Hypergraph-Based Type Theory for Specifications-Conformant Code and Generalized Lambda Calculus: A case study in safety protocols for biomedical devices

Nathaniel Christen

August 17, 2019

Abstract

Most CyberPhysical Systems are connected to a software hub which takes responsibility for monitoring, validating, and documenting the state of the system's networked devices. Developing robust, user-friedly central software is an essential project in any CyberPhysical Systems deployment. In this chapter, I will refer to systems' central software as their "software hub". Implementing software hubs introduces technical challenges which are distinct from manufacturing CyberPhysical devices themselves — in particular, devices are usually narrowly focused on a particular kind of data and measurement, while software hubs are multi-purpose applications that need to understand and integrate data from a variety of different kinds of devices. CyberPhysical software hubs also present technical challenges that are different from other kinds of software applications, even if these hubs are one specialized domain in the larger class of user-focused software applications.

Any software application provides human users with tools to interactively and visually access data and computer files, either locally (data encoded on the "host" computer running the software) or remotely (data accessed over a network). Computer programs can be generally classified as *applications* (which are designed with a priority to User Experience) and *background processes* (which often start and maintain their state automatically and have little input or visibility to human users, except for special troubleshooting circumstances). Applications, in turn, can be generally classified as "web applications" (where users usually see one resource at a time, such as a web page displaying some collection of data, and where data is usually stored on remote servers) and "native applications" (which typically provide multiple windows and Graphical User Interface components, and which often work with data and files saved locally — i.e., saved on the filesystem of the host computer). Contemporary sofware design also recognizes "hybrid" applications which combine features of web and of native (desktop) software.

Within this taxonomy, the typical CyberPhysical software hub should be classified as a native, desktop-style application, representing the state of networked devices through special-purpose Graphical User Interface (GUI) components. Networked CyberPhysical devices are not necessarily connected to the Internet, or communicate via Internet protocols. In many cases, software hubs will access device data through securitized, closed-circuit mechanisms which (barring malicious intrusion) ensure that only the hub application can read or alter devices' state. Accordingly, an application reading device data is fundamentally different than a web application obtaining information from an Internet server.[1] CyberPhysical networks are designed to prioritize real-time connections between device and software points, and minimize network latency. Ideally, human monitors should be able (via the centralized software) to alter device state almost instantaneously. Moreover, in contrast to Internet communications with the

[1] It may be appropriate for some device data — either in real time or retroactively — to be shared with the public via Internet connections, but this is an additional feature complementing the monitoring software's primary oversight roles.

TCP protocol, data is canonically shared between devices and software hubs in complete units — rather than in packets which the software needs to reassemble. These properties of CyberPhysical networks imply that software design practices for monitoring CyberPhysical Systems are technically different than requirements for web-based components, such as HTTP servers.

At the same time, we can assume that an increasing quantity of CyberPhysical data *will* be shared via the World Wide Web. This reflects a confluence of societal and technological forces: public demand is increasing for access both to conventional medical information and to real-time health-related data (often via "wearable" sensors and other technologies that, when properly deployed, can promote health-conscious lifestyles). Similarly, the public demands greater transparency for civic and environmental data, and science is learning how to use CyberPhysical technology to track ecological conditions and urban infrastructure — analysis of traffic patterns, for instance, can help civic planners optimize public transit routes (which benefit both the public and the environment).

Meanwhile, parralel to the rise of accessible health or civic data, companies are bringing to market an increasing array of software products and "apps" which access and leverage this data. These applications do not necessarily fit the profile of "hub software". Nevertheless, it is still useful to focus attention on the design and securitazation of hub software, because hub-software methodology can provide a foundation for the design of other styles of application that access CyberPhysical data. Over time, we may realize that relatively "light-weight" portals like web sites and phone apps are suboptimal for interfacing with CyberPhysical networks — too vulnerable and/or too limited in User Interface features. In that scenario, software used by the general public may adopted many of the practices and implementations of mainframe hub applications.

As I argued, software hubs have different design principles than web or phone apps. Because they deal with raw device data (and not, for example, primarily with local filesystem files), software hubs also have different requirements than conventional desktop applications. As CyberPhysical Systems become an increasingly significant part of our Information Technology ecosystem, it will be necessary for engineers to developed rigorous models and design workflows modeled expressly around the unique challenges and niche specific to CyberPhysical software hubs.

Hubs have at least three key responsibilities:

1. To present device and system data for human users, in graphical, interactive formats suitable for humans to oversee the system and intervene as needed.

2. To validate device and system data ensuring that the system is behaving correctly and predictably.

3. To log data (in whole or in part) for subsequent analysis and maintenance.

Prior to each of those capabilities is of course receiving data from devices and pooling disparate data profiles into a central runtime-picture of device and system state. It may be, however, that direct device connection is proper not to the software hub itself but to drivers and background processes that are computationally distinct from the main application. Therefore, a theoretical model of hub software design should assume that there is an intermediate layer of background processes separating the central application from the actual physical devices. Engineers can assume that these background processes communicate information about device state either by exposing certain functions which the central application can call (analogous to system kernel functions) or by sending signals to the central application when devices' state changes. I will discuss these architectural stipulations more rigorously later in this chapter.

Once software receives device data, it needs to marshal this information between different formats, exposing the data in the different contexts of GUI components, database storage, and analytic review. Consider the example of a temperature reading, with GPS device location and timestamp data (therefore a four-part structure giving temperature at one place and time). The software needs, in a typical scenario, to do several things with this information: it has to check the data to ensure it fits within expected ranges (because malformed data can indicate physical malfunction in the devices or the network). It may need to show the temperature reading to a human user via some visual or textual indicator. And it may need to store the reading in a database for future study or troubleshooting. In these tasks, the original four-part data structure is transformed into new structures which are suitable for verification-analytics, GUI programming, and database persistence, respectively.

The more rigorously that engineers understand and document the morphology of information across these different software roles, the more clearly we can define protocols

for software design and user expectations. Careful design requires answering many technical questions: how should the application respond if it encounters unexpected data? How, in the presence of erroneous data, can we distinguish device malfunction from coding error? How should application users and/or support staff be notified of errors? What is the optimal Interface Design for users to identify anomalies, or identify situations needing human intervention, and then be able to perform the necessary actions via the software? What kind of database should hold system data retroactively, and what kind of queries or analyses should engineers be able to perform so as to study system data, to access the system's past states and performance?

I believe that the software development community has neglected to consider general models of CyberPhysical sofware which could answer these kinds of questions in a rigorous, theoretically informed manner. There is of course a robust field of cybersecurity and code-safety, which establishes Best Practices for different kinds of computing projects. Certainly this established knowledge can and does influence the implementation of software connected to CyberPhysical systems no less than any other kind of software. But models of programming Best Practices are often associated with specific coding paradigms, and therefore reflect implementations' programming environment more than they reflect the empirical domain targeted by a particular software project.

For example, Object-Oriented Programming, Functional Programming, and Web-Based Programming present different capabilities and vulerabilities and therefore each have their own "Best Practices". As a result, our understanding of how to deploy robust, well-documented and well-tested sofware tends to be decentralized among distinct programming styles and development environments. External analysis of a code base — e.g., searching for security vulnerabilities (attack routes for malicious code) — are then separate disciplines with their own methods and paradigms. Such dissipated wisdom is unfortunate if we aspire to develop integrated, broadly-applicable models of CyberPhysical safety and optimal application design, models which transcend paradigmatic differences between coding styles and roles (treating implementation, testing, and code review as distinct technical roles, for instance).

It is also helpful to distinguish cyber *security* from *safety*. When these concepts are separted, *security* generally refers to preventing *deliberate*, *malicious* intrusion into CyberPhysical networks. Cyber *safety* refers to preventing unintended or dangerous system behavior due to innocent human error, physical malfunction, or incorrect programming. Malicious attacks — in particular the risks of "cyber warfare" — are prominent in the public imagination, but innocent coding errors or design flaws are equally dangerous. Incorrect data readings, for example, led to recent Boeing 737 MAX jet accidents causing over 200 fatalities (plus the worldwide grouding of that airplane model and billions of dollars in losses for the company). Software failures either in runtime maintenance or anticipatory risk-assessment have been identified as contributing factors to high-profile accidents like Chernobyl [15] and the Fukushima nuclear reactor meltdown [27]. A less tragic but noteworthy case was the 1999 crash of NASA's US $125 million Mars Climate Orbiter. This crash was caused by software malfunctions which in turn were caused by two different software components producing incompatible data — in particular, using incompatible scales of measurement (resulting in an unanticipated mixture of imperial and metric units). In general, it is reasonable to assume that coding errors are among the deadliest and costliest sources of man-made injury and property damage.

Given the risks of undetected data corruption, seemingly mundane questions about how CyberPhysical applications verify data — and respond to apparent anomalies — become essential aspects of planning and development. Consider even a simple data aggregate like blood pressure (combining systolic and diastolic measurements). Empirically, systolic pressure is always greater than diastolic. Software systems need to agree on a protocol for encoding the number to ensure that they are in the correct order, and that they represent biologically plausible measurements. How should a particular software component test that received blood pressure data is accurate? Should it always test that the systolic quantity is indeed greater than the diastolic, and that both numbers fall in medically possible ranges? How should the component report data which fails this test? If such data checking is not performed — on the premise that the data will be proofed elsewhere — then how can this assumption be justified? How can engineers identify, in a large and complex software system, all the points where data is subject to validation tests; and then by modeling the overall system in term of these check-points ensure that all needed verifications are performed at least one time? To take the blood-pressure example, how would a software procedure that *does* check the integrity of the systolic/diastolic pair indicate for the overall system model that it performs that particular verification? Conversely, how would a procedure which does *not* perform that verification indi-

cate that this verification must be performed elsewhere in the system to guarantee that the procedure's assumptions are satisfied?

These questions are important not only for objective, measurable assessments of software quality, but also for people's more subjective trust in the reliability of software systems. In the modern world we allow sofware to be a determining factor, in places where malfunction can be fatal — airplanes, hospitals, electricity grids, trains carrying toxic chemicals, highways and city streets, etc. Consider the model of "Ubiquitous Computing" pertinent to the book series to which this volume (and hence this chapter) belongs. As explained in the series introduction:

> U-healthcare systems ... will allow physicians to remotely diagnose, access, and monitor critical patient's symptoms and will enable real time communication with patients. [This] series will contain systems based on the four future ubiquitous sensing for healthcare (USH) principles, namely i) proactiveness, where healthcare data transmission to healthcare providers has to be done proactively to enable necessary interventions, ii) transparency, where the healthcare monitoring system design should transparent, iii) awareness, where monitors and devices should be tuned to the context of the wearer, and iv) trustworthiness, where the personal health data transmission over a wireless medium requires security, control and authorize access.[2]

Observe that in this scenario, patients will have to place a level of trust in Ubiquitous Health technology comparable to the trust that they place in human doctors and other health professionals.

All of this should cause software engineers and developers to take notice. Modern society places trust in doctors for well-rehearsed and legally scrutinized reasons: physicians need to rigorously prove their competence before being allowed to practice medicine, and this right can be revoked due to malpractice. Treatment and diagnostic clinics need to be licenced, and pharmaceuticals (as well as medical equipment) are subject to rigorous testing and scientific investigation before being marketable. Notwithstanding "free market" ideologies, governments are aggressively involved in regulating medical practices; commercial practices (like marketing) are constrained, and operational transparency (like reporting adverse outcomes) is mandated, more so than in most other sectors of the economy. This level of oversight *causes* the public to trust that clinicians' recommendations are usually correct, or that medicines are usually beneficial more than harmful.

The problem, as software becomes an increasingly central feature of the biomedical ecosystem, is that no commensurate oversight framework exists in the sofware world. Biomedical IT regulations tend to be ad-hoc and narrowly domain-focused. For example, code bases in the United States which manage HL-7 data (the current federal Electronic Medical Record format) must meet certain requirements, but there is no comparable framework for software targeting other kinds of health-care information. This is not only — or not primarily — an issue of lax government oversight. The deeper problem is that we do not have a clear picture, in the framework of computer programming and software development, of what a robust regulatory framework would look like: what kind of questions it would ask; what steps a company could follow to demonstrate regulatory compliance; what indicators the public should consult to check that any software that could affect their medical outcomes is properly vetted. And, outside the medical arena, similar comments could be made regarding software in CyberPhysical settings like transportation, energy (power generation and electrical grids), physical infrastructure, environmental protections, government and civic data, and so forth — settings where sofware errors threaten personal and/or property damages.

