes across syntagmatic neighborhood-set connections. In the latter case, label the "nested" channelized neighborhood with *black* numbers (starting at one for the central node overall and incrementing as needed) and, for argument-nodes connected to other neighborhoods, insert a negative integer whose absolute value is the external neighborhood's index number. Create a matrix row for each channelized neighborhood, in order of their index numbers. The resulting matrix will structurally mimic red-blue-gray-black matrices in the context of applicative structures outlined above.

Theorem 1. Applicative structures as presented earlier and syntagmatic neighborhood-sets within a lambda-restricted channel system are isomorphic to the same set of lambda-feasible red-blue-gray-black matrices, assuming they share the same set of function-symbols/procedure-labels.

Proof Lemmas (1) and (4) detail the construction of red-blue-gray-black matrices from applicative structures and syntagmatic neighborhood-sets respectively. We claim that comparing the two constructions documents that each step in the construction would modify the resulting matrices in equivalent ways, and so the set of matrices generated from applicative structures will be identical to that generated from syntagmatic neighborhood-sets — both are precisely the lambda-feasible matrices according to the earlier definition of lambda-feasibility which precludes infinite recursion.

In other words, lambda-restricted syntagmatic neighborhood-sets are a graph-theoretic encoding of applicative structures in the classical lambda-calculus sense.

We have formally reviewed "classical" applicative structures, however, primarily to demonstrate that within the context of channel systems these structures represent only one (relatively restricted) model of function/procedure-application, characterized by a simplified (input/output) channel semantics (plus at most one output per function). The constructions for channelized neighborhoods and syntagmatic neighborhood-sets can be naturally extended to more expressive channel systems, yielding (so we claim) graph representations which are more consistent with actual programming languages (whereas lambda calculus models a kind of abstract programming language for purposes of mathematical treatment).

There are at least two significant extensions which can be made when transitioning from "lambda-restricted" channel systems to ones that are more free-form: first, channels can have multiple kinds (beyond just input and output) and, second, it is possible for the output to one procedure to be a *function value* which in turn is applied to other arguments. The latter scenario implies that an output node (in one neighborhood) can be linked (via a directed edge) to the *central* node of a different neighborhood, not just to one of its argument nodes (in this case the central node would not be labeled with a string denoting a function-name, but rather would receive a function-value from that output edge).

Definition. *Syntagmatic Graphs*: Consider syntagmatic neighborhood-sets as above, with the following generalizations: each component syntagmatic neighborhood-set may have channels of multiple kinds (though we can maintain, as an optional restriction, the stipulation that each neighborhood has at most one channel of each kind), and the central/procedural node of a neighborhood may have an incoming edge that passes a function value assigned to that node. This latter possibility can be accommodated by defining a special "function value" channel kind which may be occupied by at most one edge; the non-central node incident to that edge can be considered a special form of argument node, one which has an incoming edge from another neighborhood (representing a passed function-value), but instead of this argument node being an *input* parameter to the central-node's procedure, it is a function value representing the actual procedure which that node iconifies. This formation would thereby model situations where nodes encapsulate numerous possible procedures and the actual procedure designated is dynamically calculated (presumably just prior to the function-call which is notated via the syntagmatic graph).

Intuitively, each Syntagmatic Graph describes a structure connecting one or more procedure-calls, interconnected by parameter passing — outputs from one procedure become inputs to another. Working in a more flexible "channel system," however, the connections between such calls may have more subtle relationships than output-to-input chaining. For example, multiple nodes in a syntagmatic graph may represent the same "variable" (cf. "shared supplication" nodes from earlier) which in turn might be modified by one procedure, so an input given a new value by a procedure and then passed to another procedure is a form of inter-procedure precedence — the effects of the first are consequential to the second — even if the two are not linked by an explicit output-to-input handoff. These are the kinds of scenarios that a procedure-call semantics reflecting actual programming languages (as opposed more strictly mathematical function theories) should address; they will be the focus of Section 3.

3 Semantic Interpretation of Syntagmatic Graphs

Assuming we remain within the context of applicative structures proper, the point of these formations is to represent the idea of functions (in some sense) which take input values. Notating this process via variable-symbols allows function composition (and arity-reducing projections) to be described; thus we have "substitution," replacing variable-symbols with concrete values. When all free variables in an expression describing (potentially nested) function-applications are thus substituted, we have sufficient information to evaluate the function, or — in the sense of lambda-calculus "beta" reduction — reduce (or "collapse") the aggregate expression to a single resulting value. This is the central dynamic figured by applicative structures in their simpler, classical sense — input parameters yield applications which "reduce" the inputs to a single output value, that may in turn be input to other functions. Chaining inputs and outputs in this sense engenders a model of computations as graphs, where edges encode how values travel between applications, with multiple inputs potentially each being outputs from multiple precursor function-applications.

The semantic interpretation we adopt here for "syntagmatic" graphs is noticeably different than this model (we have analyzed the specific differences in [45, page 150]). Rather than taking input/output as a fundamental divide within function-parameters, we consider more general channel systems here. Note that in contrast to (more typical) graph-representations of computation where outputs are marked by directed edges *away from* procedure nodes, in Syntagmatic Graphs (hereafter SGs) all edges point *into* procedure-nodes, including those embodying "outputs" ([45, section 9.2] has some comments about why this can make sense). More elaborate rationales for the notational and interpretive differences between SGs and other procedure-models is tangential to the current chapter, so we'll focus instead on outlining the semantic interpretation itself.[5]

Definition. Call a *marking* of a Syntagmatic Graph to be an association between some of its nodes and a collection of typed values, against some type system. Markings might be context-dependent; that is, a graph could be subject to multiple markings concurrently, each restricted to one context. We'll say that values are *bound to* nodes, but indirectly, as explained in the following.

Definition. Syntagmatic Graph nodes can be associated with *types* and *states*. For the present, we will not rigorously define "types," but we'll comment that types are introduced vis-à-vis nodes *through* states. For any type, each possible instance of that type is a potential *state* for Syntagmatic Graph nodes, the state of being "occupied" by that specific value. However, not every node-state need correspond to a type. In particular, nodes can have a state corresponding to a "void" or lack-of-value.

That is, we approach the notion of "void" as a *state* possibly evinced by nodes, rather than through type systems themselves. That is, we do not need a "bottom" or "nothing" type with some special value selected (essentially by fiat) to represent non-initialization or pre-initialization. We find it more semantically coherent to express such "nothing" states as the state of *having no value*, rather than as the presence of a construed "nothing" value (and nothing-type which it instantiates).[6]

Against this background, the semantics of procedure-calls can be expressed via before-and-after states for each argument-node incident to a procedure-node. The procedure's semantics is, as such, the cumulative state-changes, or *change in marking state*, effectuated by the procedure. In a general case, some of these changes will be void (no-value state) nodes transitioning to having a value; these would typically correspond to "output" nodes in classical applicative models. However — accommodating scenarios like mutable references — nodes with the state of binding to some value could migrate to the state of binding to a *different* value. In general, notions such as input-versus-output are expressed in this system "semantically" as a manifestation of state-changes, rather than "syntactically" as edge-direction or arrows assigning different directions to inputs versus outputs.

Roughly as an analog to "beta" reduction, we propose the term *digamma* reduction to express marking-state changes due to a procedure-call. A digamma reduction is the cumulative state-change in all nodes affected by the procedure (or, seen syntactically, all nodes adjacent to the procedure-node). For multi- (channelized) neighborhood SGs, digamma reduction happens in multiple stages, each contextualized to a single neighborhood. Note that we like the term "digamma reduction" partly as an oblique reference to "sigma" calculus (an object-oriented extension to lambda calculus, and the Greek digamma numeric symbol looks like a enlarged lower-case sigma) and partly because "gamma" is a common symbol for graphs, so "digamma" suggests "two graphs," or a computational interaction between one graph assembling a procedure-call and one graph implementing it.

