Hypergraph-Based Type Theory for Requirements Engineering and Generalized Lambda Calculus (with a case study in safety protocols for biomedical devices)

Nathaniel Christen

August 18, 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 vary-

ing), 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 Cyber-Physical 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. 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 differ-

ent 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 ques-

tions 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

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

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 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]

[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

To avoid confusion, I will hereafter use the word "hyponode" 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

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.

(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" ("⟨SUBJECT, PREDICATE, OBJECT⟩", 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 hyperedges 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 components, 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 programming 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.

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 $\not\triangleleft_{\text{LAM}}.xy$.

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:

[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).

- In most Object-Oriented languages, any procedure can have at most one **this** ("message receiver") object. Let ⫽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 ⫽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.

2.4.2 Channelized Type Systems

Collectively, to summarize my disuccion to this point, 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.

3 Modeling Procedures via Channelized Hypergraphs

Assuming we have a suitable Source Code Ontology, software procedures can be seen from two perspectives. On the one hand, they are examples of well-formed code graphs: annotated graph structures convey the lexical symbols, input/output parameters (via different "abstractions", in the sense of λ-abstraction, subject to relevant channel protocols), and calls to other procedures, through which any given procedure's functionality is achieved. On the other hand, we can see procedures as instances of function-like types, where the types carried in each channel determine the type of the procedure itself, as a functional value. Although these two perspectives are usually mutually consistent, the notion of functional values is more general than procedures which are

expressly implemented in computer code. In particular, as I briefly mentioned earlier, sometime functional values are denoted via inter-function operators (like the composition $f \circ g$) rather than by giving an explicit implementation. Programmers would say that functions defined via operators (such as $\circ$) lack a "function body".

Going forward, I will generally use the term *procedure* with reference to function-like type instances that are defined *with* function bodies: that is, they are associated with sections of code that supply the procedure's implementation, and can be represented via code-graphs. I will use the term *function* more generally for instances of function-like types, irregardless of their provenance. In particular, functions are *values* — instances of types in a relevant type-system $\mathbb{T}$ — whereas I will not usually discuss procedures as "values".

Most functions which may be called by a software component are defined as procedures with their own function bodies. Indeed, the essential software-development process involves implementing procedures in code, and then implementing other procedures — which call those already-created procedures (in various combinations) — to realize their own functionality. That is, source code itself produces instances of functional values, via procedures described by the code. Accordingly, code-graphs for individual procedures capture the implementations through which function-like types are (mostly) populated with concrete values.

These various concepts — procedures, functions, implementations, and function-bodies or code-graphs — are important for broad architectural topics like Requirements Engineering, because they concern the building-blocks from which larger software components are assembled. When discussing procedures' coding assumptions (more fine-grained than type constraints alone can model), for example, we need a rigorous presentation of what a procedure itself is, to identify the relevant entity whose assumptions can be documented and tested. To model the general maxim that any coding assumptions made (but not verified) by one procedure — say, $\mathcal{P}_1$ — should be tested by other procedures which call $\mathcal{P}_1$, we need a systematic outline capturing the notion of procedures calling other procedures, in the course of their own implementation. Here I propose to model these details via channels and interrelationships between channels.

In my formulation of these representations, channels are conceptually integrated with hypergraph code models. That is, code-graphs are a formal device within the theory I present here. In many other context code-graphs would instead be, at most, convenient expository devices; for instance, functional programming languages generally do not attach much significance to the contrast of procedures (with function-bodies) and function-values constructed by other means. As a result, one consequence of my graph-oriented approach is that the technical distinctions between procedures and function-values (in general) have to be duly observed. There are some relevant complications appertaining to the general picture of source-code segments instantiating function-like types. I will briefly review these issues now, before pivoting to more macro-scale themes like Requirements Engineering.

3.1 Initializing Function-Typed Values

Although in general function-typed values are *initialized* from code-graphs that blueprint their implementation, this glosses over several different mechanisms by which function-typed values may be defined:

1. In the simplest case, there is a one-to-one relationship between a code graph and an implemented function (**f**, say). If **f** is polymorphic, in this case, it must be an example of subtype (or "runtime") polymorphism where the declared types of **f**'s parameters are actually instantiated, at runtime, by values of their subtypes.