In the case of personal medical data, as one example, there is general aggreement that data should be accessed when it is medically necessary — say, in an emergency room — but that each patient should mostly control how and whether their data is used. When data is pooled for epidemiological or meta-analytic studies, we generally believe that such information should be anomalized so that socioeconomic or "cohort" data is considered, whereas unique "personal" data remains hidden. These seem like common-sense requirements. However, they rely on concepts which we may intuitively understand, but whose precise definitions are elusive or controversial. What exactly does it mean to distinguish uniquely *personal* data, that is indelibly fixed to one person and therefore particularly sensitive as a matter of due privacy, from *demographic* data which is also personal but which, tieing patients to a cohort of their peers, is of potential public interest insofar as race, gender, and other social qualities can sometimes be

[2] https://sites.google.com/view/series-title-ausah/home?authuser=0

statistically significant? How do privacy rights intersect with the legitimate desire to identify all scientific factors that can affect epidemiological trends or treatment outcomes? More deeply, how should we actually demarcate *demographic* from *personal* data? What details indicate that some part of some data structure is one or the other?

More fundamentally, what exactly is data sharing? What are the technical situations such that certain software operations are to be *sharing* data in a fashion that triggers concerns about privacy and patient oversight? Although again we may intuitively picture what "data sharing" entails, producing a rigorous definition is surprisingly difficult.

In short, the public has a relatively inchoate idea of issues related to cyber safety, security, and privacy: we (collectively) have an informal impression that current technology is failing to meet the public's desired standards, but there is no clear picture of what IT engineers can or should do to improve the technology going forward. Needless to say, software should prevent industrial catastrophes, and private financial data should not be stolen by crime syndicates. But, beyond these obvious points, it is not clearly defined how the public's expectations for safer and more secure technology translates to low-level programming practices. How should developers earn public trust, and when is that trust deserved? Maxims like "try to avoid catastrophic failure" are too self-evident to be useful. We need more technical structures to identify which coding practices are explicitly recommended, in the context of a dynamic where engineers need to earn the public trust, but also need to define the parameters for where this trust is warranted. Without software safety models rooted in low-level computer code, software safety can only be ex-post-facto engineered, imposing requirements relatively late in the development cycle and checking code externally, via code review and analysis methods that are sepated from the core development process. While such secondary checking is important, it cannot replace software built with an eye to safety from the ground up.

This chapter, then, is written from the viewpoint that cyber safety practices have not been clearly articulated at the level of software implementation itself, separate and apart from institutional or governmental oversight. Regulatory oversight is only effective in proportion to scientific clarity vis-à-vis desired outcomes and how technology promotes them. Drugs and treatment protocols, for instance, can be evaluated through "gold standard" double-blind clinical trials — alongside statistical models, like "five-sigma" criteria, which measure scientists' confidence that trial results are truly predictive, rather than results of random chance. This package of scientific methodology provides a framework which can then be adopted in legal or legislative contexts. Continuing the example, policy makers can stipulate that pharmaceuticals need to be tested in double-blind trials, with statistically verifiable positive results, before being approved for general-purpose clinical use. Such a well-defined policy approach *is only possible* because there are biomedical paradigms which define how treatments can be tested to maximize the chance that positive test results predict similar results for the general patient population.

Analogously, a general theory of cyber safety has to be a software-design issue before it becomes a policy or contractual issue. It is at the level of low-level software design — of actual source code in its local implementation and holistic integration — that engineers can develop technical "best practices" which then provide substance to regulative oversight. Stakeholders or governments can recommend (or require) that certain practices adopted, but only if engineers have identified practices which are believed, on firm theoretical ground, to effectuate safer, more robust software.

This chapter, then, considers code-safety from the perspective of computer code outward; it is grounded on code-writing practice and in the theoretical systems which have historically been linked to programming (like type theory and lambda calculus), yielding its scientific basis. I assume that formal safety models formulated in this low-level context can propagate to institutional and governmental stake-holders, but discussion of the legal or contractual norms that can guide software practice are outside the chapter's central scope.

In the CyberPhysical context, I assume here that the most relevant software projects are hub applications; and that the preeminent issues in cyber safety are validating data and responding safely and predictably to incorrect or malformed data. Here we run into gaps between proper safety protocols and common programming practice and programming language design. In particular, most mainstream languages have limited *language-level* support for foundational safety practices such as dimensional checking (ensuring that algorithms do not work with incommensurate measurement axes) or range checking (ensuring that inaccurate CyberPhysical data is properly identified as such — in the hopes of avoiding cases like the Boeing 737 crashes, where onboard software failed to recognize inaccurate data from angle-of-attack sensors). More robust safety models are often implicit in software

libraries, outside the core language; however, to the degree that such libraries are considered experimental, or tangential to core language features, they are not likely to "propagate" outside the narrow domain of software development proper. To put it differently, no safety model appears to have been developed in the context of any mainstream programming language far enough that the very existence of such a model provides a concrete foundation for stakeholders to define requirements that developers can then follow.

This chapter's discussion will be oriented toward the C++ programming language, which is arguably the most central point from which to consider the integration of concerns — GUI, device networking, analytics — characteristic of CyberPhysical hub software. In practice, low-level code that interfaces with devices (or their drivers) might be written in C rather than C++; likewise, there is often a role for functional programming languages — even theorem-proving systems — in mission-critical data checking and system design validation. But C++ is unique in having extensive resources traversing various programming domains, like native GUI components alongside low-level networking and logically rigorous data verification. For this reason C++ is a reasonable default language for examining how these various concerns interoperate.

In that spirit, then, the C++ core language is a good case-study in language-level cyber-safety support (and the lack thereof). There are numerous C++ libraries, mostly from scientific computing, which provide features that would be essential to a robust cyber safety model (such as bounded number types and unit-of-measurement types). If some version of these libraries were adopted into a future C++ standard (analogous to the "concepts" library, a kind of metaprogramming validator, which has been included in C++20 after many years of preparation), then C++ coders would have a canonical framework for safety-oriented programming — a specific set of data types and core libraries that could become an essential part of critical CyberPhysical components. That specific circle of libraries, along with their scientific and computational principles, would then become a "cyber safety model" available to CyberPhysical applications. Moreover, the existence of such a model might then serve as a concrete foundation for defining coding and project requirements. Stakeholders should stipulate that developers use those specific libraries intrinsic to the cyber safety model, or if this is infeasible, alternate libraries offering similar features.

Of course, the last paragraph was counterfactual — *without* such a canonical "cyber safety model", there is no firm foundation for identifying stakeholder priorities. We may have generic guidelines — try to protect against physical error; try to restrict access to private data — but we do not have a canonical model, integrated with a core language, against which compliant code can be designed. I believe this is a reasonable claim to make in the context of C++, and most or all other mainstream programming languages as well.

Having said that, we should not "blame" software language engineers for gaps in mainstream languages. It turns out that such features as dimensional-unit types and bounded numerics are surprisingly difficult to implement, particularly at the core language level where such types must seamlessly interoperate with all other language features (examine the code — or even documentation — for the `boost::units` library for a sense of the technical intricasies these implementations involve). Consequently, progress toward core-language cyber safety features will be advanced with methodological progress in software language design and engineering itself.

But this situation also implies that language designers and library developers can play a lead role in establishing a safety-oriented CyberPhysical foundation. Insofar as this foundation lies in programming languages and software engineering — in data types, procedural implementations, and code analytics — then the responsibility for developing a safety-oriented theory and practice lies with the software community, not with CyberPhysical device makers or with civic or institutional stakeholders. The core principles of a next-generation CyberPhysical architecture would then be worked out in the context of software language design and software-based data modeling. My goal in this chapter is accordingly to define what I believe are fundamental and canonical structures for theorizing data structures and the computer code which operates on them, with an eye toward cyber safety and Software Quality Assurance.

In general, software requirements can be studied either from the perspective of computer code, or from the persepctive of data models. Consider again the requirement that systolic blood pressure must always be a greater quantity than diastolic: we can define this as a precondition for any code which displays, records, or performs computations on blood pressure (e.g. comparing a patient's pressure at different times). Such code is only operationally well-defined if it is provided data conforming to the systolic-over-diastolic mandate. The code *should not* execute if this mandate fails.

Design and testing should therefore guarantee that the code *will not* execute inappropriately. Conversely, these same requirements can be expressed within a data model: a structure representing blood pressure is only well-formed if its component part (or "field") representing systolic pressure measures greater than its field representing diastolic pressure.

These perspectives are complementary: a database which tracks blood pressure should be screened to ensure that all of its data is well-formed (including systolic-over-diastolic). At the same time, an application which works with medical data should double-check data when relevant procedures are called (e.g., those working with blood pressure), particularly if the data is from uncertain provance. Data could certainly come from multiple databases, or perhaps directly from CyberPhysical devices, and developers cannot be sure that all sources check their data with sufficient rigor (moreover, in the case of CyberPhysical sensors, validation in the device itself may be impossible).

Conceptually, however, validation through data models and code requirements represent distinct methodologies with distinct theoretical backgrounds. This chapter will therefore consider both perspectives, as practically alligned but conceptually *sui generis*. I will also, however, argue that certain theoretical foundations — particularly hypergraph-based data representation, and type systems derived from that basis — serve as a unifying element. I will therefore trace a construction of *hypergraph-based* type theory across both data- and code-modeling methodologies.

1 Gatekeeper Code

There are several design principles which can help ensure safety in large-scale, native/desktop-style GUI-based applications. These include:

1. Identify operational relationships between types. Suppose $\mathcal{S}$ is a data structure modeled via type $\mathcal{T}$. This type can then be associated with a type (say, $\mathcal{T}'$) of GUI component which visually displays values of type *type-*$\mathcal{T}$. There may also be a type (say, $\mathcal{T}''$) representing *type-*$\mathcal{T}$ values in a format suitable for database persistence. Application code should explicitly indicate these sorts of inter-type relationships.

2. Identify coding assumptions which determine the validity of typed values and of function calls. For each application-specific data type, consider whether every computationally possible instance of that type is actually meaningful for the real-world domain which the type represents. For instance, a type representing blood pressure has a subset of values which are biologically meaningful — where systolic pressure is greater than diastolic and where both numbers are in a sensible range. Likewise, for every procedure defined on application-specific data types, consider whether the procedure might receive arguments that are computationally feasible but empirically nonsensical. Then, establish a protocol for acting upon erroneous data values or procedure parameters. How should the error be handled, without disrupting the overall application?

3. Identify points in the code base which represent new data being introduced into the application, or code which can materially affect the "outside world". Most of the code behind GUI software will manage data being transferred between different parts of the system, internally. However, there will be specific code sites — e.g., specific procedures — which receive new data from external sources, or respond to external signals. A simple example is, for desktop applications, the preliminary code which runs when users click a mouse button. In the CyberPhysical context, an example might be code which is activated when motion-detector sensors signal something moving in their vicinity. These are the "surface" points where data "enters the system".

   Conversely, other code points locate the software's capabilities to initiate external effects. For instance, one consequence of users clicking a mouse button might be that the on-screen cursor changes shape. Or, motion detection might trigger lights to be turned on. In these cases the software is hooked up to external devices which have tangible capabilities, such as activating a light-source or modifying the on-screen cursor. The specific code points which leverage such capabilities represent data "leaving the system".

   In general, it is important to identify points where data "enters" and "leaves" the system, and to distinguish these points from sites where data is transferred "inside" the application. This helps ensure that incoming data and external effects are properly vetted. Several mathematical frameworks have been developed which codify the intuition of software components as "systems" with external data sources and effects, extending the model of software as self-contained information spaces: notably, Functional-Reactive Programming (see e.g. [18], [19], [9]) and the

theory of Hypergraph Categories ([3], [7], [8], [14]).

Methods I propose in this chapter are applicable to each of these concerns, but for purposes of exposition I will focus on the second issue: testing type instances and procedure parameters for fine-grained specifications (more precise than strong typing alone).

Strongly-typed programming language offer some guarantees on types and procedures: a function which takes an integer will never be called on a value that is *not* an integer (e.g., the character-string "46" instead of the *number* 46). Likewise, a type where one field is an integer (representing someone's age, say), will never be instantiated with something *other than* an integer in that field. Such minimal guarantees, however, are too coarse for safety-conscious programming. Even the smallest (8-bit) unsigned integer type would permit someone's age to be 255 years, which is surely an error. So any safety-conscious code dealing with ages needs to check that the numbers fall in a range narrower than built-in types allow on their own, or to ensure that such checks are performed ahead of time.

The central technical challenge of safety-conscious coding is therefore to *extend* or *complement* each programming languages' built-in type system so as to represent more fine-grained assumptions and specifications. While individual tests may seem straightforward on a local level, a consistent data-verification architecture — how this coding dimension integrates with the totality of software features and responsibility — can be much more complicated. Developers need to consider several overarching questions, such as:

- Should data validation be included in the same procedures which operate on (validated) data, or should validation be factored into separate procedures?
- Should data validation be implemented at the type level or the procedural level? That is, should specialized data types be implemented that are guaranteed only to hold valid data? Or should procedures work with more generic data types, and perform validations on a case-by-case basis?
- How should incorrect data be handled? In CyberPhysical software, there may be no obvious way to abort an operation in the presence of corrupt data. Terminating the application may not be an option; silently canceling the desired operation or trying to substitute "correct" or "default" data may be unwise; and presenting technical error messages to human users may be confusing.