Semantically, then, SGs model digamma reductions *in sequence*, each localized to individual channelized neighborhoods. This idea can then be extended to sequences of SGs in turn. We are getting closer to representing actual computer code; one further step might then be incorporating something like "stack frames." Assume that SGs are represented *in sequence* and moreover that such sequences occur in the context of a *symbol scope*, or an environment where string labels can serve as "carriers" for typed values; we'll sometimes refer to " SG-scopes," scopes contextualizing SG sequences wherein symbols designate carriers which (when in an initialized state) hold concrete values. In particular, assume that carriers can reveal "carrier states"

[5] Some of this terminology is derived from Petri nets [24], [48], [46], [53], [2], [38], [34], [36].

[6] Note that in type theory we can also define "bottom" types via constructions that preclude bottom from ever being instantiated at all (e.g., by stipulating that bottom is a subtype of any other type, i.e., the subtype relation between that and any type is defined to be true). A type system may have an uninhabitable type insofar as predicates on intertype relations (such as subtype-of) can be non-paradoxically defined on the full collection of types, including uninhabited ones. A bottom type does then have a reasonable correspondence with uninitialized code-symbols because this is the only type that correctly characterizes the state of such symbols prior to their being assigned a value (without a value, there is no inhabited type where we can say the symbol's value is an instance of that type). Still, the purpose of assigning a type to symbols in practice is to constrain how symbols can be used (when passed to procedures, in particular), and for these purposes it is arguably misleading to use type-attributions *per se* to model value lifetimes. For this reason it seems better to capture the fact of pre-initialization via a *state* characterizing the symbol itself rather than a *type* attributed to its value.

reciprocating the states defined on nodes earlier: for any type there is a spectrum of states equivalent to each type-instance, but there are also states involving *lack* of typed values (the generic example of such a state we'll call *pre-initialized*). The states available for carriers could be called "type-based" in that they are organized around typed values but include non-type states as well. SG nodes may then take on states by associating with carriers such that carrier-state propagates to the nodes. Nodes are in "shared supplication" if their carriers are synchronized to be perpetually in the same state (this can be considered a refinement of the earlier definition). Symbols in SG-scopes might derive from multiple sources: assume that scopes' symbol-list can be grouped into *declared* symbols internal to the scope, *argument* symbols bound to procedures' signatures, and (potentially) *global*, or perhaps "ambient" symbols shared among multiple scopes (we'll use "ambient" for the general case and "global" for analogs to "global variables" in typical programming language).

Definition. *Tripartite Scope*: We'll call SG scopes *tripartite* in that they are mappings from symbol-names to carriers (or groups of state-synchronized carriers) with type-based states, and symbols can be classified as *declared*, *argument*, or *ambient*. The semantics of these groupings will be clarified below.

Definition. We will say an "SG-described procedure" (or just "procedure" when the SG topic is obvious) is an SG-sequence unfolding in a tripartite scope where argument-symbols are bound to values held by carriers in a *different* procedure (or, recursively, the same procedure in a different context, viz., different symbol-bindings). Such interactions between SG-described procedures model the semantics of procedure calls. Of course, procedure-calls in turn are modeled by channelized neighborhoods in individual SGs. Specifically, channelized neighborhoods embody procedure-calls wherein peripheral nodes' carrier-states in the *calling* neighborhood become the basis for initializing carriers in the called procedure; specifically, the carriers bound to argument symbols in the latter's tripartite scope. The switch in execution context from the calling to the called graph is analogous to binding symbols to values in process calculi, or beta-substitution in lambda calculus, except that carriers' connections (via their argument nodes) to procedure-nodes is organized via channels, which can affect procedure-call semantics. For example, it may be stipulated that carriers in channels modeling "outputs" are understood to be in pre-initialized state for the duration of called procedures' execution, and attempting to utilize (e.g., read values from) such carriers *within* the procedure is a logical error. There are various ways of enforcing this kind of restriction, of course, but channels offer a convenient representation of the relevant constraints viewed under a semantic interpretation.

We now have a semantic interpretation of channelized neighborhoods: each such neighborhood embodies a procedure-call, which entails binding call-site carriers to procedures' argument symbols and then carrying through other procedure calls notated via SGs associated with the *called* procedures. When they are finished, the original carriers will (potentially) be in a modified state, so the called procedure effectuates a "digamma reduction" on carriers around the calling site. State-changes then propagate across channelized neighborhoods insofar as values from output-like channels in one neighborhood are handed off to input-like channels in later-executed neighborhoods. Each SG has a "top-level" neighborhood which is the last to "execute" (viz., compel a procedure-call) or, from a different perspective, the root of a directed acyclic graph showing carrier hand-offs and shared supplication across channelized neighborhoods. Note that output channels on the top-level neighborhood cannot be connected to other neighborhoods in the form of procedure outputs become inputs to a subsequent procedure (there is none) — so either top-level procedure-calls are useful solely for their side-effects or the carriers in their output-like channels are bound to symbols in the current scope. In the general case, we'll say that an SG is *anchored* by one or more symbols if those symbols (strictly, the carriers associated with them) acquire values from the SGs top-level neighborhood (particularly from the channels there which obey output-like semantics).

The mechanism just described constitutes how *declared* symbols in an SG tripartite scope can acquire values; they become initialized from procedure-calls enacted *in* the procedure, rather than from arguments passed *to* the procedure (or from an ambient space of values visible from multiple scopes). Once initialized, declared symbols can then supply values for procedures called *inside* a procedure — that is, an argument-symbol in one procedure can derived from a declared symbol in a calling procedure. It is reasonable to assume in the typical case that all procedure-symbols (whether declared, argument, or ambient) hold values which originate from a *declared* symbol somewhere, so the anchoring-points where declared symbols are first initialized represent precisely the points in any holistic collection of SG-described procedures where new values are "introduced" into the system.

This last point has implications for how we theorize *typed values* as well. Implicitly here we have assumed that we are operating in a strongly-typed system where types and values are interconnected: any value is *typed* (an instantiation of a type) and types exists by virtue of their possible values. We assume that in the general case types are not equivalent to their *extensions*, that is, to any fixed set of values. There are indeed some types whose extensions are fully "enumerable," so we can specify exactly how many possible instances a type has: the type corresponding to signed one-byte integers, for example, denotes precisely the set of numbers at least -128 and at most 127 (the analogous range for *unsigned* bytes is 0-255). However, for types such as *lists* of integers, the extension cannot be specified *a priori*, in the sense that it is impossible to know from a specific computing environment (at a specific moment in time) whether or not a particular value (here, a particular list of integers) which logically fits the type's criteria can in fact be represented. The size of a list which may be instantiated is limited by factors such as the computer's available memory; in theory, the extent of a collection-type like "list of integers" is infinitely large, because there is no rule to group all potential such lists into a finite set (if there were, take the longest list, an an integer to the end, yielding a new list which has no reason not to be included in the first place). We assume moreover that we are working in a *non-constructive* type environment where we cannot consider types' extensions as a logical construction abstracted from computational feasibility in concrete computing environments. We can discuss types' *hypothetical* extension, which may indeed be infinite, but we assume we need some other mechanism to manage uncertainties regarding types *actual* extension (in some contexts, such as lazy-evaluated lists or ranges, working directly with a logical model abstracted from actual realizability is appropriate, but in contrast to functional-programming theory, for example, we do not take such "constructive" formalizations as an essential aspect to the relevant type systems).[7]

So as to give a rigorous semantics to *non-constructive* types, instead, we leverage the interconnections between types, symbols, and SG-nodes insofar as we are working in an SG context. Recall the heuristic that

[7] See [45], Chapter 4, for a more extensive analysis of non-constructive type systems and why assuming that types are non-constructive in general (rather than the opposite) is practically appropriate.

all symbols acquire values originating from declared symbols, which in turn have a specific initialization-point (although some symbols might be initialized by non-constant reference, passed into a procedure for initialization rather than being bound to a procedure-result, the values they are bound *to* in this case will itself be determined by an anchoring-initialization, either in that procedure itself or some other procedure called in turn; so, even if not all declared symbols receive their values via anchoring, all declared symbols receive values which *originate* from an anchoring somewhere). In other words, when we say that a symbol has some type value, we mean that a symbol's value is derived from an initialization from anchoring point, where a declared symbol's value was set via digamma reduction at an anchoring point — the symbol was associated with a node which transitioned from a pre-initialized to a type-bound state as a result of binding to nodes in an output-like channel. Symbols may be "re-anchored," initialized to new values due to anchoring points, but (assume we work in a fixed-type environment) symbols and carriers retain the same types once they are initialized, so long as they remain in an initialized state. Types themselves, then, can be construed in terms of continuities in carrier-state: once a carrier is initialized to be in a state determined by a type-instance, it will remain in states determined by the same type so long as it is initialized at all; and all initializations can be traced to anchoring-points, together with synchronization among states of distinct carriers. We do not need to theorize types' logical extensions in general, because we aren't concerned with the totality of a type's possible values, only with the states directly associated with the type at specific anchoring-initialization sites.