2. A different situation ("compile-time" polymorphism) applies to generic code as in C++ templates. Here, a single code-graph generates multiple function bodies, which differ only by virtue of their expected types. For example, a templated **sort** function will generate multiple function bodies — one for integers, say, one for strings, etc. These functions may be structurally similar, but they have different signatures by virtue of working with different types. This means that symbols used in the function-bodies may refer to different functions even though the symbols themselves do not vary between function-bodies (since, after all, they come from the same node in a single code-graph). That is, the code-graphs rely on symbol-overloading for function names to achieve a kind of polymorphism, where one code-graph yields multiple bodies. In this compile-time polymorphism, symbols are resolved to the proper overload-implementation at compile-time, whereas in runtime polymorphism this decision is deferred until the runtime-polymorphic function is actually being executed. The key difference is that runtime-polymorphic functions

are *one* function-typed value, which can work for diverse types only via subtyping — or via more exotic forms of indirection, like using function-pointers in place of function symbols; whereas compile-time-polymorphic (i.e., templated) functions are *multiple* values, which share the same code-graph representation but are otherwise unrelated.

3. A third possibility for producing function values is to define operators on function types themselves, which transform function-typed values to other function-typed values, by analogy to how arithmetic operations transform numbers to other numbers. As will be discussed below, this may or may not be different from initializing function-typed values via code-graphs, depending on how the relevant programming language is implemented. For instance, given the composition operator $\circ$, **f∘g** may or may not be treated as only a convenient shorthand for a code graph spelling out something like **f(g(x))**.

4. Finally, as a special case of operators on function-typed values, one function may be obtained from another by "Currying", that is, fixing the value of one or more of the original function's arguments. For example, the **inc** ("increment") function which adds **1** to a value is a special case of addition, where the added value is always **1**. Here again, Currying may or may not be treated as a function-value-initialization process different from ones starting from code-graphs.

The differences between how languages may process the *initialization* of function-type values, which I alluded to in (3) and (4), reflect differences in how function-type values are internally represented. One option is to store these values solely in blocks of memory, which would correspond to treating all initializations of these values as via code-graphs. For example, suppose we have an **add** function and want to define an **inc** function, as in **int inc(int x){return add(x,1)}**. Even if a language has a special Currying notation, that notation could translate behind-the-scenes to an explicit function body, like the code at the end of the last sentence. However, a language engine may also note that **inc** is derived from **add** and can be wholly described by a handle denoting **add** (a pointer, say) along with a designation of the fixed value: in other words, ⟨**&add, 1**⟩. Instead of initializing **inc** from a code-graph, the language can represent it via a two-part data structure like ⟨**&add, 1**⟩ — but only if the language *can* represent function-typed values as compound data structures.

Let's assume a language can always represent *some* function-typed values, ones that are obtained from code-graphs, via pointers to (or some other unique identifier for) an internal memory area where at least *some* compiled function bodies are stored. The interesting question is whether *all* function-typed values are represented in this manner and, in either case, the consequences for the semantics of functional types — semantic issues such as $f \circ g$ composition operators and Currying (and also, as I will argue, Dependent Types).

3.2 *Addressability and Implementation*

As *Intermediate* Representations, formal code models (including those based on DHs) are not strictly identical to actual computer code as seen in source-code files. In particular, what appears to be a single function body may actually form multiple implementations via code templates (or even preprocessor macros). Talk about polymorphism in a language like C++ covers several distinct language features: achieving code reuse by templating on type symbols is internally very different from using virtual methods calls. The key difference — highlighted by the contrast between runtime- and compile-time polymorphism — is that there are some function implementations which actually compile to *single* functions, meaning in particular that their compiled code has a single place in memory and that they may be invoked through function pointers. Conversely, what appears in written code as one function body may actually be duplicated, somewhere in the compiler workflow, generating multiple function values. The most common cases of such duplication are templated code as discussed above (though there are more exotic options, e.g. via C++ macros and/or repeated file **#include**s). Implementations of the first sort I will call "addressable", whereas those of the second I will dub "multi-addressable". These concepts prove to be consequential in the abstract theory of types, although for non-obvious reasons.