These questions do not have simple answers. As such, we should develop a rigorous theoretical framework so as to codify the various options involved — what architectural decisions can be made, and what are the strengths and weaknesses of different solutions.

This section will sketch an overview of the data-validation issues from the broader vantage of planning and stakeholder expectations, before addressing narrower programming concern in subsequent sections.

1.1 *Gatekeeper Code and Fragile Code*

I will use the term *gatekeeper code* for any code which checks programming assumptions more fine-grained than strong typing alone allows — for example, that someone's age is not reported as 255 years, or that systolic pressure is not recorded as less than diastolic. I will use the term *fragile code* for code which *makes* programming assumptions *without itself* verifying that such assumptions are obeyed. Fragile code is especially consequential when incorrect data would cause the code to fail significantly — to crash the application, enter an infinite loop, or any other nonrecoverable scenario.

Note that "fragile" is not a term of criticism — some algorithms simply work on a restricted space of values, and it is inevitable that code implementing such algorithms will only work properly when provided values with the requisite properties. It is necessary to ensure that such algorithms are *only* called with correct data. But insofar as testing of the data lies outside the algorithms themselves, the proper validation has to occur *before* the algorithms commense. In short, *fragile* and *gatekeeper* code often has to be paired off: for each segment of fragile code which *makes* assumptions, there has to be a correspondng segment of gatekeeper code which *checks* those assumptions.

In that general outline, however, there is room for a variety of coding styles and paradigms. Perhaps these can be broadly classified into three groups:

1. Combine gatekeeper and fragile code in one procedure.
2. Separate gatekeeper and fragile code into different procedures.
3. Implement narrower types so that gatekeeper code is called when types are first instantiated.

Consider a function which calculates the difference between sytolic and diastolic blood pressure, returning an unsigned integer. If this code were called with malformed data where systolic and diastolic were inverted, the difference would be a negative number, which (under binary conversion to an unsigned integer) would come out as a potentially extremely large positive number (as if the patient had blood pressure in, say, the tens-of-thousands). This nonsensical outcome indicates that the basic calculation is fragile. We then have three options: test that systolic-greater-than diastolic *within the procedure*; require that this test be performed prior to the procedure being called; or use a special data structure so that systolic-over-diastolic can be confirmed as soon as any blood-pressure value is constructed in the system.

There are strengths and weaknesses of each option. Checking parameters at the start of a procedure makes code more complex and harder to maintain, and also makes updating the code more difficult. The blood-pressure case is a simple example, but in real situations there may be more complex data-validation requirements, and separating code which *checks* data from code which *uses* data, into different procedures, may simplify subsequent code maintanence. If the *validation* code needs to be modified — and if it is factored into its its own procedure — this can be done without modifying the code which actually works on the data (reducing the risk of new coding errors). In short, factoring *gatekeeper* and *fragile* code into separate procedures exemplifies the programming principle of "separation of concerns". On the other hand, such separation creates a new problem of ensuring that the gatekeeping procedure is always called. Meanwhile, using special-purpose, narrowed data types adds comlexity to the overall software if these data types are unique to that one code base, and therefore incommensurate with data provided by external sources. In these situations the software must transform data between more generic and more specific representations before sharing it (as sender or receiver), which makes the code more complicated.

In this preliminary discussion I refrain from any concrete analysis of the coding or type-theoretic models that can shed light on these options; I merely want to identify the kinds of questions which need to be addressed in preparation for a software project, particularly in the CyberPhysical domain. Ideally, protocols for pairing up fragile and gatekeeper code should be consistent through the code base.

In the specific CyberPhysical context, gatekeeping is especially important when working with device data. Such data is almost always constrained by the physical construction of devices and the kinds of physical quantities they measure (if they are sensors) or their physical capabilities (if they are "actuators", devices that cause changes in their environments). For sensors, it is an empirical question what range of values can be expected from properly functioning devices (and therefore what validations can check that the device is working as intended). For actuators, it should be similarly understood what range of values guarantee safe, correct behavior. For any device then we can construct a *profile* — an abstract, mathematical picture of the space of "normal" values associated with proper device performance. Gatekeeping code can then ensure that data received from or sent to devices fits within the profile. Defining device profiles, and explicitly notating the corresponding gatekeeping code, should therefore be an essential pre-implementation planning step for CyberPhysical software hubs.

1.2 *Proactive Design*

I have thus far argued that applications which process CyberPhysical data need to rigorously organize their functionality around specific devices' data profiles. The functions that directly interact with devices — receiving data from and perhaps sending instructions to each one — will in many instances be "fragile" in the sense I invoke in this chapter. Each of these functions may make assumptions legislated by the relevant device's specifications, to the extent that using any function too broadly constitutes a system error. Furthermore, CyberPhysical devices that are not full-fledged computers may exhibit errors due to mechanical malfunction, hostile attacks, or one-off errors in electrical-computing operations, causing performance anomalies which look like software mistakes even if the code is entirely correct (see [5] and [20], for example). As a consequence, *error classification* is especially important — distinguishing kinds of software errors and even which problems are software errors to begin with.

To cite concrete examples, a heart-rate sensor generates continuously-sampled integer values whose understood Dimension of Measurement is in "beats per minute" and whose maximum sensible range (inclusive of both rest and exercise) corresponds roughly to the $[40-200]$ interval. Meanwhile, an accelerometer presents data as voltage changes in two or three directional axes, data which may only produce signals when a change occurs (and therefore is not continuously varying), and which is mathematically converted to yield informa-

tion about physical objects' (including a person's) movement and incline. The pairwise combination of heart-rate and acceleration data (common in wearable devices) is then a mixture of these two measurement profiles — partly continuous and partly discrete sampling, with variegated axes and inter-axial relationships.

These data profiles need to be integrated with Cyber-Physical code from a perspective that cuts across multiple dimensions of project scale and lifetime. Do we design for bi-axial or triaxial accelerometers, or both, and may this change? Is heart rate to be sampled in a context where the range considered normal is based on "resting" rate or is it expanded to factor in subjects who are exercising? These kinds of questions point to the multitude of subtle and project-specific specifications that have to be established when implementing and then deploying software systems in a domain like Ubiquitous Computing. It is unreasonable to expect that all relevant standards will be settled *a priori* by sufficiently monolithic and comprehensive data models. Instead, developers and end-users need to acquire trust in a development process which is ordered to make standardization questions become apparent and capable of being followed-up in system-wide ways.

For instance, the hypothetical questions I pondered in the last paragraph — about biaxial vs. triaxial accelerometers and about at-rest vs. exercise heart-rate ranges — would not necessarily be evident to software engineers or project architects when the system is first conceived. These are the kind of modeling questions that tend to emerge as individual functions and datatypes are implemented. For this reason, code development serves a role beyond just concretizing a system's deployment software. The code at fine-grained scales also reveals questions that need to be asked at larger scales, and then the larger answers reflected back in the fine-grained coding assumptions, plus annotations and documentation. The overall project community needs to recognize software implementation as a crucial source for insights into the specifications that have to be established to make the deployed system correct and resilient.

For these reasons, code-writing — especially at the smallest scales — should proceed via paradigms disposed to maximize the "discovery of questions" effect. Systems in operation will be more trustworthy when and insofar as their software bears witness to a project evolution that has been well-poised to unearth questions that could otherwise diminish the system's trustworthiness. Lest this seem like common sense and unworthy of being emphasized so lengthily, I'd comment that literature on Ubiquitous Sensing for Healthcare (USH), for example, appears to place much greater emphasis on Ontologies or Modeling Languages whose goal is to predetermine software design at such detail that the actual code merely enacts a preformulated schema, rather than incorporate subjects (like type Theory and Software Language Engineering) whose insights can help ensure that code development plays a more proactive role.

"Proactiveness", like transparency and trustworthiness, has been identified as a core USH principle, referring (again in the series intro, as above) to "data transmission to healthcare providers ... *to enable necessary interventions*" (my emphasis). In other words — or so this language implies, as an unstated axiom — patients need to be confident in deployed USH products to such degree that they are comfortable with clinical/logistical procedures — the functional design of medical spaces; decisions about course of treatment — being grounded in part on data generated from a USH ecosystem. This level of trust, or so I would argue, is only warranted if patients feel that the preconceived notions of a USH project have been vetted against operational reality — which can happen through the interplay between the domain experts who germinally envision a project and the programmers (software and software-language engineers) who, in the end, produce its digital substratum.

"Transparency" in this environment means that USH code needs to explicitly declare its operational assumptions, on the zoomed-in procedure-by-procedure scale, and also exhibit its Quality Assurance strategies, on the zoomed-out system-wide scale. It needs to demonstrate, for example, that the code base has sufficiently strong typing and thorough testing that devices are always matched to the proper processing and/or management functions: e.g., that there are no coding errors or version-control mismatches which might cause situations where functions are assigned to the wrong devices, or the wrong versions of correct devices. Furthermore, insofar as most USH data qualifies as patient-centered information that may be personal and sensitive, there needs to be well-structured transparency concerning how sensitive data is allowed to "leak" across the system. Because functions handling USH devices are inherently fragile, the overall system needs extensive and openly documented gatekeeping code that both validates their input/output and controls access to potentially sensitive patient data.

Fragile code is not necessarily a sign of poor design.

Sometimes implementations can be optimized for special circumstances, and optimizations are valuable and should be used wherever possible. Consider an optimized algorithm that works with two lists that must be the same size. Such an algorithm should be preferred over a less efficient one whenever possible — which is to say, whenever dealing with two lists which are indeed the same size. Suppose this algorithm is included in an open-source library intended to be shared among many different projects. The library's engineer might, quite reasonably, deliberately choose not to check that the algorithm is invoked on same-sized lists — checks that would complicate the code, and sometimes slow the algorithm unnecessarily. It is then the responsibility of code that *calls* whatever procedure implements the algorithm to ensure that it is being employed correctly — specifically, that this "client" code does *not* try to use the algorithm with *different-sized* lists. Here "fragility" is probably well-motivated: accepting that algorithms are sometimes implemented in fragile code can make the code cleaner, its intentions clearer, and permits their being optimized for speed.

The opposite of fragile code is sometimes called "robust" code. While robustness is desirable in principle, code which simplistically avoids fragility may be harder to maintain than deliberately fragile but carefully documented code. Robust code often has to check for many conditions to ensure that it is being used properly, which can make the code harder to maintain and understand. The hypothetical algorithm that I contemplated last paragraph could be made robust by *checking* (rather than just *assuming*) that it is invoked with same-sized lists. But if it has other requirements — that the lists are non-empty, and so forth — the implementation can get padded with a chain of preliminary "gatekeeper" code. In such cases the gatekeeper code may be better factored into a different procedure, or expressed as a specification which engineers must study before attempting to use the implementation itself.

Such transparent declaration of coding assumptions and specifications can inspire developers using the code to proceed attentively, which can be safer in the long run than trying to avoid fragile code through engineering alone. The takeaway is that while "robust" is contrasted with "fragile" at the smallest scales (such as a single function), the overall goal is systems and components that are robust at the largest scale — which often means accepting *locally* fragile code. Architecturally, the ideal design may combine individual, *locally fragile* units with rigorous documentation and gatekeeping. So defining and declaring specifications is an intrinsic part of implementing code bases which are both robust and maintainable.

Unfortunately, specifications are often created only as human-readable documents, which might have a semi-formal structure but are not actually machine-readable. There is then a disconnect between features *in the code itself* that promote robustness, and specifications intended for *human* readers — developers and engineers. The code-level and human-level features promoting robustness will tend to overlap partially but not completely, demanding a complex evaluation of where gatekeeping code is needed and how to double-check via unit tests and other post-implementation examinations. This is the kind of situation — an impasse, or partial but incomplete overlap, between formal and semi-formal specifications — which many programmers hope to avoid via strong type systems.

Most programming language will provide some basic (typically relatively coarse-grained) specification semantics, usually through type systems and straightforward code observations (like compiler warnings about unused or uninitialized variables). For sake of discussion, assume that all languages have distinct compile-time and run-time stages (though these may be opaque to the codewriter). We can therefore distinguish compile-time tests/errors from run-time tests and errors/exceptions. Via Software Language Engineering, we can study questions like: how should code requirements be expressed? How and to what extent should requirements be tested by the language engine itself — and beyond that how can the language help coders implement more sophisticated gatekeepers than the language natively offers? What checks can and should be compile-time or run-time? How does "gatekeeping" integrate with the overall semantics and syntax of a language?

Given the maxim that procedures should have single and narrow roles — "separation of concerns" — note that *validating* input is actually a different role than *doing* calculations. This is why procedures with fine requirements might be split into two: a gatekeeper that validates input before a fragile procedure is called, separate and apart from that procedures own implementation. A related idea is overloading fragile functions: for example, a function which takes one value can be overloaded in terms of whether the value fits in some prespecified range. These two can be combined: gatekeepers can test inputs and call one of several overloaded functions, based on which overload's specifications are satisfied by the input.