Many programming languages distinguish between *constructors*, which are intrinsic to a type's definition, from other procedures that return values of a given type. To the degree that this distinction is in effect, any value can be traced back specifically to a *constructor* for its type. To be sure, a value might be bound to the output of a *non*-constructor procedure, but in that case this return value will have been formed initially by a constructor proper called in that procedure, or perhaps in a procedure it in turn calls, iteratively. In a channel system we can leverage a comparable distinction be defining a special kind of channel, which functions like a normal output except that procedures whose signature includes such a channel are considered to be constructors, or analogous to constructors (in [10] we used the term "co-constructor" to suggest allowing procedures being analyzed as if they were constructors even in programming contexts where, for technical reasons, they would not be classified as such according to a programming language whose code is being analyzed). For generality we'll call procedures with constructor channels "co-constructors" with the assumption that different environments may be more or less lenient in conditions where co-constructors may be declared (only in code specifically defining a type, say, in contexts where a narrower policy is warranted, but without imposing restrictions along these lines *a priori*).

The significance of co-constructor channels is that they allow us to pinpoint the specific anchorings where new typed values are assigned *ab initio*. Assuming an SG system uses co-constructors, for any carrier in a type-bound state we can be sure that the value traces back to a digamma reduction on a carrier specifically situated in a co-constructor channel. As such, our semantics for types may not give a set-theoretic account of types' extensions, but we do have a mechanism for locating the origination point for all typed values. In particular, values are acquired via carrier state changes localized to co-constructor channels; such a change only occurs when a procedure has completed with an initialized value in that channel available to be bound to its eventual symbol. Any value therefore witnesses the fact that a procedure with a co-constructor channel completed and triggered a state-change accordingly. If a type system is designed to recognize types as (associated with) certain contracts, such that a type being instantiated confirms certain properties about the value thereby manifest, we can recognize such contracts in an SG-style system so long as co-constructors adhere to type-specific contracts in the course of depositing values in a co-constructor channel.

Once a carrier with a given type is initialized, its value can of course be handed off to other carriers assigned the same type (or potentially a supertype thereof, or some other related type wherein casting is possible). Channel systems and SG representations do not preclude polymorphism — specifically, the labels assigned to procedure-nodes need not uniquely identify one single procedure available to be called, but may instead provide a premise from which to select one of numerous available procedures, each with the same name — so type resolution may come into effect: if there are multiple candidate procedures, the one whose signature best matches the procedure-call's argument types is selected. In the context of channels, such disambiguation extends to recognize different channel-kinds, which become part of a function's signature. In effect, each node in a channelized neighborhood has a type (based on its carrier-state) that might be matched against candidate procedures' signatures. Types therefore are consequential primarily in the context of overload-resolution. The issue here is not types' extension *per se* (two types having the same extension would have no bearing on their appropriateness for matching call-site neighborhoods to procedure signatures; and co-extensive types are not interconvertible unless such casts are implicitly declared according to the same conventions as casts between types with different extensions). However, since successfully matching call-sites to signatures implies that all affected carriers' types are suitably aligned, whatever guarantees on carrier-states are implicit in type-bound states becomes transferred from the calling to the called procedure, insofar as the formers' carrier-states carry over to the latter's argument nodes. Therefore, carrier-states associated with being initialized according to a given type propagate from co-constructor anchorings across all subsequent procedure-calls, with type-specific guarantees propagated alongside.

In effect, types' semantics in such a framework is based on the premise that once there is an *originating* initialization via digamma reduction in a co-constructor channel then there is one specific carrier whose state is thereby changed, and subsequently (presumably) a series of further procedure-calls where that originating carrier's state become synchronized with subsequent carriers in subsequent procedure-calls. Each type is essentially a premise warranting the propagation of this shared carrier-state: the type's presence as a condition on procedure-signatures or on declared symbols indicates that a carrier whose state is synchronized in accordance with such propagation would be in a valid state according to the expectations of the procedure which uses the corresponding value. For the sake of discussion, we'll refer to this approach to the issue of type-semantics as a *propagation semantics*, in contrast to a *set-theoretic* semantics which would construe types' semantic interpretations in terms of their extensions, or a *constructive* semantics which would read type semantics through the logic of constructive patterns that could give rise to values of a given type. The essential point about propagation semantics is that type contracts would be checked at origination points (to the degree that they are in effect) and presumed to have been enforced whenever

there is an originating digamma reduction in the first place, and that subsequently any use of a type as a polymorphism-disambiguating device should proceed under the assumption that any carrier-state inherited as synchronized with the state of an originating carrier in this sense should be deemed a valid state for the called procedure's purposes.

This discussion regarding type semantics has passed over some non-trivial details, such as issues of type casting and inheritance, which we will briefly address in the next subsection.

3.1 Distinguishing Non-Constructive from Extensional Type Semantics

A plausible objection to the above presentation is that it is burdened with special terminology and representational conventions to define constructions which are not radically different from traditional notions of, say, stack frames and lambda-abstraction. Beyond just expanding from inputs and outputs to more flexible "channel" systems, we have constructed a semantic interpretation for these systems based on formulations such as type-based states and channelized neighborhoods, without clarifying the theoretical merits of this overall semantic presentation. Our contention is that the full semantic details (not just, say, extending applicative structures to richer channel systems) become valuable in the context of Virtual Machines. We will attempt to warrant this claim by addressing VM implementation, in particular.

We suggest the idea of digamma reduction and propagation-from-constructors to replace extensional and/or constructive semantics for type theory. Implicitly, then, we propose that (what we here call) "propagation" semantics can displace two other paradigms for type theory, *extensional* and *constructive*. We will try to defend the idea first that this is a reasonable three-fold division in options for type "semantics" and second that there are formal or theoretical merits to the "propagation" approach.

An extensional semantics would essentially proceed by grounding the semantics of any specific type in the set of its possible inhabitants. To attribute a type to a value is thereby to assert that the value belongs to the type's extension-set. This paradigm has certain benefits, to be sure, such as providing an elegant approach to functions defined on a type which have different behaviors for different parts of the extension. As a canonical example, a procedure operating on list-like types may require one implementation when the lists are empty and a different one when they are non-empty. Writing functions with distinct code-bodies for different extensional alternatives (another example would be types with possible "null" values, such as **Maybe** in Haskell or pointers in C/C++) is a widespread practice in functional programming. Extensional semantics gives a concise reading to this coding style: insofar as types semantically are effectively equivalent to their extensions, any function defined over a covering collection of extension-parts is equivalent to a function defined over the type as a whole.

On the other hand, we can model extension-partitions and the prospect of quantifying over them as a feature for procedural implementation without actually equating types with extensions; we merely need to posit that types can be *associated* with extensions in such a way that extension-partitions are well-founded. Even where types have nondeterministic extensions, such associations work so long as code using their partitions tolerates similar nondeterminism at the partition level. For example, the nondeterministic facet of list-like types lies with non-empty types, because there is no way to know how large instances can be (in practice), so this uncertainty translates to the part of their extension including non-empty instances. But code working with such types has to accept uncertainty anyhow (for instance, by recognizing that extending a list's size might fail), so the proper accommodations need merely be accepted in code narrowed to an extension-part that would be present in code without extension-alternation.

