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

Nathaniel Christen

August 15, 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 [**?**] and the Fukushima nuclear reactor meltdown [**?**]. 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 indicate 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. [6], [7], [4]) and the theory of Hypergraph Categories ([**?**], [2], [3], [5]).

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 permits 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 procures 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.

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*

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 and does not itself verify that these 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 exteremely 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. With 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, 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 [**?**] and [**?**], 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 information 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 CyberPhysical code from a perspective that cuts across multiple dimensions of project scale and lifetime. Do we design for biaxial 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 (like Ontologies, or database schema). 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 from the ground up as individual functions and datatypes are implemented. For this reason, code development serves a role beyond just providing the software which a system, once placed in operation, will use. 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 operationalized 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 that I just highlighted. Deployed systems 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 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 function-by-function 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 function 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 function, 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 (SLE), 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?

Most type systems provide only relatively coarse classification of a universe of typed values — even though many functions require their arguments to fit more precise specifications than practical type systems allow. This is unfortunate given the premise of "separation of concerns" and the maxim that functions should have single and narrow roles: *validating* input is actually a different role than *doing* calculations. Maximwise, then, functions with fine requirements can be split into two: a gatekeeper that validates input before a fragile function is called, and separate from that the function's own implementation itself. 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.

Despite their potential elegance, most practical programming languages do not supply much language-level support for expressing groups of fine-grained functions along these lines. Dependent Types, typestate, and effect-systems are each models or paradigms which can document function implementation requirements with more precision than can be achieved via conventional type systems alone. Integrating these paradigms into type systems — which is done, at least incompletely, at least in some versions of some programming languages — allows requirements to be confirmed by general 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). So there are no intractable *formal* obstacles to augmenting the expressiveness of practical languages' type systems. Nevertheless, such enhancements appear to be either sufficiently difficult to "language engineer", and/or to adversely affect language performance, enough that the benefits of type-expressiveness do not on net improve the language. Or at least, it is reasonable to assume that those responsible for maintaining languages and language tools (specifications, compilers, standard libraries) believe as much, so that Dependent Types (for example) are only internally supported by a few (mostly academic) languages.

This impasse is frustrating not only for practical reasons — common programming languages lack features which would make developers more productive — but, also, more philosophically. I would argue that the most *philosophically* well-grounded and justifiable style of language — a collection of SLE norms whose paradigms carry the most weight when considerations from multiple disciplines and theories are factored in, like linguistics, mathematics, and cognitive science — would feature default "lazy" evaluation as in Haskell (expressions are not evaluated until their results are needed), Dependent Types as in Idris, and "effect typing", of a genre hesitant to single out effect-free functions as preferable but which *does* recognize effect-capabilities as a discriminating factor in a mature type system. However, that particular trio of features or approaches has not been embraced by any realistic language, certainly not any in widespread use. We can debate whether this reflects technical language-implementation difficulties — and whether this trio of paradigms is superior to alternatives — but the overarching point is that Software Language Implementation should be flexible enough to support multiple paradigms — precisely because we'd like to embrace SLE paradigms for reasons not exclusively driven by engineering feasibility.

Programming languages and the code written in them are a kind of structural creole, partly engineered artifacts and machinery existing in virtual/digital spaces, and partly human texts and conventions. The structure and elemental form (grammar, data layout) of these artifacts similarly joins human and engineering concerns. Language design should therefore be informed by human as well as mathematical and computing sciences — after all, a programming language is a *language*, a vehicle of human expression and communication. Code "expresses" routines for computer processing rather than address to other humans, but as programmers we also write code to be understood by others, communicating indirectly via our shared understanding of how computer languages work. So programming languages are indeed communicative media of a certain sort. As human artifacts rigid enough for a digital environment, they are an exceptional forum for exploring human cognition and signification in a structured, formally tractable milieu. Software Language design should be based on human cognitive and communicative structures, as well as on structural mathematics, and philosophy centered on human thought and experience should be able to guide (and learn from) Software Language Engineering (William Rapaport [**?**] has an interesting discussion along these lines, which is intriguing to read alongside overviews of the OpenCog project I will cite later).

This makes SLE implementational limitations especially troubling: language design should be guided by philosophy and mathematics, not by estimations of which language features are practical given the state of current programming languages.

If these observations are correct, I maintain that it is a worthwhile endeavor to return to the theoretical drawing board and explore how type theory itself can shed light on the implementational obstacles that we observe in practice. I will argue that topics which influence the feasibility of concrete language-level implementations are insufficiently modeled by conventional type theory; so grounding analyses on Software Language Engineering practice can potentially extend the theory. Certain language-specific phenomena do not have evident conceptualizations in the kind of formal type theory whose "language" is more of an abstraction, like a Lambda (written λ-) Calculus, than a physically realized complex system.