But despite their potential elegance, mainstream programming languages do not supply much language-level support for expressing groups of fine-grained functions along these lines. Advanced type-theoretic constructs — including Dependent Types, typestate, and effect-systems — model requirements with more precision than can be achieved via conventional type systems alone. Integrating these paradigms into core-language type systems permits data validation to be integrated with general-purpose type checking, without the need for static analyzers or other "third party" tools (that is, projects maintained orthogonally to the actual language engineering; i.e., to compiler and runtime implementations). Unfortunately, these advanced type systems are also more complex to implement. If software language engineers aspire to make Dependent Types and similar advanced constructs part of their core language, creating compilers and runtime engines for these languages becomes proportionately more difficult.

If these observations are correct, I maintain that it is a worthwhile endeavor to return to the theoretical drawing board, with the goal of improving programming language technology itself. Programming languages are, at one level, artificial *languages* — they allow humans to communicate algorithms and procedures to computer processors, and to one another. But programming languages are also themselves engineering artificats. It is a complex project to transform textual source-code — which is human-readable and looks a little bit like natural language — into binary instructions that computers can execute. For each language, there is a stack of tools — parsers, compilers, and/or runtime libraries — which enable source code to be executed according to the language specifications. Language design is therefore constrained by what is technically feasible for these supporting tools. Practical language design, then, is an interdisciplinary process which needs to consider both the dimension of programming languages as communicative media and as digital artifacts with their own engineering challenges and limitations.

1.3 *Core Language vs. External Tools*

Because of programming languages' engineering limitations, such as I just outlined, software projects should not necessarily rely on core-language features for responsible, safety-consious programming. Academic and experimental languages tend to have more advanced features, and to embody more cutting-edge language engineering, compared to mainstream programming languages. However, it is not always feasible or desirable to implement important software with experimental, non-mainstream languages. By their nature, such projects tend to produce code that must be understood by many different developers and must remain usable years into the future. These requirements point toward well-established, mainstream languages — and mainstream development techniques overall — as opposed to unfamiliar and experimental methodologies, even if those methodologies have potential for safer, more productive coding in the future.

In short, methodologies for safety-conscious coding can be split between those which depend on core-language features, and those which rely on external, retroactive analysis of sensitive code. On the one hand, some languages and projects prioritize specifications that are intrinsic to the language and integrate seamlessly and operationally into the language's foundational compile-and-run sequence. Improper code (relative to specifications) should not compile, or, as a last resort, should fail gracefully at run-time. Moreover, in terms of programmers' thought processes, the description of specifications should be intellectually continuous with other cognitive processes involved in composing code, such as designing types or implementing algorithms. For sake of discussion, I will call this paradigm "internalism".

The "internalist" mindset seeks to integrate data validation seamlessly with other language features. Malformed data should be flagged via similar mechanisms as code which fails to type-check; and errors should be detected as early in the development process as possible. Such a mindset is evident in passages like this (describing the Ivory programming language):

> Ivory's type system is shallowly embedded within Haskell's type system, taking advantage of the extensions provided by [the Glasgow Haskell Compiler]. Thus, well-typed Ivory programs are guaranteed to produce memory safe executables, *all without writing a stand-alone type-checker* [my emphasis]. In contrast, the Ivory syntax is *deeply* embedded within Haskell. This novel combination of shallowly-embedded types and deeply-embedded syntax permits ease of development without sacrificing the ability to develop various back-ends and verification tools [such as] a theorem-prover back-end. All these back-ends share the same AST [Abstract Syntax Tree]: Ivory verifies what it compiles. [4, p. 1].

In other words, the creators of Ivory are promoting the fact that their language buttresses via its type system — and via a mathematical precision suitable for proof engines — code guarantees that for most languages require external analysis tools.

Contrary to this "internalist" philosophy, other approaches (perhaps I can call them "externalist") favor a neater separation of specification, declaration and testing from the core language, and from basic-level coding activity. In particular — according to the "externalist" mind-set — most of the more important or complex safety-checking does not natively integrate with the underlying language, but instead requires either an external source code analyzer, or regulatory runtime libraries, or some combination of the two. Moreover, it is unrealistic to expect all programming errors to be avoided with enough proactive planning, strong typing, and safety-focused paradigms: any complex code base requires some retroactive design, some combination of unit-testing and mechanisms (including those third-party to both the language and the projects whose code is implemented in the language) for externally analyzing, observing, and higher-scale testing for the code, plus post-deployment monitoring.

As a counterpoint to the features cited as benefits to the Ivory language, which I identified as representing the "internalist" paradigm, consider Santanu Paul's Source Code Algebra (SCA) system described in [17] and [16], [23]:

> Source code Files are processed using tools such as parsers, static analyzers, etc. and the necessary information (according to the SCA data model) is stored in a repository. A user interacts with the system, in principle, through a variety of high-level languages, or by specifying SCA expressions directly. Queries are mapped to SCA expressions, the SCA optimizer tries to simplify the expressions, and finally, the SCA evaluator evaluates the expression and returns the results to the user.
>
> We expect that many source code queries will be expressed using high-level query languages or invoked through graphical user interfaces. High-level queries in the appropriate form (e.g., graphical, command-line, relational, or pattern-based) will be translated into equivalent SCA expressions. An SCA expression can then be evaluated using a standard SCA evaluator, which will serve as a common query processing engine. The analogy from relational database systems is the translation of SQL to expressions based on relational algebra. [17, p. 15]

So the *algebraic* representation of source code is favored here because it makes computer code available as a data structure that can be processed via *external* technologies, like "high-level languages", query languages, and graphical tools. The vision of an optimal development environment guiding this kind of project is opposite, or at least complementary, to a project like Ivory: the whole point of Source Code Algebra is to pull code verification — the analysis of code to build trust in its safety and robustness — *outside* the language itself and into the surrounding Development Environment ecosystem.

These philosophical differences (what I dub "internalist" vs. "externalist") are normative as well as descriptive: they influence programming language design, and how languages in turn influence coding practices. One goal of language design is to produce languages which offer rigorous guarantees — fine-tuning the languages' type system and compilation model to maximize the level of detail guaranteed for any code which type-checks and compiles. Another goal of language design is to define syntax and semantics permitting valid source code to be analyzed as a data structure in its own right. Ideally, languages can aspire to both goals. In practice, however, achieving both equally can be technically difficult. The internal representations conducive to strong type and compiler guarantees are not necessarily amenable to convenient source-level analysis, and vice-versa.

Language engineers, then, have to work with two rather different constituencies. One community of programmers tends to prefer that specification and validation be integral to/integrated with the language's type system and compile-run cycle (and standard runtime environment); whereas a different community prefers to treat code evaluation as a distinct part of the development process, something logically, operationally, and cognitively separate from hand-to-screen codewriting (and may chafe at languages restricting certain code constructs because they can theoretically produce coding errors, even when the anomalies involved are trivial enough to be tractable for even barely adequate code review). One challenge for language engineers is accordingly to serve both communities. We can, for example, aspire to implement type systems which are sufficiently expressive to model many specification, validation, and gatekeeping scenarios, while also anticipating that language code should be syntactically and semantic designed to be useful in the context of external tools (like static analyzers) and models (like Source Code Algebras and Source Code Ontologies).

The techniques I discuss here work toward these goals on two levels. First, I propose a general-purpose representation of computer code in terms of Directed Hypergraphs, sufficiently rigorous to codify a theory of "functional types" as types whose values are initialized from formal representations of source code — which is to say, in the present context, code graphs. Next, I analyze different kinds of "lambda abstraction" — the idea of converting closed expressions to open-ended formulae by asserting that some symbols are "input parameters" rather than fixed values, as in λ-Calculus — from the perspective of axioms regulating how inputs and outputs may be passed to and obtained from computational procedures. I bridge these topics — Hypergraphs and Generalized λ-Calculi — by taking abstraction as a feature of code graphs wherein some hypernodes are singled out as procedural "inputs" or "outputs". The basic form of this model — combining what are essentially two otherwise unrelated mathematical formations, Directed Hypergraphs and (typed) Lambda Calculus — is laid out in Sections §II and §III.

Following that sketch-out, I engage a more rigorous study of code-graph hypernodes as "carriers" of runtime values, some of which collectively form "channels" concerning values which vary at runtime between different executions of a function body. Carriers and channels piece together to form "Channel Complexes" that describe structures with meaning both within source code as an organized system (at "compile time" and during static code analysis) and at runtime. Channel Complexes have four different semantic interpretations, varying via the distinctions between runtime and compile-time and between *expressions* and (function) *signatures*. I use the framework of Channel Complexes to identify design patterns that achieve many goals of "expressive" type systems while being implementationally feasible given the constraints of mainstream programming languages and compilers (with an emphasis on C++).

After this mostly theoretical prelude, I conclude this chapter with a discussion of code annotation, particularly in the context of CyberPhysical Systems. Because CyberPhysical applications directly manage physical devices, it is especially important that they be vetted to ensure that they do not convey erroneous instructions to devices, do not fail in ways that leave devices uncontrolled, and do not incorrectly process the data obtained from devices. Moreover, CyberPhysical devices are intrinsically *networked*, enlarging the "surface area" for vulnerability, and often worn by people or used in a domestic setting, so they tend carry personal (e.g., location) information, making network security protocols especially important ([2], [10], [21], [22]). The dangers of coding errors and software vulnerabilities, in CyberPhysical Systems like the Internet of Things (IoT), are even more pronounced than in other application domains. While it is unfortunate if a software crash causes someone to lose data, for example, it is even more serious if a CyberPhysical "dashboard" application were to malfunction and leave physical, networked devices in a dangerous state.

To put it differently, computer code which directly interacts with CyberPhysical Systems will typically have many fragile pieces, which means that applications providing user portals to maintain and control CyberPhysical Systems need a lot of gatekeeping code. Consequently, code verification is an important part of preparing CyberPhysical Systems for deployment. The "Channelized Hypergraph" framework I develop here can be practically expressed in terms of code annotations that benefit code-validation pipelines. This use case is shown in demo code published as a data set alongside this chapter (available for download at `https://github.com/scignscape/PGVM`). These techniques are not designed to substitute for Test Suites or Test-Driven Development, though they can help to clarify the breadth of coverage of a test suite — in other words, to justify claims about tests being thorough enough that the code base passing all tests actually does argue for the code being safe and reliable. Nor are code annotations intended to automatically verify that code is safe or standards-compliant, or to substitute for more purely mathematical code analysis using proof-assistants. But the constructions presented here, I claim, can be used as part of a code-review process that will enhance stakeholders' trust in safety-critical computer code, in cost-effective, practically effective ways.

2 Directed Hypergraphs and Generalized Lambda Calculus

Thus far, I have written in general terms about architectural features related to CyberPhysical software; in particular, verifying coding assumptions concerning individual data types and/or procedures. My comments were intended to summarize the relevant territory, so that I can add some theoretical details or suggestions from this point forward. In particular, I will explore how to model software components at different scales so as to facilitate robust, safety-conscious coding practices.

Note that almost all non-trivial software is in some sense "procedural"; the total package of functionality provided by each software component is distributed among many individual procedures, which are interconnected. Each procedure, in general, implements its functionality by calling *other* procedures in some strategic order. Of course, often inter-procedure calls are *conditional* — a calling procedure will call one (or some sequence of) procedures when some condition holds, but call diffdrent procedures when some other conditions hold. In any case, computer code can be analyzed as a graph, where connections exist between procedures insofar as one procedure calls, or sometimes calls, the other.

This general picture is only of only limited applicability to actual applications, however, because the basic concept of "procedure" varies somewhat between different programming languages. As a result, it takes some effort to develop a comprehensive model of computer code which accommodates a representative spectrum of coding styles and paradigms.

There are perhaps three different perspectives which can be taken toward such a comprehensive theory. One route is to consider source code as a data structure in its own right, employing a Source Code Algebra or Source Code Ontology to assert properties of source code and enable queries against source code, qua information space. A second option derives from type theory: to consider procedures as instances of funtional types, characterized by sets of inputs and output types. A procedure is then a transform which, in the presence of (zero or more) inputs having the proper types, produces (one or more) outputs with their own types. In practice, some procedures do not return values, but they *do* have some kind of side-effect, which can be analyzed as a variety of "output". Finally, procedures can be studied via mathematical frameworks such as the Lambda Calculus, which allows notions of functions on typed parameters, and of functional application — apply functions to concrete values, which is analogous to calling procedures with concrete input arguments — to be made formally rigorous.

I will briefly consider all three of perspectives — Source Code Ontology, type-theoretic models, and Lambda Calculus — in this section. I will also propose a new model, based on the idea of "channels", which combines elements of all three.

2.1 *Generalized Lambda Calculus*

Lambda (or λ-) Calculus emerged in the early 20th century as a formal model of mathematical functions and function-application. There are many mathematical constructions which can be subsumed under the notion of "function-application", but these have myriad notations and conventions. Consider the visual differences between mathematical notations — integrals, square roots, super- and sub-scripted indices, and so forth — to the much simpler alphabets of mainstream programming languages. But the early 20th century was a time of great interest in "mathematical foundations", seeking to provide philosophical underpinnings for mathematical reasoning in general, unifying disparate mathematical methods and subdisciplines. One consequence of this foundational program was an attempt to capture the formal essence of the concept of "function" and of functions being applied to concrete values.