A larger issue is perhaps *how* extension-partition alternatives should work; in particular, whether mapping a value to one partition or another is a compile-time or runtime operation. The code-over-alternatives paradigm isn't only a notational convenience; it manifests benefits of functional languages' type system which allows questions like whether lists are empty to be resolved at compile time (at least more often than other languages). If we have a function written so that the code branches differently for empty and non-empty lists (or, say, for valid and null pointers) the relevant programming language might treat this as an alternative to be resolved at compile time or as something deferred to runtime (so that the alternatives exist more or less like if-then execution paths), or some combination. Ideally, it seems, a language engine would compile-time optimize when possible but allow decisions to be deferred to runtime when necessary (rather than insisting on compile-time uncertainty renders code non-compilable at all). By analogy, a compiler might reasonably stipulate that type-attribution ambiguity (as when procedures have two equally valid overloads, say) is a logic error such that offending code should not compile (unless types are deliberately chosen for pre-runtime ambiguity as with "variant" types, such as **boost::any**) but ambiguity in how to map values to types' extensional *partitions* should not necessarily foreclose compilation in the same manner.

These, however, are issues in the context of tracking compile- and runtime information about values, and quality of a language's support for managing value-partition mapping is an orthogonal matter to the semantic interpretation of partitions. In general, languages can take advantage of extension-based reasoning without embracing the theory that types are semantically derived from their extensions.

Moreover, we can similarly distinguish types' semantics from construction-sequences, in that constructive type theory essentially works by using construction patterns as a model for type-extensions, so that the extension embodied by that model fixes the type's semantics. Every non-empty list, say, may be derived from a smaller-by-one list by appending a specific value to its end; so for any list there is a specific construction-sequence which terminates at that list, and we can take the set of those sequences as a model for the list-type's extension. Extension partitions can then be defined over distinct construction-patterns. In the case of lists, non-empty list derivation via smaller-by-one instances is analogous to marking values which can be constructed according to a specific pattern (involving a different list and an **append** operation), so the non-empty part of the extension is precisely the set of instances constructible via that pattern. This is the rationale for functional programming languages branching over construction patterns.

Constructing lists from their smaller-by-one kin actually leverages two facets of "constructive" types: first, that for any given value of a given type we can identify a constructor which produces that value; and, second, that construction-sequences are isomorphic to the type's extension and therefore "model" the type, in the manner that sets endowed with specific operations model "theories" (i.e., axiom-combinations) in symbolic logic. Unless we consider these notions

only as logical abstractions, however, in concrete programming they are often wrong, or at least misleading. If a procedure is supplied a pointer to a list (any list-like type) there is no way to determine a construction-sequence for that value short of examining each item in the list one at a time — especially if the pointer is derived from a binary stream supplied by some external component, along the lines of cross-application interop discussed in Chapter 20. There isn't even a guarantee that we can learn the *length* of the list without similarly scanning the whole value. The presence of a **size()** like procedure depends on the public interface, and there are scenarios (such as **stacks** and **queues**) where the type may only provide **push**/**pop** (or **enqueue**/**dequeue**) access to the collection's state, and no size-related (public) functions apart from a test for **empty()**. In these cases functional-programming idioms related to treating each value as a destructured head/tail pairing have no optimization value compared to procedural if...then branching (although a language could still support these idioms as "syntactic sugar").

Of course, collections types *could* be engineered so that destructuring becomes convenient. If resizable lists are stored via multiple inverted arrays (so that the list-head always has smaller memory addresses than its preceding elements) then obtaining a reference to the smaller-by-one "tail" becomes just pointer arithmetic (we use multiple array-allocations rather than one single array to ensure that elements never have to be moved in memory, which would invalidate pointers/iterators, if the list's size grows beyond the memory initially requested for it). In general, then, the idea of "constructive" types — seen not as a theoretical issue of type semantics but as a tool modeled in source code — depends on how types themselves are engineered, their memory-management protocols and public interface. Similar comments could be made about extension-partitions. Rather than a logical paradigm, we can treat *extensional* and (or and/or) *constructive* semantics as *design patterns* which may guide *some* types' implementations (in the sense of code libraries that realize types via providing implementations of their needed procedures).

Again, though, the fact that some types can be associated with extension-partitions defined via construction-patterns need not signal the semantic equivalence of types and type extensions (or type-extensions and construction-sequences). While recognizing extensional reasoning as a useful tool for language implementation, we believe that it is more conceptually accurate to model types' semantics instead via "non-constructive" notions we discuss here, such as digamma reduction, co-constructor channels, and carrier-state propagation. In other words, we believe that this latter group of formulations better capture how people reason about computer code and design patterns *in the general case*, only switching to the alternative semantics when types' explicit designs warrant as much.

This argument, however, is localized to the question of defining semantics for type systems, which is arguably a philosophical issue more than a directly practical one. We intend to refine the discussion by considering how the semantic notions hereby sketched out could yield practical benefits. In this context we will focus on the hypergraph structure of computer code (within the framework of "syntagmatic" graphs) and the use of channels as structuring vehicles.

3.2 Syntagmatic Graph Sequences as a Virtual Machine Protocol

We are representing Syntagmatic Graphs as hypergraphs, in two senses: partly, for one, because *channels* embody a grouping operation (gathering multiple edges, by analogous to hypernodes being sets of nodes — although channels are not identical to hyper*edges*, wherein *one* edge spans multiple nodes). Second, we also leave open the possibility of SG nodes having internal structure (i.e., becoming hypernodes). However, such hypergraph representation is mostly heuristic; technically, we consider hypergraphs to be essentially a visual shorthand for *virtual machine* constructions. Here we appeal to comments we made in Chapter 20: a hypergraph (or, in essence, any structured representation) iconifies the series of steps needed to construct it in its precise state. For example, focusing on channelized neighborhoods, the elements of such structures — procedure nodes, peripheral nodes, and channels — correlate with operations setting out these details sequentially (identifying a procedure node, e.g., by label; identifying a channel-kind to serve as a context for subsequent peripheral nodes; initializing the latter nodes; switching to a different channel-kind as needed; and so on). This outline could be expanded to Syntagmatic Graphs in general by aggregating multiple channelized neighborhoods and then asserting where carrier handoffs (via cross-neighorhood edges) occur, to SG-sequences by constructing multiple SGs along with their respective anchorings; and finally to procedure-descriptions by enumerating declared, argument, and ambient symbols in a tripartite scope. From this perspective, SG procedures can be seen as compact representations summarizing sequences of VM operations, or perhaps as configurations which guide the construction of VM procedures by defining and end-goal toward which VM operations approach incrementally.

Ordinary VMs, of course, depend on such concepts as stack frames and procedure calls; one of the fundamental operations (or sequences thereof) typical VMs carry out involves push input values onto a stack and then redirecting to the address of a procedure which uses them, at least insofar as the VM in question emulates assembly code to provide a runtime for higher-level programming languages. Modeling procedure-calls via hypergraphs, and in particular *channels* (in the sense we propose here) adds representational parameters to this model, by analogy to how hypergraphs present a richer expressive tableau than other metamodels for encoding data structures. The benefits of added metamodeling detailing the procedure-call context actually help illustrate why expressivity pays dividends with respect to data structures (e.g., in query evaluation) as well.

The primary rationale for VMs is to execute code, of course (whether scripts, queries, or some other programming category that does not feat neatly into one or the other, such as workflows, or one or both ends of remote-service functionality). Secondarily, VM compilation can support static code analysis even if the intermediate code is not actually run. Code analysis factors into execution as well, at least to the degree that a VM can internally support runtime checks and guarantees — insert debugging breaks where certain conditions are met, preventing code from running in certain circumstances, allowing procedures to make contract-like assumptions (stronger than type-checks alone could enforce), and so on. For example, stipulations that values should fit within a fixed range (narrower than theoretically possible via their types) could potentially be verified or implemented by weaving "gatekeeping" code into VM operation-sequences. In short, VMs acquire more flexible capabilities — they present opportunities to implement Requirements Engineering style features, even if only via add-ons — to the degree that they present op-sets and data models configured for static and dynamic/runtime analysis. These observations motivate our proposals to augment the modeling parameters for VMs (in procedure-call contexts) via formulations such as channels and carrier-state.