Of the many approaches to specifications and verification, we might recognize two distinct tendencies. 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.

The attitude I just summarized — which perhaps can be called "internalist" — 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*" [**?**, p. 1] (my emphasis). In other words, the creators of Ivory are promoting the fact that their language buttresses via its type system 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" programming language and coding activity. In particular, most of the more important or complex safety-checking does not natively integrate with the underlying language, but instead requires some external source code analyzer, or runtime libraries. 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, consider Santanu Paul's Source Code Algebra (SCA) system described in [**?**] and [**?**], [**?**]:

> 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. [**?**, 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 are normative as well as descriptive: they influence language design and how languages influence coding practices. For Functional Programmers — taking them as representative of an influential but not dominant mindset — sufficiently expressive type systems and a preference for side-effect-free function types yields code which has fewer locations where erroneous run-time behavior can occur, and so is easier to evaluate and maintain. Nonetheless, advocates for Object-Oriented architectures might reply, paradigms like OO provide more conceptually accurate and intuitive models, in terms of the cross-fit between digital representations and real-world phenomena. If it requires greater structural alteration to translate conceptual models into functional-programming designs, this undermines the apparent benefits of Functional Programming in the areas of code evaluation and maintenance.

A conceptual/modeling rejoinder along these lines — that is, as a counter to functional-programming advocacy if it becomes too dogmatic — might be weakened if the kinds of performative guarantees that can be made in Functional contexts had no equivalents vis-à-vis other paradigms. But even in an OO context, say, there *are* analyses focused on detecting code which (via constructs that Functional Programming avoids) threatens problematic behavior. What is different in the OO context compared to Functional Programming is that the safety-critical analyses (to find "dangling pointers", race conditions, and so forth) are not so much tools within the language but external projects which take source code as a data structure to be traversed and queried (and/or running program instances as empirical phenomena to be observed). Insofar as code anomalies can be found through rigorous methods, source code and live software have formally tractable layers of organization that are not exhausted by the mathematical concepts internal to programming languages and language engines (compilers and runtimes) themselves. Errors in imperative or Object-Oriented code may be detectable through a sufficiently powerful analytic framework (or at least a sufficiently thorough test suite) even if they are not *formal* errors vis-à-vis the type system or vis-à-vis a "programs are proofs" SLE implementation strategy.

To be sure, functional programmers might argue that strong type systems and functional idioms (algebraic datatypes, pattern matching as a control flow device, by-need/"lazy" evaluation, immutable but cheaply copyable data structures) produce more elegant and practical formalizations than "externalist" analyses, which are more likely to be adhoc and trial-end-error — and, perhaps significantly, require maintaining or updating dependencies on an entirely separate code base for testing/evaluation tools, with origins distinct from both the language and the projects that use it. Just because external analysis of an OO code base is *possible*, they might argue, it does not follow that OO design is a better choice, compared to a Functional design whose *external* analytic needs may be simpler and more cost-effective. These are plausible but rather subjective assessments. In cases where OO designs lead to computational models of human or scientific realms which are *conceptually* more accurate, but also require more investment in external analysis tools for safety and review, is the added difficulty of retroactive code analysis an acceptable trade-off? That question can depend on many factors: a responsible Software Language Engineer may have to accept that different programming paradigms (plus multiparadigm combinations) are best suited for different projects and circumstances, and that the broadest tools and languages need multi-paradigm orientations.

Language engineers, then — particularly for general-purpose, multi-paradigm programming languages — 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). If this gloss (which I admit rests on some speculation and mind-reading) has any merit, it implies that one challenge for language engineers is to serve both communities. For example, we can 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 §I and §II.

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, such as C++.

After this mostly theoretical prelude, I conclude this chapter with a discussion of code annotation, particularly in the context of CyberPhysical Systems. Because Cyber-Physical 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 ([**?**], [**?**], [**?**], [**?**], [**?**]). 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.

In particular, to take an example especially relevant for this volume, the code which directly interacts with USH devices needs particularly thorough documentation, review, and (perhaps, as a way to achieve these) annotations. Code designed and annotated via techniques reviewed in this chapter will not be guaranteed to protect privacy, block malware, or detect all device-related errors. But such code *will* be amenable to analytic processes which should increase different parties' (doctors, patients, application developers) assessment of its transparency and trustworthiness. In the end, components earn trust not through one monolithic show of robustness but via designs judged to reflect quality according to multiple standards and paradigms, with each approach to code evaluation adding its own measure to stakeholders' overall trust in the system.

References

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

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

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

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

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

6 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

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