A related foundational concern is how mathematical formulae can be nested, yielding new formulae. For example, the volume of a sphere (expressed in terms of its radius R) is $\frac{4\pi R^3}{3}$. The symbol R is just a mnemonic which could be replaced with a different symbol, without the formula being different. But it can also be replaced by a more complex expression, to yield a new formula. In this case, substituting the formula for a cube's half-diagonal — $\sqrt{3}\sqrt[3]{V}$ where V is its volume — for R, in the first formula, yields $\frac{4}{3}\sqrt{27}\pi V$: a formula for the sphere's volume in terms of the volume of the largest cube that can fit inside it ([**?**] has similar interesting examples in the context of code optimization). This kind of tinkering with equations is of course the bread-and-butter of mathematical discovery. In terms of foundations research, though, observe that the derivation depended on two givens: that the R symbol is "free" in the first formula — it is a placeholder rather than the designation of a concrete value, like π — and that free symbols (like R) can be bound to other formulae, yielding new equations.

From cases like these — relative simple geometric expressions — mathematicians began to ask foundation questions about mathematical formulae: what are all formulae that can be built up from a set of core equations via repetedly substituting nested expressions for free symbols? This questions turns out to be related to the issue of finite calculations: in lieu of building complex formulae out of simpler parts, we can proceed in the opposite direction, replacing nested expressions with values. Formulae are constructed in terms

of unknown values; when we have concrete measurements to plug in to those formulae, the set of unkowns decreases. If *all* values are known, then a well-constructed formula will converge to a (possibly empty) set of outcomes. This is roughly analogous to a computation which terminates in real time. On the other hand, a *recursive* formula — an expression nested inside itself, such as a continued fraction — is analogous to a computation which loops indefinitely.[3]

In the early days of computer programming, it was natural to turn to λ-Calculus as a formal model of computer procedures, which are in some ways analogous to mathematical formulae. As a mathematical subject, λ-Calculus predates digital computers as we know them. While there were no digital computers at the time, there *was* a growing interest in mechanical computing devices, which led to the evolution of cryptographic machines used during the Second World War. So there was indeed a practical interest in "computing machines", which eventually led to the John von Neumann formal prototypes for digital computer.

Early on, though, λ-Calculus was less about blueprints for calculating machines and more about *abstract* formulation of calculational processes. Historically, the original purpose of λ-Calculus was largely a mathematical *simulation* of computations, which is not the same as a mathematical *prototype* for computating machines. Mathematicians in the decades before WWII investigated logical properties of computations, with particular emphasis on what sort of problems could always be solved in finite time, or what kind of procedures can be guaranteed to terminate — a "Computable Number", for example, is a number which can be approximated to any degree of precision by a terminating function. Similarly, a Computable Function is a function from input values to output values that can be associated with an always-terminating procedure which necessarily calculates the desired outputs from a set of inputs. The space of Computable Functions and Computable Numbers are mathematical objects whose properties can be studied through mathematical techniques — for instance, Computable Numbers are known to be a countable field within the real numbers. These mathematical properties are proven using a formal description of "any computer whatsoever", which has no concern for the size and physical design of the "computers" or the time required for its "programs", so long as they are finite. Computational procedures in this context are not actual implementations but rather mathematical distillations that can stand in for calculations for the purpose of mathematical analysis (interesting and representative contemporary articles continuing these perspectives include, e.g., [**?**], [**?**], [**?**]). ‘p‘ It was only after the emergence of modern digital computers that λ-Calculus become reinterpreted as a model of *concrete* computing machines. In its guise as a Computer Science (and not just Mathematical Foundations) discipline, λ-Calculus has been most influential not in its original form but in a plethora of more complex models which track the evolution of programming languages. Many programming languages have important differences which are not describable on a purely mathematical basis: two languages which are both "Turing complete" are abstractly interchangable, but it is important to reprecent the contrast between, say, Object-Oriented and Functional programming. In lieu of a straightforward, mathematical model of formulae as procedures which map inputs to outputs, modern programming languages add may new constructs which determine differet mechanisms whereby procedures can read and modify values: objects, exceptions, closures, mutable references, side-effects, signal/slot connections, and so forth. Accordingly, new programming constructions have inspired new variants of λ-Calculus, analyzing different features of modern programming languages — Object Orientation, Exceptions, call-by-name, call-by-reference, side effects, polymorphic type systems, lazy evaluation — in the hopes of deriving formal proofs of program behavior insofar as computer code uses the relevant constructions. In short, a reasonable history can say that λ-Calculus mutated from being an abstract model for studying Computability as a mathematical concept, to being a paradigm for prototype-specifications of concretely realized computing environments. ‘p‘

Modern programming languages have many different ways of handing-off values between procedures. The "inputs" to a function can be "message receivers" as in Object-Oriented programming, or lexically scoped values "captured" in an anonymous function that inherits values from the lexical scope (loosely, the area of source code) where its body is composed. Procedures can also "receive" data indirectly from pipes, streams, sockets, network connections, database connections, or files. All of these are potential "input channels" whereby a function implementation may access a value that it needs. In addition, functions can "return" values not just by providing a final result but by throwing exceptions, writing to files or pipes, and so forth. To represent these myriad "channels of communication" computer scientists have invented a menagerie of extensions to λ-Calculus — a noteworthy

[3] Although there are sometimes techniques for converting formulae like Continued Fractions into "closed form" equations which do "terminate".

example is the "Sigma" calculus to model Object-Oriented Programming; but parallel extensions represent call-by-need evaluation, exceptions, by-value and by-reference capture, etc.

Rather than study each system in isolation, in this chapter I propose an integrated strategy for unifying disparate λ-Calculus extensions into an overarching framework. The "channel-based" tactic I endorse here may not be optimal for a *mathematical* calculus which has formal axioms and provable theorems, but I believe it can be useful for the more practical goal of modeling computer code and software components, to establish recommended design patterns and document coding assumptions.

In this perspective, different extensions or variations to λ-Calculus model different *channels*, or data-sources through which procedures receive and/or modify values. Different channels have their own protocols and semantics for passing values to functions. We can generically discuss "input" and "output" channels, but programming languages have different specifications for different genres of input/output, which we can model via different channels. For a particular channel, we can recognize language-specific limitations on how values passed in to or received from those channels are used, and how the symbols carrying those values interact with other symbols both in function call-sites and bodies. For example, functions can output values by throwing exceptions, but exceptions are unusual values which have to be handled in specific ways — languages use exceptions to signal possible programming errors, and they are engineered to interrupt normal program flow until or unless exceptions are "caught".

Computer scientists have explored these more complex programming paradigms in part by inventing new variations on λ-calculi. Here I will develop one theory representing code in terms of Directed Hypergraphs, which are subject to multiple kinds of lambda abstraction — in principle, replacing many disparate λ-Calculus extensions with one overarching framework. This section will lay out the details of this form of Directed Hypergraph and how λ-calculi can be defined on its foundation. The following section will discuss an expanded type theory which follows organically from this approach, and the third section will situate lambda calculi in terms of "Channel Algebras".

Many concepts outlined here are reflected in the accompanying code set, which includes a C++ Directed Hypergraph library and also parsers and runtimes for an Interface Definition Language. The design choices behind these components will be suggested in the text, but hopefully the code will illustrate how the ideas can be manifest in concrete implementations, which in turn provide evidence that they are logically sound at least to the level of properly-behaving application code.

2.2 *Directed Hypergraphs and "Channel Abstractions"*

A *hypergraph* is a graph whose edges (a.k.a. "hyperedges") can span more than two nodes ([**?**, e.g. p. 24], [**?**], [**?**]; [**?**], [**?**]). A *directed* hypergraph ("DH") is a hypergraph where each edge has a *head set* and *tail set* (both possibly empty). Both of these are sets of nodes which (when non-empty) are called *hypernodes*. A hypernode can also be thought of as a hyperedge whose tail-set (or head-set) is empty. Note that a typical hyperedge connects two hypernodes (its head- and tail-sets), so if we consider just hypernodes, a hypergraph potentially reduces to a directed ordinary graph. While "edge" and "hyperedge" are formally equivalent, I will use the former term when attending more to the edge's representational role as linking two hypernodes, and use the latter term when focusing more on its tuple of spanned nodes irrespective of their partition into *head* and *tail*.

I assume that hyperedges always span an *ordered* node-tuple which induces an ordering in the head- and tail-sets: so a hypernode is an *ordered list* of nodes, not just a *set* of nodes. I will say that two hypernodes *overlap* if they share at least one node; they are *identical* if they share exactly the same nodes in the same order; and *disjoint* if they do not overlap at all. I call a Directed Hypergraph "reducible" if all hypernodes are either disjoint or identical. The information in reducible DHs can be factored into two "scales", one a directed graph whose nodes are the original hypernodes, and then a table of all nodes contained in each hypernode. Reducible DHs allow ordinary graph traversal algorithms when hypernodes are treated as ordinary nodes on the coarser scale (so that their internal information — their list of contained nodes — is ignored).[4]

To avoid confusion, I will hereafter use the word "hy-

[4] A weaker restriction on DH nodes is that two non-identical hypernodes *can* overlap, but must preserve node-order: i.e., if the first hypernode includes nodes N_1, and N_2 immediately after, and the second hypernode also includes N_1, then the second hypernode must also include N_2 immediately thereafter. Overlapping hypernodes can not "permute" nodes — cannot include them in different orders or in a way that "skips" nodes. Trivially, all reducible DHs meet this condition. Any graphs discussed here are assumed to meet this condition.

ponode" in place of "node", to emphasize the container/contained relation between hypernodes and hyponodes. I will use "node" as an informal word for comments applicable to both hyper- and hypo-nodes. Some Hypergraph theories and/or implementations allow hypernodes to be nested: i.e., a hypernode can contain another hypernode. In these theories, in the general case any node is potentially both a hypernode and a hyponode. For this chapter, I assume the converse: any "node" (as I am hereafter using the term) is *either* hypo- or hyper-. However, multi-scale Hypergraphs can be approximated by using hyponodes whose values are proxies to hypernodes.

Here I will focus on a class of DHs which (for reasons to emerge) I will call "Channelizable". Channelizable Hypergraphs (CHs) have these properties:

1. They have a Type System $\mathbb{T}$ and all hyponodes and hypernodes are assigned exactly one canonical type (they may also be considered instances of super- or subtypes of that type).

2. All hyponodes can have (or "express") at most one value, an instance of its canonical type, which I will call a *hypovertex*. Hypernodes, similarly, can have at most one *hypervertex*. Like "node" being an informal designation for hypo- and hyper-nodes, "vertex" will be a general term for both hypo- and hyper-vertices. Nodes which do have a vertex are called *initialized*. The hypovertices "of" a hypernode are those of its hyponodes.

3. Two hyponodes are "equatable" if they express the same value of the same type. Two (possibly non-identical) hypernodes are "equatable" if all of their hyponodes, compared one-by-one in order, are equatable. I will also say that values are "equatable" (rather than just saying "equal") to emphasize that they are the respective values of equatable nodes.

4. There may be a stronger relation, defined on equatable non-equivalent hypernodes, whereby two hypernodes are *inferentially equivalent* if any inference justified via edges incident to the first hypernode can be freely combined with inferences justified via edges incident to the second hypernode. Equatable nodes are not necessarily inferentially equivalent.

5. Hypernodes can be assumed to be unique in each graph, but it is unwarranted to assume (without type-level semantics) that two equatable hypernodes in different graphs are or are not inferentially equivalent. Conversely, even if graphs are uniquely labeled — which would appear to enable a formal distinction between hypernodes in one graph from those in another, CH semantics does not permit the assumption that this separation alone justifies inferences presupposing that their hypernodes *are not* inferentially equivalent.

6. All hypo- and hypernodes have a "proxy", meaning there is a type in $\mathbb{T}$ including, for each node, a unique identifier designating that node, that can be expressed in other hyponodes.

7. There are some types (including these proxies) which may only be expressed in hyponodes. There may be other types which may only be expressed in hypernodes. Types can then be classified as "hypotypes" and "hypertypes". The $\mathbb{T}$ may stipulate that all types are *either* hypo or hyper. In this case, it is reasonable to assume that each hypotype maps to a unique hypertype, similar to "boxing" in a language which recognizes "primitive" types (in Object-Oriented languages, boxing allows non-class-type values to be used as if they were objects).

8. Types may be subject to the restriction that any hypernode which has that type can only be a tail-set, not a head-set; call these *tail-only* types.

9. Hyponodes may not appear in the graph outside of hypernodes. However, a hypernode is permitted to contain only one hyponode.