To be sure, a suite of static-analysis capabilities is endemic to

traditional (e.g., stack-based) VMs as well (reachability, control paths, initialization guarantees, etc.). We would suggest, however, that future generations of VM technology will seek a broader scope vis-à-vis contexts where code analysis is applied, considering GUI programming, Cyber-Physical Systems, multi-modal front-ends, AI integration, and other capabilities that may be unified under the rubric of "Industry 4.0." Future VM architecture may be optimized for the *intersection* of such User-Experience and multi-modality concerns with traditional static analysis and Requirements Engineering.

Consider the case of Channel Systems. According to the SG model we have outlined here, procedure-calls are represented via channelized neighborhoods, and in such neighborhoods all edges incident to a central ("procedure") node lie within a channel. In VM translation, this configuration implies that procedural stack frames are split up into multiple channels, so that whenever an argument is "pushed" as a parameter for some near-future call this action occurs in the context of a specific channel-kind. As a result, argument-push operations may be analyzed in terms of semantic protocols specific to the currently active channel. For example, VMs might want to add extra information in the context of channels representing "message receivers" in Object-Oriented contexts (which we have earlier alluded to as "sigma" channels, essentially the **this** or **self** of languages like C++, Java, Rust, etc.), particularly when such channels contain more than one node (insofar as most languages providing kernel functions accessed from a VM do not allow multiple **this** objects: the demo code for this part's chapters has examples of how to emulate multi-sigma calls in C++, but in general it would presumably take extra effort to map multi-sigma VM calls to the underlying native functions). Moreover, channel-specific semantics can be implemented on an extensible basis, allowing the VMs to be augmented with special-purpose channel kinds which supply their own functionality for managing (in effect) stack frames when such channels are active.

Furthermore, channels can act as a grouping mechanism tieing together procedure arguments which are logically interconnected (to a greater degree than merely co-existing as arguments). An example would be unit/scale-decorated types: suppose a function calculates a formula which requires two arguments with the same dimension and measurement units (kilometers, say) plus a third argument which is just a scalar; evidently the first two arguments are mutually constrained in a fashion distinct from the third. Or consider a calculation with a scalar plus two mathematical vectors which should have the same length — the latter condition might be modeled with a dependent-type construction on the second vector, or some runtime check for the two vectors together, in either case demarcating the vectors as paired arguments distinct from the scalar. One strategy to satisfy these runtime use-cases would be to implement channel semantics where special guarantees (finer than type-checking alone) are enforced while the channels are being populated.[8]

Doubtless, some of the features enabled by channels could be achieved via other means.[9] Channels, however, present a convenient interface for organizing the range of functionality entailed by the relevant channel semantics. Insofar as every argument-node in an SG occurs in the context of a channel, before any such nodes are set in place the VM would first construct a channel of the relevant kind; that is, there is a specific operation to "open" a channel given its kind. This operation is therefore an VM-site (for static or dynamic analysis) that can be targeted by extension code enforcing or examining channel semantics. Once channels are opened, operations exist to indicate the type and value-source for nodes added to the channel. Because these latter operations always occur in the context of a specific channel-kind, they can be filtered or re-implemented based on the channel kind in effect, so extensions could modify the treatment of certain channels while preserving the underlying VM implementation in most cases. For example, a VM extension could modify exception-handling protocols by re-implementing operations for inserting nodes into (or, on the call side, initializing carriers in) special "exception" channels, which are partitioned from "ordinary" output channels (given the obvious behavioral contrast between exiting via exceptions versus normal returns). Procedures typically involve multiple channels, so there are VM operations for "closing" one channel and opening another of a different kind, which results in subsequent operations occurring in the context of a different semantics. Explicitly introducing channels as part of the underlying VM machinery allows variegated channel-semantic protocols to be implemented in an organized and extensible manner.

Last paragraph's discussion regarding (for instance) special-purpose channel semantics perhaps does not obviously connect the idiosyncratic constructions endemic to SG representation with concerns in the latter sense, so we'll try to present a case more concretely. Consider first the issue of "reactive programming" in GUI contexts. Analysis of procedures as a series of calls to other procedures — what we are calling SG-sequence descriptions — fails to directly address how call-sequences in this sense fit into over application execution (we touched on this theme at the start of Chapter 20, as well). In general, an application is not a one-dimensional *program* which simply executes some sequence of operations and then terminates. Instead, applications construct a graphical and runtime environment and then wait for user-initiated actions, responding accordingly, and resetting to a passive state awaiting further user actions. Users signal their intention for applications to take specific steps via interactions which can generically be called *gestures* (e.g., typing something via a keypad, or clicking somewhere via a mouse). Gestures in turn are presented to application code as *signals* that are *handled* by implemented procedures. The overall programming model entailed by composing applications as collections of procedures poised to handle signals — rather than fixed operation-sequences designed in advanced — is generically called *reactive* or (with some additional technical specifications) *functional-reactive* programming [54], [30], [21], [60], [62], [11], [47]. Such a programming model introduces a variety of issues for VM implementation, at least for VMs which model/analyze or execute reactive procedures.

In particular, the central idea of reactive programming is that certain procedures are called (or initiated) in response to signals (typically those due to user gestures, though certain signals may be prompted by non-user changes to the current environment which might be relevant for applications, such as a sudden loss or gain of internet connectivity).[10] Procedures in this sense are not called *from other procedures*, so the normal analysis of procedure-calls in terms of stack frames being transferred from one site to another has to be modified.

[8] In this case the one-channel-kind-per-neighborhood restriction might be relaxed, since mutual connections could exist between (say) some input parameters and not others; or one could adopt something like "subchannels" which semantically refine the channels around them.

[9] For example, aggregating input parameters as mentioned last paragraph might be achieved alternatively via explicit dependent typing or via smashing tuples into single (aggregate) arguments. However, dependent types are notoriously difficult to implement in the context of Software Language Engineering, and the latter alternative could be syntactically unwieldy (if done explicitly in source code) or complex in its own right to implement (if done behind the scenes).

[10] More precisely — because there are not necessarily signals which actually notify applications about, say, network-connection changes — a dropped connection would implicitly *cause* signals to be emitted. For example, an object representing a web request could emit a network-error signal rather than a usual "finished" signal.

A canonical approach to reactive programming involves *signals* and *slots*, with the idea that instead of one procedure directly calling calling another, procedures instead emit "signals" that are *connected* to other procedures, which in such contexts become "slots." Unlike hard-coded procedure-calls, signal-to-slot connections can be dynamically altered, created, or suspended. Such a mechanism depends on a centralized routing component to observe when a signal has been emitted (for instance, added on to an "event queue") and transfer control to one or more slots registered as connected to that signal. Insofar as a centralized processor in this sense is active, it can also (in typical application-runtime frameworks) receive signals originating *outside* the application, i.e., resulting from external conditions apart from one *procedure* emitting a signals. Typically, such external signals would result from user-generated events, such as clicking a mouse button or moving the mouse.

In effect, application code based on signals and slots includes some procedures which are called because they are registered as "slots" whose signatures match signals that may arise from *outside* the application property. These procedures serve as "entry points" where operations specific to the application originate. In other words, applications are environments which, after an initial setup, wait in suspension until external signals initiate a chain of actions in response. The details of such signals cannot be known ahead of time — for example, one cannot say which user gestures will occur (whether the user first types something, or moves the mouse, or clicks the mouse, etc.) nor their specific details (which keyboard keys are pressed, which mouse buttons are clicked, where on-screen the mouse-cursor is located in the latter event, etc.). Application code therefore needs to prepare for multiple forms of external signals, and to analyze their properties to respond correctly (in accord with user intentions and/or design requirements). A left mouse click with the cursor hovering over one GUI element typically signifies a different user intent than a right-mouse click over a different element, for example.

The unique characteristics of reactive programming have numerous consequences for VM design. First, note that applications typically implement many procedures so as to possess requisite capabilities to handle user actions. When and whether a given procedure is called depends on user pragmatics; for instance, a procedure involved in saving a file (at least a file users know about, as opposed to, e.g., a database-related file storing configuration information) would only be called in circumstances where users signal their desire to save files they are currently working on. A well-organized code base will specify which procedures could potentially be called as the *initial* handler responding to external signals. This information makes it possible to factor in external signaling conditions during code-analysis. For example, suppose file-saving is disabled for some reason (e.g., the current user does not have permission to modify the local file system). In that case, procedures which are *only* called within a call-chain leading from external signals specific to saving files would become effectively unreachable. Notions such as reachability and execution paths have to be evaluated in the context of procedures registered as external signal-handlers.

Consider a scenario where an application, being upgraded, is redesigned to support two different file-handling models: one for local filesystems and one for cloud storage. Certain procedures (e.g., one to check whether the file has been modified since the time of last save) may be relevant for both scenarios; others would only be active in contexts where cloud-saving is possible (an open internet connection, say) or, respectively, local file-system access. Introducing new procedures to manage the cloud-storage case alters the applications inter-procedural "connectivity," potentially requiring analyses to be updated with respect to conditions wherein a certain procedure might be called (or might be unreachable). The fact that such information *about* applications is subject to change (insofar as applications are continually refined or redesigned) indicates that information *about* application code should be managed in a systematic fashion. This might be done through VM representations directly, or at least source code and/or documentation *available* to VM compilers can draw meta-data from such sources. In other words, we can assume that applications are engineered in coding environments where information (including, for example, which procedures play the role of external signal-handlers) is detailed with enough rigor to be read by VM compilers and/or runtimes. For example, VM code could then internally incorporate representations related to reactive control flow and GUI objects which iconify pragmas for initiating application actions from users' points of view.

Analogous to "co-constructor channels" as special-purpose sites notating the origination of typed *values*, a channel-based VM could similarly support special-purpose *input* channels which carry external-signal data (we might call these "reactive" channels). In the same way that "co-constructors" are declared by providing co-constructor channels, external-signal handlers could then be identified through the presence of reactive channels. This makes it easy to identify all execution points where external signals could trigger procedure-chains: simply observe for whenever a reactive channel is "opened" during the course of building a channelized neighborhood.

Functional-Reactive Programming (FRP) has engendered various strategies and analyses for extending type systems to incorporate signal/slot signature (how should signals a slots be assigned functional types, because they are different from ordinary procedures)? Channel systems offer one solution to this problem, because channel-semantics and carrier-state are available as modeling parameters orthogonal to types themselves. For instance, slot-procedures and the signals they specifically respond to would not need to be notated with special type variants (like "World" environments or "discrete time" or other constructions endemic to FRP); instead, we could stipulate that idiosyncratic *channel kinds* designate specific procedures as signals or slots; carriers *in* those channels would hold normal typed values.

External signals (as initiators of procedure-chains that disrupt applications' passive "event-loop" states) offer one example of how reactive programming and VM design intersect, but there are related scenarios that could also be mentioned. For example, applications' GUIs are typically seen as a 2D visual extent populated with viewable objects that are simultaneously spaces to represent pieces of information to users (e.g., text, via readable characters; or numbers, via printed characters or indicators like dials and sliders; or graphics, via image or 3D displays) and origination-points for user gestures (e.g., scrolling on a slider to increase/decrease a value). Our above comments touched on the latter capabilities, but interpreting user gestures depends on the information currently presented through the relevant GUI control. As such, it is useful to track systematically how GUIs are populated with data. A useful maxim is that each class representing distinct GUI controls should be paired with a separate class representing the data which is visible within such controls. This is straightforward if a control (or in general a GUI "gadget") indicates one simple quantity (consider a slider that adjusts the zoom-level for viewing an image), but even more complex GUI areas whose display is spread over multiple subcontrols can be associated with a multi-field datatype (these sorts

of correlations are analyzed further in Chapter 24). Two datatypes are then closely interconnected: one represents some data-aggregate from a computational perspective (ensuring that all fields are properly initialized, packaging the data for persistence or serialization, and so forth) while the other translates the same information into visual indicators for user interaction. To the degree that GUI and application-level datatypes are closely aligned, VM code can be annotated to mark and leverage such alignment.

In general — continuing these GUI-related examples — interconnections between GUI components, user actions, and the datatypes shown and affected by either tend to appear in multiple contexts. Consider (as above) an image zoom level, indicated (and adjusted) by a slider-control. It is not uncommon for zooming (in and out) also to be initiated by small arrows adjacent to a slider, or by context-menu options on image-displays themselves, or by keystroke sequences like control-plus either plus or minus (holding the "control" key and tapping the "plus" or "minus" keys to increase or decrease zoom). Presumably, altering zoom levels via these other gestures should cause the zoom-slider to be adjusted proportionately. There are, then, potentially five different gestures which might affect zoom levels, each associated with GUI controls in different ways: directly interacting with either a slider or arrow buttons, or via a context menu (within an image-display), plus via keyboard actions. Such interrelationships should be tracked with some degree of rigor to maintain a consistent "User Experience."

Continuing this example, a common pattern in GUI programming is top implement "tool tips," or floating text blurbs that appear when users hover over GUI controls, explaining the purpose and pragmatics associated with the control itself. Insofar as multiple controls can be used for image-zoom, each should be provided tool-tip text accordingly; it would be useful to confirm that policies along these links are sustained in an application code-base, ideally via static code-analysis. Important gesture/feature connections need be programmed in multiple contexts, apart from the underlying signal-handlers themselves, including tool-tips, help-menu information, documentation, application history and undo/redo capabilities, notifications (e.g., small labels sometimes displayed near the bottom of application windows clarifying recent actions, such as a string confirming that an image was zoom to a particular percentage) and unit or integration testing.

Or, consider again issues with application upgrades. In the example we suggested vis-à-vis cloud versus filesystem saves, there may be multiple controls and data types associated with the cloud functionality, including windows where users register credentials to access a cloud service and strings giving a remote path for files on cloud servers — each of the GUI elements presenting the corresponding strings (path names, users names, passwords, etc.) are logically interconnected by their common utility in the guise of cloud file backup. Meanwhile, the "image zoom" case can likewise be extended to hypothetical "upgrade" examples: consider the fact that some image displays use modified *mouse gestures* for zooms, such as cursor up/down with a key pressed, often the shift key (these pragmas are borrowed from 3D displays, which try to fit the 6 or 12 degrees of freedom in 3D graphics to conventional mouse and keyboard gestures — rotations; zoom; and translations, i.e. moving parallel to x, y, or z axes; each of which can occur within the model or within the "camera" — using the keyboard to compensate for mouse-move gestures have only *two* degrees of freedom). Consider an application which decides to support this latter 3D-style zoom gesture added on to prior functionality; this decision would propagate to concerns such as those itemized at the end of last paragraph (test suites, documentation, help menus, undo/redo, etc.). Applications which tend to be intuitive and responsive from a User Experience point of view — where it is easy for users to understand how to initiate their desired actions by interacting with the software, and to learn the requisite steps when they do not know the pragmas ahead to time — consistently model the full network of interconnections between application-level capabilities, GUI display elements, and user pragmatics.

It is also worth pointing out that such goals overlap with database engineering. Controls to set zoom levels may be factored in to a database profile insofar as the optimal (or most recent) zoom level for viewing an image might be stored as one metadata-point in an image database. Similarly, a file's cloud-hosted save-path would be a relevant piece of information to track in a database for which that file is an external resource-object. Cyber-Physical networks belong in the discussion as well: CPS devices, for example, tend to have quantitative profiles (data ranges and measurement units) which would be relevant both for database persistence and for GUI admin controls. When a GUI indicator models CPS sensor readings or actuator settings (a thermostat's temperature level, say) the relevant value-range and units (e.g., Fahrenheit or Celsius) should be explicated in GUI code (obviously, an indicator needs to know how each value to be displayed compares to valid minima/maxima, and should identify and clarify for the user, perhaps supporting alternation between, Fahrenheit/Celsius, or metric/imperial, and so forth).