10. Each edge, separate and apart from the CH's actual graph structure, is associated with a distinct hypernode, called its *annotation*. This annotation cannot (except via a proxy) be associated with any other hypernode (it cannot be a head- or tail-set in any hypernode). The first hyponode in its annotation I will dub a hyperedge's *classifier*. The outgoing edge-set of a hypernode can always be represented as an associative array indexed by the classifier's vertex.

11. A hypernode's type may be subject to restrictions such that there is a single number of hyponodes shared by all instances. However, other types may be expressed in hypernodes whose size may vary. In this case the hyponode types cannot be random; there must be some pattern linking the distribution of hyponode types evident in hypernodes (with the same hypernode types) of different sizes. For example, the hypernodes may be dividable into a fixed-size, possibly empty sequence of hyponodes, followed by a chain of hyponode-sequences repeating the same type pattern. The simplest manifestation of this structure is a hypernode all of whose hyponodes are the same type.

12. Call a *product-type transform* of a hypernode to be a different hypernode whose hypovertices are tuples of values equatable

to those from the first hypernode, typed in terms of product types (i.e., tuples). For example, consider two different representations of semi-transparent colors: as a 4-vector RGBT, or as an RGB three-vector paired with a transparency magnitude. The second representation is a product-type transform of the first, because the first three values are grouped into a three-valued tuple. We can assert the requirement in most contexts that CHs whose hypernodes are product-type transforms of each other contain "the same information" and as sources of information are interchangeable.

13. The Type System $\mathbb{T}$ is *channelized*, i.e., closed under a Channel Algebra, as will be discussed below.

These definitions allude to two strategies for computationally representing CHs. One, already mentioned, is to reduce them to directed graphs by treating hypernodes as integral units (ignoring their internal structure). A second is to model hypernodes as a "table of associations" whose keys are the values of the classifier hyponodes on each of their edges. A CH can also be transformed into an *undirected* hypergraph by collapsing head- and tail- sets into an overarching tuple. All of these transformations may be useful in some analytic/representational contexts, and CHs are flexible in part by morphing naturally into these various forms.

Diagram 1: "Unplugging" a Node

Notice that information present *within* a hypernode can also be expressed as relations *between* hypernodes. For example, consider the information that I (Nathaniel), age 45, live in Brooklyn as a registered Democrat. This may be represented as a hypernode with hyponodes $\langle\lceil\textbf{Nathaniel}\rceil, \lceil\textbf{45}\rceil\rangle$, connected to a hypernode with hyponodes $\langle\lceil\textbf{Brooklyn}\rceil, \lceil\textbf{Democrat}\rceil\rangle$, via a hyperedge whose classifier encodes the concept "lives in" or "is a resident of". However, it may also be encoded by "unplugging" the "age" attribute so the first hypernode becomes just $\lceil\textbf{Nathaniel}\rceil$ and it acquires a new edge, whose tail has a single hyponode $\lceil 45\rceil$ and a classifier (encoding the concept) "age" (see the comparison in Diagram 1). This construction can work in reverse: information present in a hyperedge can be refactored so that it "plugs in" to a single hypernode.

These alternatives are not redundant. Generally, representing information via hyperedges connecting two hypernodes implies that this information is somehow conceptually apart from the hypernodes themselves, whereas representing information via hyponodes *inside* hypernodes implies that this information is central and recurring (enforced by types), and that the data thereby aggregated forms a recurring logical unit. In a political survey, people's names may *always* be joined to their age, and likewise their district of residence *always* joined to their political affiliation. The left-hand side representation of the info (seen as an undirected hyperedge) $\langle\lceil\textbf{Nathaniel}\rceil, \lceil\textbf{45}\rceil, \lceil\textbf{Brooklyn}\rceil, \lceil\textbf{Democrat}\rceil\rangle$ in Diagram 1 captures this semantics better because it describes the name/age and place/party pairings as *types* which require analogous node-tuples when expressed by other hypernodes. For example, any two hypernodes with the same type as $\langle\lceil\textbf{Nathaniel}\rceil, \lceil\textbf{45}\rceil\rangle$ will necessarily have an "age" hypovertex and so can predictably be compared along this one axis. By contrast, the right-hand ("unplugged") version in Diagram 1 implies no guarantees that the "age" data point is present as part of a recurring pattern.

In general, graph representations like CH and RDF serve two goals: first, they are used to *serialize* data structures (so that they may be shared between different locations; such as, via the internet); and, second, they provide formal, machine-readable descriptions of information content, allowing for analyses and transformations, to infer new information or produce new data structures. The design and rationale of representational paradigms is influenced differently by these two goals, as I will review now with an eye in part on drawing comparisons between CH and RDF.

2.3 *Channelized Hypergraphs and RDF*

The Resource Description Framework (RDF) models information via directed graphs ([**?**], [**?**], [**?**], and [**?**] are good discussions of Semantic Web technologies from a graph-theoretic perspective), whose edges are labeled with concepts that, in well-structured contexts, are drawn from published Ontologies (these labels play a similar role to "classifiers" in CHs). In principle, all data expressed via RDF graphs is defined by unordered sets of labeled edges, also called "triples" ("$\langle$SUBJECT, PREDICATE, OBJECT$\rangle$", where the "Predicate" is the label). In practice, however, higher-level RDF notation such as TTL (TURTLE or "Terse RDF Triple Language") and Notation3 (N3) deal with aggregate groups of data, such as RDF containers and collections.

Diagram 2: CH vs. RDF Collections.

For example, imagine a representation of the fact

"(A/The person named) Nathaniel, 45, has lived in Brooklyn, Buffalo, and Montreal" (shown in Diagram 2 as both a CH and in RDF). An N3 graph of the sentence might look like this: The final (N3 proper) expression, in particular, actually seems structurally closer to the CH model than to the RDF. If we consider TURTLE or N3 as *languages* and not just *notations*, it would appear as if their semantics is built around hyper-edges rather than triples. It would seem that these languages encode many-to-many or one-to-many assertions, graphed as edges having more than one subject and/or predicate. Indeed, Tim Berners-Lee himself suggests that "Implementations may treat list as a data type rather than just a ladder of rdf:first and rdf:rest properties" [**?**, p. 6]. That is, the specification for RDF list-type data structures invites us to consider that they *may* be regarded integral units rather than just aggregates that get pulled apart in semantic interpretation.

Technically, perhaps, this is an illusion. Despite their higher-level expressiveness, RDF expression languages are, perhaps, supposed to be deemed "syntactic sugar" for a more primitive listing of triples: the *semantics* of TURTLE and N3 are conceived to be defined by translating expressions down to the triple-sets that they logically imply (see also [**?**]). This intention accepts the paradigm that providing semantics for a formal language is closely related to defining which propositions are logically entailed by its statements.

There is, however, a divergent tradition in formal semantics that is oriented to type theory more than logic. It is consistent with this alternative approach to see a different semantics for a language like TURTLE, where larger-scale aggregates become "first class" values. So, ⟨⌈**Nathaniel**⌉, ⌈**45**⌉⟩ can be seen as a (single, integral) *value* whose *type* is a ⟨**name**, **age**⟩ pair. Such a value has an "internal structure" which subsumes multiple data-points. The RDF version is organized, instead, around a *blank node* which ties together disparate data points, such as my name and age. This blank node is also connected to another blank node which ties together place and party. The blank nodes play an organizational role, since nodes are grouped together insofar as they connect to the same blank node. But the implied organization is less strictly entailed; one might assume that the ⟨⌈**Brooklyn**⌉, ⌈**Democrat**⌉⟩ nodes could just as readily be attached individually to the "name/age" blank (i.e., I live in Brooklyn, *and* I vote Democratic).

Why, that is, are Brooklyn and Democratic grouped together? What concept does this fusion model? There is a presumptive rationale for the name/age blank (i.e., the fusing name/age by joining them to a blank node rather than allowing them to take edges independently): conceivably there are multiple 45-year-olds named Nathaniel, so *that* blank node plays a key semantic role (analogous to the quantifier in "*There is* a Nathaniel, age 45..."); it provides an unambiguous nexus so that further predicates can be attached to *one specific* 45-year-old Nathaniel rather than any old ⟨⌈**Nathaniel**⌉, ⌈**45**⌉⟩. But there is no similarly suggested semantic role for the "place/party" grouping. The name cannot logically be teased apart from the name/age blank (because there are multiple Nathaniels); but there seems to be no *logical* significance to the place/party grouping. Yet pairing these values *can* be motivated by a modeling convention — reflecting that geographic and party affiliation data are grouped together in a data set or data model. The logical semantics of RDF make it harder to express these kinds of modeling assumptions that are driven by convention more than logic — an abstracting from data's modeling environment that can be desirable in some contexts but not in others.

So, why does the Semantic Web community effectively insist on a semantic interpretation of TURTLE and N3 as *just* a notational convenience for N-TRIPLES rather than as higher-level languages with a different higher-level semantics — and despite statements like the above Tim Berners-Lee quote insinuating that an alternative interpretation has been contemplated even by those at the heart of Semantic Web specifications? To the degree that this question has an answer, it probably has something to do with reasoning engines: the tools that evaluate SPARQL queries operate on a triplestore basis. So the "reductive" semantic interpretation is arguably justified via the warrant that the definitive criteria for Semantic Web representations are not their conceptual elegance vis-à-vis human judgments but their utility in cross-Ontology and cross-context inferences. As a counter-argument, however, note that many inference engines in Constraint Solving, Computer Vision, and so forth, rely on specialized algorithms and cannot be reduced to a canonical query format. Libraries such as GECODE and ITK are important because problem-solving in many domains demands fine-tuned application-level engineering. We can think of these libraries as supporting *special* or domain-specific reasoning engines, often built for specific projects, whereas OWL-based reasoners like FACT++ are *general* engines that work on general-purpose RDF data without further qualification. In order to apply "special" reasoners to RDF, a contingent of nodes must be selected which are consistent with reasoners' runtime requirements.

Of course, special reasoners cannot be expected to run on the domain of the entire Semantic Web, or even on "very large" data sets in general. A typical analysis will subdivide its problem into smaller parts that are each tractable to custom reasoners — in radiology, say, a diagnosis may proceed by first selecting a medical image series and then performing image-by-image segmentation. Applied to RDF, this two-step process can be considered a combination of general and special reasoners: a general language like SPARQL filters many nodes down to a smaller subset, which are then mapped/deserialized to domain-specific representations (including runtime memory). For example, RDF can link a patient to a diagnostic test, ordered on a particular date by a particular doctor, whose results can be obtained as a suite of images — thereby selecting the particular series relevant for a diagnostic task. General reasoners can *find* the images of interest and then pass them to special reasoners (such as segmentation algorithms) to analyze. Insofar as this architecture is in effect, Semantic Web data is a site for many kinds of reasoning engines. Some of these engines need to operate by transforming RDF data and resources to an optimized, internal representation. Moreover, the semantics of these representations will typically be closer to a high-level N3 semantics taken as *sui generis*, rather than as interpreted reductively as a notational convenience for lower-level formats like N-TRIPLE. This appears to undermine the justification for reductive semantics in terms of OWL reasoners.

Perhaps the most accurate paradigm is that Semantic Web data has two different interpretations, differing in being consistent with special and general semantics, respectively. It makes sense to label these the "special semantic interpretation" or "semantic interpretation for special-purpose reasoners" (SSI, maybe) and the "general semantic interpretation" (GSI), respectively. Both these interpretations should be deemed to have a role in the "semantics" of the Semantic Web.

Another order of considerations involve the semantics of RDF nodes and CH hypernodes particularly with respect to uniqueness. Nodes in RDF fall into three classes: blank nodes; nodes with values from a small set of basic types like strings and integers; and nodes with URLs which are understood to be unique across the entire World Wide Web. There are no blank nodes in CH; and intrinsically no URLs either, although one can certainly define a URL *type*. There is nothing in the semantics of URLs which guarantees that each URL designates a distinct internet resource; this is just a convention which essentially, *de facto*, fulfills itself because it structures a web of commercial and legal practices, not just digital ones; e.g. ownership is uniquely granted for each internet domain name. In CH, a data type may be structured to reflect institutional practices which guarantee the uniqueness of values in some context: books have unique ISBN codes; places have distinct GIS locations, etc. These uniqueness requirements, however, are not intrinsically part of CH, and need to be expressed with additional axioms. In general, a CH hypernode is a tuple of relatively simple values and any additional semantics are determined by type definitions (recall the idea that CH hypernodes are roughly analogous to C **struct**s — which have no *a priori* uniqueness mechanism).

Also, RDF types are less intrinsic to RDF semantics than in CH (see [**?**]). The foundational elements of CH are value-tuples (via nodes expressing values, whose tuples in turn are hypernodes). Tuples are indexed by position, not by labels: the tuple $\langle \lceil \textbf{Nathaniel} \rceil, \lceil \textbf{45} \rceil \rangle$ does not in itself draw in the labels "name" or "age", which instead are defined at the type-level (insofar as type-definitions may stipulate that the label "age" is an alias for the node in its second position, etc.). So there is no way to ascertain the semantic/conceptual intent of hypernodes without considering both hyponode and hypernode types. Conversely, RDF does not have actual tuples (though these can be represented as collections, if desired); and nodes are always joined to other nodes via labeled connectors — there is no direct equivalent to the basis-level CH modeling unit of a hyponode being included in a hypernode by position.