4 Conclusion

User Experience (UX) sometimes appears to be treated as if it were a stylistic and subjective dimension to software engineering more than a technical or mathematical problem, but effective User Interface design — and its interconnections with such themes as database engineering and CPS networks — hopefully show that rigorous UX engineering requires highly structured models of software and GUI components, as well as application-level data types, with their interconnections and value-synchronizations. There are various ways to enforce disciplined engineering standards to sustain quality UX, but at least one set of tools for this purpose can derive from VM resources — whether in the form of script-like layers interposed between GUI elements and application procedures, or runtimes for unit-testing and prototyping, or representations of application code for static analysis (or some combination).

We contend that trends such as Industry 4.0 foretell an increasing convergence between technologies related to User Experience and User Interface, Cyber-Physical Systems, 3D graphics, and Requirements Engineering. The forces behind such dynamics reflect how digitization and computer analytics has the power to increasingly shape industries such as manufacturing, architecture, green tech, and scientific research. Modern graphics cards can fluidly display interactive 3D models of industrial parts (on a small scale) to building renderings and urban designs (on a large scale) and everything in between, allowing users to visualize planned or manufactured objects/spaces for design, training, or simulations. Digital "twins," moreover, serve as presentations or anchoring-points for data-aggregates specifying objects' or designs' requirements and specifications. The digital twin concept implicitly posits relationships between an object's or design's 3D spatial form and its functional purpose, material/mechanical properties, fault tolerance, energy consumptions, and similar usability and quality-assurance metrics. Stakeholders might use software both to form visual images of materials and spaces as physical environments and as functionally

organized systems, with software being able to migrate between more geometric (e.g., 3D rendering) and more data-oriented (e.g., specs-table) modes. Given these application capabilities, digital platforms can simulate and analyze physical and/or functional systems with an unprecedented degree of computational accuracy and, simultaneously, user interaction, allowing software ecosystems to be a larger presence in industrial/architectural (and etc.) planning, construction, and maintenance.

We do not sketch this account from the point of view of advocacy, but instead to point out that maximizing the benefits of existing digitization capabilities depends on software which systematically and interactive unifies visualization and data curation/analytic aspects (e.g., 3D graphics alongside specs presentations), which in turn calls for advanced database and application-development capabilities, at least if software fitting these characterizations is to be developed in a cost-effective, extensible/adaptable, and widely available manner. We have emphasized how Virtual Machine engineering can fit into this overall sea-change in application-development requirements and standards, with a more holistic attention to such details as database query-processing and GUI design than were, arguably, emphasized by earlier VM paradigms.

References

[1] David Adger, "Constructions and Grammatical Explanation". https://www.qmul.ac.uk/sllf/media/sllf-new/department-of-linguistics/26-QMOPAL-Adger.pdf

[2] Nadeem Akhtar, *et. al.*, "Hierarchical Coloured Petri-Net Based Multi-Agent System for Flood Monitoring, Prediction, and Rescue (FMPR)". https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8926354

[3] Kyoungho An, *et. al.*, "Cloud Computing for Cyber Physical Systems: Reliability and Security Challenges and Solutions". http://www.dre.vanderbilt.edu/~gokhale/WWW/papers/CC4CPS13_FT.pdf

[4] Anees Ara, *et. al.*, "A Secure Service Provisioning Framework for Cyber-Physical Cloud Computing Systems". https://arxiv.org/abs/1611.00374

[5] Stefan Berghofer and Christian Urban, "A Head-to-Head Comparison of de Bruijn Indices and Names". https://www.sciencedirect.com/science/article/pii/S1571066107002319

[6] Cem Bozşahin, "Combinatory Logic and Natural Language Parsing". https://users.metu.edu.tr/bozsahin/bozsahin-tjeecs.pdf

[7] _____, "Combinatory Linguistics". https://library.oapen.org/bitstream/handle/20.500.12657/24661/1005450.pdf?sequence=1&isAllowed=y

[8] Yi Cai, *et. al.*, "Sensor Data and Information Fusion to Construct Digital-Twins Virtual Machine Tools for Cyber-Physical Manufacturing". https://www.researchgate.net/publication/318291645_Sensor_Data_and_Information_Fusion_to_Construct_Digital-twins_Virtual_Machine_Tools_for_Cyber-physical_Manufacturing/link/596140e8aca2728c11e0b5fe/download

[9] Felice Cardone and J. Roger Hindley, "History of Lambda-Calculus and Combinatory Logic". http://www.users.waitrose.com/~hindley/SomePapers_PDFs/2006CarHin,HistlamRp.pdf

[10] Nathaniel Christen, "Hypergraph-Based Type Theory for Software Development in a Cyber-Physical Context". In *Advances in Ubiquitous Computing: Cyber-Physical Systems, Smart Cities, and Ecological Monitoring*, Amy Neustein, ed., Elsevier, 2020, Chapter 3. https://raw.githubusercontent.com/ScignScape/ntxh/ctg/documents/NathanielChristen-Hypergraph.ngml.pdf

[11] Guerric Chupin and Henrik Nilsson, "Functional Reactive Programming, Restated". https://nottingham-repository.worktribe.com/preview/4922128/ppdp2019.pdf

[12] Liron Cohen, *et. al.*, "The Effects of Effects on Constructivism". https://www.cs.bgu.ac.il/~cliron/pubs/Effects.pdf

[13] Evie Coussé, *et. al.*, "Grammaticalization Meets Construction Grammar: Opportunities, challenges and potential incompatibilities". http://eviecousse.be/publications/cousse-ea-forthc-grammaticalizationmeetscxg.pdf

[14] René David, *et. al.*, "A Direct Proof of the Confluence of Combinatory Strong Reduction". https://arxiv.org/abs/0905.2545

[15] Parastoo Delgoshaei, *et. al.*, "A Semantic Framework for Modeling and Simulation of Cyber-Physical Systems". https://user.eng.umd.edu/~austin/reports.d/IARIA2014-PD-MA-AP-Journal-Paper.pdf

[16] Jean-Pierre Desclés, "Reasoning in Natural Language in Using Combinatory Logic and Topology An Example with Aspect and Temporal Relations". https://www.aaai.org/ocs/index.php/FLAIRS/2010/paper/viewFile/1350/1736

[17] Pietro Di Gianantonio and Furio Honsell, "An Abstract Notion of Application". https://users.dimi.uniud.it/~pietro.digianantonio/papers/copy_pdf/ana.pdf

[18] Alessandra Di Pierro, *et. al.*, "On Reversible Combinatory Logic". https://www.sciencedirect.com/science/article/pii/S1571066106000843

[19] Peter Ford Dominey, *et. al.*, "Dynamic Construction Grammar and Steps Towards the Narrative Construction of Meaning". https://users.cs.fiu.edu/~markaf/doc/w14.dominey.2017.procccgnlu.1.163_archival.pdf

[20] Manfred Droste and Heiko Vogler, "The Chomsky-Schützenberger Theorem for Quantitative Context-Free Languages". https://arxiv.org/abs/1208.3942

[21] Bernd Finkbeiner, *et. al.*, "Synthesizing Functional Reactive Programs". https://arxiv.org/pdf/1905.09825.pdf

[22] Jean H. Gallier, "Realizability, Covers, and Sheaves: Application to the Simply-Typed Lambda-Calculus". https://repository.upenn.edu/cgi/viewcontent.cgi?article=1280&context=cis_reports

[23] Silvia Ghilezan and Simona Kašterović, "Semantics for Combinatory Logic With Intersection Types". https://www.frontiersin.org/articles/10.3389/fcomp.2022.792570/full

[24] Robert Gold, "Petri Nets in Software Engineering". https://www.thi.de/fileadmin/daten/Working_Papers/thi_workingpaper_05_gold.pdf