At its core, then, RDF semantics are built on the proposition that many nodes can be declared globally unique by fiat. This does not need to be true of all nodes — RDF types like integers and floats are more ethereal; the number 45 in one graph is indistinguishable from 45 in another graph. This can be formalized by saying that some nodes can be *objects* but never *subjects*. If such restrictions were not enforced, then RDF graphs could become in some sense overdetermined, implying relationships by virtue of quantitative magnitudes devoid of semantic content. This would open the door to bizarre judgments like "my age is non-prime" or "I am older than Mohamed Salah's goal totals". The way to block these inferences is to prevent nodes like "the number 45" from being subjects as well as objects. But nodes which are not primitive values — ones, say, designating Mohamed Salah himself rather than his goal totals — are justifiably globally unique, since we have compelling reasons to adopt a model where

there is exactly one thing which is *that* Mohamed Salah. So RDF semantics basically marries some primitive types which are objects but never subjects with a web of globally unique but internally unstructured values which can be either subject or object.

In CH the "primitive" types are effectively hypotypes; hyponodes are (at least indirectly) analogous to object-only RDF nodes insofar as they can only be represented via inclusion inside hypernodes. But CH hypernodes are neither (in themselves) globally unique nor lacking in internal structure. In essence, an RDF semantics based on guaranteed uniqueness for atom-like primitives is replaced by a semantics based on structured building-blocks without guaranteed uniqueness. This alternative may be considered in the context of general versus special reasoners: since general reasoners potentially take the entire Semantic Web as their domain, global uniqueness is a more desired property than internal structure. However, since special reasoners only run on specially selected data, global uniqueness is less important than efficient mapping to domain-specific representations. It is not computationally optimal to deserialize data by running SPARQL queries.

Finally, as a last point in the comparison between RDF and CH semantics, it is worth considering the distinction (introduced, notably, in the "OpenCog" system) between "declarative knowledge" and "procedural knowledge" [**?**, especially pages 182-197]. According to this distinction, canonical RDF data exemplifies *declarative* knowledge because it asserts apparent facts without explicitly trying to interpret or process them. Declarative knowledge circulates among software in canonical, reusable data formats, allowing individual components to use or make inferences from data according to their own purposes.

Counter to this paradigm, return to hypothetical USH examples as I discussed at the top of this chapter. For example, consider the conversion of Voltage data to acceleration data, which is a prerequisite to accelerometers' readings being useful in most contexts. Software possessing capabilities to process accelerometers therefore reveals what can be called *procedural* knowledge, because software so characterized not only receives data but also processes such data in standardized ways.

The declarative/procedural distinction perhaps fails to capture how procedural transformations may be understood as intrinsic to some semantic domains — so that even the information we perceive as "declarative" has a procedural element. For example, the very fact that "accelerometers" are not called "Voltmeters" (which are something else) suggests how the Ubiquitous Computing community perceives voltage-to-acceleration calculations as intrinsic to accelerometers' data. But strictly speaking the components which participate in USH networks are not just engaged in data sharing; they are functioning parts of the network because they can perform several widely-recognized computations which are understood to be central to the relevant domain — in other words, they have (and share with their peers) a certain "procedural knowledge".

RDF is structured as if static data sharing were the sole arbiter of semantically informed interactions between different components, which may have a variety of designs and rationales — which is to say, a Semantic Web. But a thorough account of formal communication semantics has to reckon with how semantic models are informed by the implicit, sometimes unconscious assumption that producers and/or consumers of data will have certain operational capacities: the dynamic processes anticipated as part of sharing data are hard to conceptually separate from the static data which is literally transferred. To continue the accelerometer example, designers can think of such instruments as "measuring acceleration" even though *physically* this is not strictly true; their output must be mathematically transformed for it to be interpreted in these terms. Whether represented via RDF graphs or Directed Hypergraphs, the semantics of shared data is incomplete unless the operations which may accompany sending and receiving data are recognized as preconditions for legitimate semantic alignment.

While Ontologies are valuable for coordinating and integrating disparate semantic models, the Semantic Web has perhaps influenced engineers to conceive of semantically informed data sharing as mostly a matter of presenting static data conformant to published Ontologies (i.e., alignment of "declarative knowledge"). In reality, robust data sharing also needs an "alignment of *procedural* knowledge": in an ideal Semantic Network, procedural capabilities are circled among components, promoting an emergent "collective procedural knowledge" driven by transparency about code and libraries as well as about data and formats. The CH model arguably supports this possibility because it makes type assertions fundamental to semantics. Rigorous typing both lays a foundation for procedural alignment and mandates that procedural capabilities be factored in to assessments of network com-

ponents, because a type attribution has no meaning without adequate libraries and code to construct and interpret type-specific values.

Still, having just identified several notable differences between RDF and the Semantic Web, on the one hand, and Hypergraph-based frameworks, on the other, I hope not to overstate these differences; both belong to the overall space of graph database and graph-oriented semantic models. RDF graphs are both a plausible serialization of CH graphs and a reasonable interpretation of CH at least in some contexts. In particular, there are several Ontologies that formally model computer source code. This implies that code can been be modeled by suitably typed DHs as well. So, for any given procedure, assume that there is a corresponding DH representation which embodies that procedure's implementation.

Procedures, of course, depend on *inputs* which are fixed for each call, and produce "outputs" once they terminate. In the context of a graph-representation, this implies that some hypernodes represent and/or express values that are *inputs*, while others represent and/or express its *outputs*. These hypernodes are *abstract* in the sense (as in Lambda Calculus) that they do not have a specific assigned value within the body, *qua* formal structure. Instead, a *runtime manifestation* of a DH (or equivalently a CH, once channelized types are introduced) populates the abstract hypernodes with concrete values, which in turn allows expressions described by the CH to be evaluated.

These points suggest a strategy for unifying Lambda Calculii with Source Code Ontologies. The essential construct in Lambda Calculii is that mathematical formulae include "free symbols" which are *abstracted*: sites where a formula can give rise to a concrete value, by supplying values to unknowns; or give rise to new formulae, via nested expressions. Analogously, nodes in a graph-based source-code representation are effectively λ-abstracted if they model input parameters, which are given concrete values when the procedure runs. Connecting the output of one procedure to the input of another — which can be modeled as a graph operation, liking two nodes — is then a graph-based analog to embedding a complex expression into a formula (via a free symbol in latter).

Carrying this analogy further, I earlier mentioned different λ-Calculus extensions inspired by programming-language features such as Object-Orientation, exceptions, and so forth. These, too, can be incorporated into a Source Code Ontology: e.g., the connection between a node holding a value passed to an input parameter node, in a procedure signature, is semantically distinct from the nodes holding "Objects" which are senders and receivers for "messages", in Object-Oriented Parlance. Variant input/output protocols, including Objects and exceptions, are certainly semantic constructs (in the computer-code domain) which Source Code Ontologies should recognize. So we can see a convergence in the modeling of multifarious input/output protocols via λ-Calculus and via Source Code Ontologies. I will now discuss a corresponding expansion in the realm of applied Type Theory, with the goal of ultimately folding type theory into this convergence as well.

2.4 *Procedural Input/Output Protocols via Type Theory*

Parallel to the historical evolution where λ-Calculus progressively diversified and re-oriented toward concrete programming languages, there has been an analogous (and to some extent overlapping) history in Type Theory. When there are multiple ways of passing input to a function, there are at potentially multiple kinds of function types. For instance, Object-Orientation inspired expanded λ-calculi that distinguish function inputs which are "method receivers" or "**this** objects" from ordinary ("lambda") inputs. Simultaneously, Object-Orientation also distinguishes "class" from "value" types and between function-types which are "methods" versus ordinary functions. So, to take one example, a function telling us the size of a list can exhibit two different types, depending on whether the list itself is passed in as a method-call target (**list.size()** vs. **size(list)**).

One way to systematize the diversity of type systems is to assume that, for any particular type system, there is a category $\mathbb{T}$ of types conformant to that system. This requires modeling important type-related concepts as "morphisms" or maps between types. Another useful concept is an "endofunctor": an "operator" which maps elements in a category to other (or sometimes the same) elements. In a $\mathbb{T}$ an endofunctor selects (or constructs) a type $\mathcal{T}_2$ from a type $\mathcal{T}_1$ — note how this is different from a morphism which maps *values of* $\mathcal{T}_1$ to $\mathcal{T}_2$. Type systems are then built up from a smaller set of "core" types via operations like products, sums, enumerations, and "function-like" types.

We may think of the "core" types for practical program-

ming as number-based (booleans, bytes, and larger integer types), with everything else built up by aggregation or encodings (like ASCII and UNICODE, allowing types to include text and alphabets).[5] Ultimately, a type system $\mathbb{T}$ is characterized (1) by which are its core types and (2) by how aggregate types can be built from simpler ones (which essentially involves endofunctors and/or products).

In Category Theory, a Category $\mathbb{C}$ is called "Cartesian Closed" if for every pair of elements e_1 and e_2 in $\mathbb{C}$ there is an element $e_1 \rightarrow e_2$ representing (for some relevant notion of "function") all functions from e_1 to e_2 [**?**]. The stipulation that a type system $\mathbb{T}$ include function-like types is roughly equivalent, then, to the requirement that $\mathbb{T}$, seen as a Category, is Cartesian-Closed. The historical basis for this concept (suggested by the terminology) is that the construction to form function-types is an "operator", something that creates new types out of old. A type system $\mathbb{T}$ first needs to be "closed" under products: if $\mathcal{T}_1$ and $\mathcal{T}_2$ are in $\mathbb{T}$ then $\mathcal{T}_1 \times \mathcal{T}_2$ must be as well. If $\mathbb{T}$ is *also* closed under "functionalization" then the $\mathcal{T}_1 \times \mathcal{T}_2$ product can be mapped onto a function-like type $\mathcal{T}_1 \rightarrow \mathcal{T}_2$.

In general, then, more sophisticated type systems $\mathbb{T}$ are described by identifying new kinds of inter-type operators and studying those type systems which are closed under these operators: if $\mathcal{T}_1$ and $\mathcal{T}_2$ are in $\mathbb{T}$ then so is the combination of $\mathcal{T}_1$ and $\mathcal{T}_2$, where the meaning of "combination" depends on the operator being introduced. Expanded λ-calculi — which define new ways of creating functions — are correlated with new type systems, insofar as "new ways of creating functions" also means "new ways of combining types into function-types".

Furthermore, "expanded" λ-calculi generally involve "new kinds of abstraction": new ways that the building-blocks of functional expressions, whether these be mathematical formulae or bodies of computer code, can be "abstracted", treated as inputs or outputs rather than as fixed values. In this chapter, I attempt to make the notion of "abstraction" rigorous by analyzing it against the background of DHs that formally model computer code. So, given the correlations I have just described between λ-calculi and type systems — specifically, on $\mathbb{T}$-closure stipulations — there are parallel correlations between type systems and *kinds of abstraction defined on Channelized Hypergraphs*. I will now discuss this further.

[5] In other contexts, however, non-mathematical core types may be appropriate: for example, the grammar of natural languages can be modeled in terms of a type system whose core are the two types **Noun** and **Proposition** and which also includes function types (maps) between pairs or tuples of types (verbs, say, map **Noun**s — maybe multiple nouns, e.g. direct objects — to **Proposition**s).

2.4.1 Kinds of Abstraction

The "abstracted" nodes in a CH can be loosely classified as "input" and "output", but in practice there are various paradigms for passing values into and out of functions, each with their own semantics. For example, a "**this**" symbol in C++ is an abstracted, "input" hypernode with special treatment in terms of overload resolution and access controls. Similarly, exiting a function via **return** presents different semantics than exiting via **throw**. As mentioned earlier, some of this variation in semantics has been formally modeled by different extensions to Lambda Calculus.

So, different hypernodes in a CH are subject to different kinds of abstraction. Speaking rather informally, hypernodes can be grouped into *channels* based on the semantics of their kind of abstraction. More precisely, channels are defined initially on *symbols*, which are associated with hypernodes: in any "body" (i.e., an "implementation graph") hypernodes can be grouped together by sharing the same symbol, and correlatively sharing the same value during a "runtime manifestation" of the CH. Therefore, the "channels of abstraction" at work in a procedure can be identified by providing a name representing the *kind* of channel and a list of symbols affected by that kind of abstraction. In the notation I adopt here, conventional lambda-abstraction like $\lambda x.\lambda y$ would be written as $\not\Leftarrow_{\text{LAM}}.xy$.