[25] Masahito Hasegawa, "The Internal Operads of Combinatory Algebras". https://www.kurims.kyoto-u.ac.jp/~hassei/papers/2022_mfps2022_pre.pdf

[26] Martin Hilpert, "Construction Grammar and its Application to English". http://members.unine.ch/martin.hilpert/CxG%20sample.pdf

[27] J. Roger Hindley, "Axioms for Strong Reduction in Combinatory Logic". https://www.cambridge.org/core/journals/journal-of-symbolic-logic/article/abs/axioms-for-strong-reduction-in-combinatory-logic/9B7CBE781BAE43FF5148ABE720D4517A

[28] J. Roger Hindley and Jonathan P. Seldin, "Lambda-Calculus and Combinators, an Introduction". Cambridge University Press, 2008. https://www.cin.ufpe.br/~djo/files/Lambda-Calculus%20and%20Combinators.pdf

[29] Furio Honsell and Donald Sannella, "Pre-logical Relations". https://homepages.inf.ed.ac.uk/dts/pub/prelogrel-report.pdf

[30] Paul Hudak, *et al.*, "Arrows, Robots, and Functional Reactive Programming". https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=e523884ac0c118b6e42261c6c1eaaa8fde7d57f7

[31] Mans Hulden, "Parsing CFGs and PCFGs with a Chomsky-Schützenberger representation". https://dingo.sbs.arizona.edu/~mhulden/hulden_ltc_2009.pdf

[32] Adam Husted Kjelstrøm and Andreas Pavolgiannis, "The Decidability and Complexity of Interleaved Bidirected Dyck Reachability". https://cs.au.dk/~pavlogiannis/publications/papers/popl22.pdf

[33] Petr Janda, *et. al.*, "Implementation of the Digital Twin Methodology". https://www.daaam.info/Downloads/Pdfs/proceedings/proceedings_2019/072.pdf

[34] 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

[35] Konstantinos Kogkalidis and Orestis Melkonian, "D^3 as a 2-MCFL". https://omelkonian.github.io/data/publications/d3.pdf

[36] 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

[37] Yuanbo Li, *et. al.*, "Fast Graph Simplification for Interleaved Dyck-Reachability". https://research.cs.wisc.edu/wpis/papers/pldi20-graph-simplification.pdf

[38] 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

[39] Samuele Maschio, "On Posetal and Complete Partial Applicative Structures". https://arxiv.org/pdf/2211.11326.pdf

[40] Paul-André Melliès and Noam Zeilberger, "Parsing as a Lifting Problem and the Chomsky-Schützenberger Representation Theorem". https://hal.archives-ouvertes.fr/hal-03702762/document

[41] Laura A Michaelis, "Construction Grammar". https://spot.colorado.edu/~michaeli/documents/Michaelis_CG_ELL_offprint.pdf

[42] John C. Mitchell and Eugenio Moggi, "Kripke-Style Models for Typed Lambda Calculus". https://person.dibris.unige.it/moggi-eugenio/ftp/kripke.pdf

[43] Michael Moortgat, "A Note on Multidimensional Dyck Languages". https://link.springer.com/chapter/10.1007/978-3-642-54789-8_16

[44] Tiago Mück, *et. al.*, "CHIPS-AHOy: A Predictable Holistic Cyber-Physical Hypervisor for MPSoCs". https://par.nsf.gov/servlets/purl/10119096

[45] Amy Neustein and Nathaniel Christen, *Covid, Cancer, and Cardiac Care*. Elsevier, 2022.

[46] Sergey P. Orlov, *et. al.*, "Application of Hierarchical Colored Petri Nets for Technological Facilities' Maintenance Process Evaluation". https://www.mdpi.com/2076-3417/11/11/5100

[47] Ivan Perez and Henrik Nilsson, "Bridging the GUI Gap with Reactive Values and Relations". https://ivanperez.io/papers/2015-HaskellSymposium-Perez-Nilsson-BridgingGUIGapReactiveValues.pdf

[48] Robert G. Pettit IV and Hassan Gomaa, "Modeling State-Dependent Objects using Colored Petri Nets". http://csis.pace.edu/~marchese/CS865/Papers/pettit-final.pdf

[49] Steven T. Piantadosi, "The Computational Origin of Representation". https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8300595/

[50] Giulia Rambelli, *et. al.*, "Distributional Semantics Meets Construction Grammar: Towards a Unified Usage-Based Model of Grammar and Meaning". https://hal.archives-ouvertes.fr/hal-02146537/document

[51] Ivan A. Sag, "Sign-Based Construction Grammar: An Informal Synopsis". https://web.stanford.edu/group/cslipublications/cslipublications/pdf/BoasSag-sag-chapter.pdf

[52] Borja Bordel Sánchez, *et. al.*, "Enhancing Process Control in Industry 4.0 Scenarios using Cyber-Physical Systems". http://isyou.info/jowua/papers/jowua-v7n4-3.pdf

[53] Anil Sawhney, "Petri Net Based Simulation of Construction Schedules ". https://dl.acm.org/doi/pdf/10.1145/268437.268747

[54] Christopher Schuster and Cormac Flanagan, "Reactive Programming with Reactive Variables". https://users.soe.ucsc.edu/~cormac/papers/16crow.pdf

[55] Vivek Kumar Sehga, *et. al.*, "A Comparative Study of Cyber Physical Cloud, Cloud of Sensors and Internet of Things: Their Ideology, Similarities and Differences". https://ieeexplore.ieee.org/document/6779411?arnumber=6779411

[56] Jonathan P. Seldin, "The Search for a Reduction in Combinatory Logic Equivalent to $\lambda\beta$-reduction". https://core.ac.uk/download/pdf/82407499.pdf

[57] Sebastian Shaumyan, "A Semiotic Theory of Language". https://publish.iupress.indiana.edu/projects/a-semiotic-theory-of-language

[58] Murray Sinclair, *et. al.*, "The Identification of Knowledge Gaps in the Technologies of Cyber-Physical Systems with Recommendations for Closing These Gaps". https://incose.onlinelibrary.wiley.com/doi/10.1002/sys.21464

[59] Mark Steedman, *et. al.*, "Plans, Affordances, and Combinatory Grammar". https://homepages.inf.ed.ac.uk/steedman/papers/affordances/pelletier.pdf

[60] Kohei Suzuki, *et. al.*, "CFRP: A Functional Reactive Programming Language for Small-Scale Embedded Systems". https://www.worldscientific.com/doi/pdf/10.1142/9789813234079_0001

[61] Szymon Szomiński, *et. al.*, "Development of a Cyber-Physical System for Mobile Robot Control using Erlang". https://annals-csis.org/Volume_1/pliks/246.pdf

[62] Sam Van den Vonder, *et. al.*, "Tackling the Awkward Squad for Reactive Programming: The Actor-Reactor Model". http://soft.vub.ac.be/~svdvonde/papers/ecoop2020-tackling-the-awkward-squad-the-actor-reactor-model.pdf

[63] Silvio Valentinia and Matteo Vialeb, "A Binary Modal Logic for the Intersection Types of Lambda-Calculus". https://www.sciencedirect.com/science/article/pii/S0890540103000890

[64] Alexander Vodyaho, *et. al.*, "Model Based Approach to Cyber{Physical Systems Status Monitoring". https://www.mdpi.com/2073-431X/9/2/47/htm

[65] Richard West and Gabriel Parmer, "A Software Architecture for Next-Generation Cyber-Physical Systems". https://www.cs.bu.edu/~richwest/papers/west_cps.pdf

[66] Ryo Yoshinaka, *et. al.*, "Chomsky-Schützenberger-Type Characterization of Multiple Context-Free Languages". http://isw3.naist.jp/IS/TechReport/report/2010003.pdf

[67] Yu Zhang, *et. al.*, "High Fidelity Virtualization of Cyber-Physical Systems". http://web.cecs.pdx.edu/~xie/pubs/IJMSSC13.pdf