I propose "Channel Algebra" as a tactic for capturing the semantics of channels, so as to model programming languages' conventions and protocols with respect to calls between procedures. Once we get beyond the basic contrast between "input" and "output" parameters, it becomes necessary to define conditions on channels' size, and on how channels are associated with different procedures may share values. Here are several examples:

- In most Object-Oriented languages, any procedure can have at most one **this** ("message receiver") object. Let $\not\Leftarrow_{\text{SIG}}$ model a "Sigma" channel, as in "Sigma Calculus" (written as ς-calculus: see e.g. [1], [28], [6]). We then have the requirement than any procedure's $\not\Leftarrow_{\text{SIG}}$ channel can carry at most one value.

- In all common languages which have exceptions, procedures can *either* throw an exception *or* return a value. If **return** and **exception** model the channels carrying standard returns and

thrown exceptions, respectively, this convention translates to a requirement that the two channels can both be non-empty.

- A thrown exception cannot be handled as an ordinary value. The whole point of throwing exceptions is to disrupt ordinary program flow, which means the exception value is only accessible in special constructs, like a **catch** block. One way to model this restriction is to forbid **exception** channels from sharing values with most other channels (tied to any other procedure). Instead, exception values are bound (in **catch** blocks) to lexically-scoped symbols (I will discuss channel-to-symbol transfers below).

- Suppose a procedure is an Object-Oriented method (it has a non-empty "⫽SIG" channel). Any other methods called from that procedure will — at least in the conventional Object-Oriented protocol — automatically receive the enclosing method's Sigma channel unless a different object for the called method is supplied expressly.

- In the object-oriented technique known as "method chaining", one procedures' **return** channel is transferred to a subsequent procedures' ⫽SIG channel. The pairing of Return and Sigma channels therefore gives rise to one function-composition operator. With suitable restrictions (on channel size), Return and Lambda channels engender a different function-composition operator. So channels can be used to define operators between procedures which yield new function-like values (i.e., instances of function-like types). In some cases, function-like values defined via inter-function operators can be used in lieu of function-like values instantiated from implemented procedures (although the specifics of this substitutability — an example of so-called "eta (η) equivalence" — varies by language).

The above examples represent possible combinations or interconnections (sharing values) between channels, together with semantic restrictions on when such connections are possible. In this chapter, I assume that notations describing these connections and restrictions can be systematized into a "Channel Algebra", and then used to model programming language-conventions and computer code. A basic example of inter-channel aggregation would be how a Lambda channel, combined with a Return channel, associated with one procedure, yields a conventional input/output pairing. One particular channel formation therefore models the basic λ-Calculus and, simultaneously, the minimal theory of function-like types (for Cartesian Closed type Categories). More complex channel combinations and protocols can then model more complex (or at least more modern) variations on λ-Calculii and programming language type systems.

Collectively, in short, I will say that formulations describing channel kinds, their restrictions, and their interrelationships describe a *Channel Algebra*, which express how channels combine to describe possible function signatures — and accordingly to describe functional *types*. The purpose of a Channel Algebra is, among other things, to describe how formal languages (like programming languages) formulate functions and the rules they put in place for inputs and outputs. If χ is a Channel Algebra, a language adequately described by its formulations (channel kinds, restrictions, and interrelationships) can be called a χ-language. The basic Lambda Calculus can be described as a χ-language for the algebra defined by a minimal **lambda** plus **return** combination (with **return** channels restricted to at most one element). Analogously, a type system $\mathbb{T}$ is a "χ-type-system", and is "closed" with respect to χ, if valid signatures described using channel kinds in χ correspond to types found in $\mathbb{T}$. Types may be less granular than signatures: as a case in point, functions differing in signature only by whether they throw exceptions may or may not be deemed the same type. But a channel construction on types in $\mathbb{T}$ must also yield a type in $\mathbb{T}$.

I say that a type system is *channelized* if it is closed with respect to some Channel Algebra. Channelized Hypergraphs are then DHs whose type system is Channelized. We can think of channel constructions as operators which combine groups of types into new types (this operative dimensions helps motivate describing channel logics and their formulae as "algebras"). Once we assert that a CH is Channelized, we know that there is a mechanism for describing some Hypergraphs as "function implementations" some of whose hypernodes are subject to kinds of abstraction present in the relevant Channel Algebra. The terse notation for Channel formulae and signatures describes logical norms which can also be expressed with more conventional Ontologies. So Channel Algebra can be seen as a generalization of (RDF-environment) Source Code Ontology (of the kinds studied for example by [11], [12], [13], [24], [25], [26]). Given the relations between RDF and Directed Hypergraphs (despite differences I have discussed here), Channel Algebras can also be seen as adding to Ontologies governing Directed Hypergraphs. Such is the perspective I will take for the remainder of this chapter.

For a Channel Algebra χ and a χ-closed type system (written, say) $\mathbb{T}^{\chi}$, χ extends $\mathbb{T}$ because function-signatures conforming to χ become types in $\mathbb{T}$. At the same time, $\mathbb{T}$ also

extends χ, because the elements that populate channels in χ have types within $\mathbb{T}$. Assume that for any type system, there is a partner "Type Expression Language" (TXL) which governs how type descriptions (especially for aggregate types that do not have a single symbol name) can be composed consistent with the logic of the system. The TXL for a type-system $\mathbb{T}$ can be notated as $\mathfrak{L}_{\mathbb{T}}$. If $\mathbb{T}$ is channelized then its TXL is also channelized — say, $\mathfrak{L}_{\mathbb{T}^{\chi}}$ for some χ.

Similarly, we can then develop for Channel Algebras a *Channel Expression Language*, or CXL, which can indeed be integrated with appropriate TXLs. The notation I adopted earlier for stating Channel Algebra axioms is one example of a CXL, though variant notations may be desired for actual computer code (as in the code samples accompanying this chapter). However, whereas the CXL expressions I have written so far describe the overall shape of channels — which channels exist in a given context and their sizes — CXL expressions can also add details concerning the *types* of values that can or do populate channels. CXL expressions with these extra specifications then become function signatures, and therefore can be type-expressions in the relevant TXL. A channelized TXL is then a superset of a CXL, because it adds — to CXL expressions for function-signatures — the stipulation that a particular signature does describe a *type*; so CXL expressions become TXL expressions when supplemented with a proviso that the stated CXL construction describes a function-type signature. With such a proviso, descriptions of channels used by a function qualifies as a type attribution, connecting function symbol-names to expressions recognized in the TXL as describing a type.

Some TXL expressions designate function-types, but not all, since there are many types (like integers, etc.) which do not have channels at all. While a TXL lies "above" a CXL by adding provisos that yield type-definition semantics from CXL expressions, the TXL simultaneously in a sense lies "beneath" the CXL in that it provides expressions for the non-functional types which in the general case are the basis for CXL expressions of functional types, since most function parameters — the input/output values that populate channels — have non-functional types. Section §3 will discuss the elements that "populate" channels (which I will call "carriers") in more detail.

In the following sections I will sketch a "Channel Algebra" that codifies the graph-based representation of functions as procedures whose inputs and outputs are related to other functions by variegated semantics (semantics that can be catalogued in a Source Code Ontology). With this foundation, I will argue that Channel-Algebraic type representations can usefully model higher-scale code segments (like statements and code blocks) within a type system, and also how type interpretations can give a rigorous interpretation to modeling constructs such as code specifications and "gatekeeping" code. I will start this discussion, however, by expanding on the idea of the use of code-graphs — hypergraphs annotated according to a Source Code Ontology — to represent procedure implementations, and therefore to model procedures as instances of function-like types.

References

1 Martin Abadi and Luca Cardelli, "A Semantics of Object Types". Proceedings of the IEEE Symposium on Logic in Computer Science, Paris, 1994. http://lucacardelli.name/Papers/PrimObjSemLICS.A4.pdf

2 Ronald Ashri, *et. al.*, "Torwards a Semantic Web Security Infrastructure". American Association for Artificial Intelligence, 2004. https://eprints.soton.ac.uk/259040/1/semantic_web_security.pdf

3 Bob Coecke, *et. al.*, "Interacting Conceptual Spaces I: Grammatical Composition of Concepts". https://arxiv.org/pdf/1703.08314.pdf

4 Trevor Elliott, *et. al.* "Guilt Free Ivory". Haskell Symposium 2015 https://github.com/GaloisInc/ivory/blob/master/ivory-paper/ivory.pdf?raw=true

5 Michael Engel, *et. al.*, "Unreliable yet Useful – Reliability Annotations for Data in Cyber-Physical Systems". https://pdfs.semanticscholar.org/d6ca/ecb4cd59e79090f3ebbf24b0e78b3d66820c.pdf

6 Kathleen Fisher, *et. al.*, "A Lambda Calculus of Objects and Method Specialization". Nordic Journal of Computing 1 (1994), pp. 3-37. https://pdfs.semanticscholar.org/5cf7/1e3120c48c23f9cecdbe5f904b884e0e1a2d.pdf

7 Brendan Fong, "Decorated Cospans" https://arxiv.org/abs/1502.00872

8 Brendan Fong, "The Algebra of Open and Interconnected Systems". Oxford University, dissertation 2016. https://arxiv.org/pdf/1609.05382.pdf

9 Wolfgang Jeltsch, "Categorical Semantics for Functional Reactive Programming with Temporal Recursion and Corecursion". https://arxiv.org/pdf/1406.2062.pdf

10 Lalana Kagal, *et. al.*, "A Policy-Based Approached to Security for the Semantic Web". https://ebiquity.umbc.edu/_file_directory_/papers/60.pdf

11 Iman Keivanloo, *et. al.*, "Semantic Web-based Source Code Search". http://wtlab.um.ac.ir/images/e-library/swese/Semantic%20Web-based%20Source%20Code%20Search.pdf

12 Werner Klieber, *et. al.*, "Using Ontologies For Software Documentation". http://www.know-center.tugraz.at/download_extern/papers/MJCAI2009%20software%20ontology.pdf

13 Johnathan Lee, *et. al.*, "Task-Based Conceptual Graphs as a Basis for Automating Software Development". https://www.csie.ntu.edu.tw/~jlee/publication/tbcg99.pdf

14 Aleks Kissinger, "Finite Matrices are Complete for (dagger-)Hypergraph Categories". https://arxiv.org/abs/1406.5942

15 Mikhail V. Malko, “The Chernobyl Reactor: Design Features and Reasons for Accident.” http://www.rri.kyoto-u.ac.jp/NSRG/reports/kr79/kr79pdf/Malko1.pdf

16 GiladMishne and Maarten de Rijke, “Source Code Retrieval using Conceptual Similarity”. https://staff.fnwi.uva.nl/m.derijke/Publications/Files/riao2004.pdf

17 Santanu Paul and Atul Prakash, “Supporting Queries on Source Code: A Formal Framework”. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.52.9136&rep=rep1&type=pdf

18 Jennifer Paykin, et. al., “Curry-Howard for GUIs: Or, User Interfaces via Linear Temporal, Classical Linear Logic”. https://www.cl.cam.ac.uk/~nk480/obt.pdf

19 Jennifer Paykin, et. al., “The Essence of Event-Driven Programming” https://jpaykin.github.io/papers/pkz_CONCUR_2016.pdf

20 Lavanya Ramapantulu, *et. al.*, “A Conceptual Framework to Federate Testbeds for Cybersecurity”. Proceedings of the 2017 Winter Simulation Conference. http://simulation.su/uploads/files/default/2017-ramapantulu-teo-chang.pdf

21 Takeshi Takahashi, *et. al.*, “Ontological Approach toward Cybersecurity in Cloud Computing”. 3rd International Conference on Security of Information and Networks (SIN 2010), Sept. 7-11, 2010, Taganrog, Rostov Oblast, Russia. https://arxiv.org/pdf/1405.6169.pdf

22 Mozhgan Tavakolifard, “On Some Challenges for Online Trust and Reputation Systems”. Dissertation, Norwegian University of Science and Technology, 2012. https://pdfs.semanticscholar.org/fc60/d309984eddd4f4229aa56de2c47f23f7b65e.pdf

23 Scott R. Tilley, *et. al.*, “Towards a Framework for Program Understanding”. https://pdfs.semanticscholar.org/71d0/4492be3c2abf9e1a88b9b263193a5c51eff1.pdf

24 Raymond Turner and Amnon H. Eden, “Towards a Programming Language Ontology”. https://www.researchgate.net/publication/242381616_Towards_a_Programming_Language_Ontology

25 Rene Witte, *et. al.*, “Ontological Text Mining of Software Documents”. https://pdfs.semanticscholar.org/7034/95109535e510f81b9891681f99bae1e704fc.pdf

26 Pornpit Wongthongtham, *et. al.*, “Development of a Software Engineering Ontology for Multi-site Software Sevelopment”. https://ifs.host.cs.st-andrews.ac.uk/Research/Publications/Papers-PDF/2005-09/TKDE-Ponpit-2009.pdf

27 Joon-Eon Yang, “Fukushima Dai-Ichi Accident: Lessons Learned and Future Actions from the Risk Perspectives.” https://www.sciencedirect.com/science/article/pii/S1738573315300875

28 Edward N. Zalta, “The Modal Object Calculus and its Interpretation”. https://mally.stanford.edu/Papers/calculus.pdf