To see why, consider first that type systems are intrinsically pluralistic: there are numerous details whereby the type system underlying one computing environment can differ from those employed by other environments. These include differences in how types are composed from other types. There is therefore no abstract vocabulary (including the language of mathematical type theory) that provides a neutral and complete way of describing types across systems. Instead, each system has its own structure of multi-type aggregation, and so requires its own style of type description (mathemati-

cally, there is no one "Category" of types, but rather multiple candidate Categories with their own logic). So there is no single, universal "Type Expression Language": each type system has its own TXL with subtly different features than others.

By intent, I use TXL to mean languages for describing types which *may* be implemented. For example, if in C++ I assert "**template<T>MyList**", it would then be consistent with a C++-specific TXL to describe a type as **MyList<int>** (which would presumably be some sort of list of integers, though of course naming hints about the intended use of a type have no bearing on how compilers and runtimes process it). However, the type **MyList<int>** is not, without further code, actually implemented. It is a *possible* type because its description conforms to a relevant TXL, but not an *actual* type. If a programmer supplies a templated implementation for **MyList<T>** (intended for multiple types **T**) then the compiler can derive a "specialization" of the template for a specific **T** — or the programmer can specialize **MyList** on a chosen type manually. But in either case the actualization of **MyList<T>** will depend on an implementation (either a templated implementation that works for multiple types or a specialization for a single type); this is separate and apart from **MyList<T>** being a valid *expression* denoting a *possible* type.

Once **MyList<T>** is instantiated, for a particular **T**, there may be a constructor or initialization function that is *addressable*, either as one duplicate of a multi-addressable implementation or as the compilation of one non-templated function body. Call a type *addressable* if it has at least one constructor (i.e., "value" or "data" constructor, a function which creates a value of a type from a literal or a value of a different type); and *multi-addressable* if it has at least one multi-addressable constructor or initializer: these terms can carry over from functions to types for which functions classified in these terms are constructors.[6]

Addressability refers at one level, as the word suggests, to "taking the address" of functions (and accordingly to function-pointers); but here I also refer to a broader question of how functions can be designated from vantage points outside their immediate implementation — code searches, scripting engines, IDE-based code exploration, and other reflection-oriented use cases. Language engines should try to minimize their reliance on "temporary function values" which are opaque to these reflection-oriented features. And yet this can complicate the implementation of type-system features. To reiterate: the goal of "expressive" type systems is to define types, as necessary, narrowly enough to type-check granular procedure requirements. The problem is that gatekeeper code induced by type-level expressiveness — particularly code which is automatically generated — can be opaque to extra-linguistic environments like IDEs and scripting engines.

For example, suppose certification requires that the function which displays the gas level on a car's dashboard never attempts to display a value above **100** (intended to mean "One Hundred percent", or completely full). One way to ensure this specification is to declare the function as taking a *type* which, by design, will only ever include whole numbers in the range **(0,100)**. Thus, a type system may support such a type by including in its TXL notation for "range-delimited" types, types derived from other types by declaring a fixed range of allowed values. A notation might be, say, **int(0,100)**, for integers in the **(0,100)** range — or, more generally expressions like **T**(V_1,V_2), meaning a *type* derived from **T** but restricted to the range spanned by V_1 and V_2 (assumed to be values of **T** — notice that a TXL supporting this notation must consequently support some notation of specific values, like numeric literals). This is a reasonable and, for programmers, potentially convenient type description.

For a language designer, however, it raises questions. What should happen if someone tries to construct an **int(0,100)** value with the number, say, **101**? How should the range-test code be exposed for reflection (is it a separate function; is it a regular function-typed value or some alternative data structure, and how does that affect its external designating)? What if the number comes from a location that opaque to the language engine, like a web API: should the compiler assume that the API is curated to the same specifications as the present code or should it report that there is no way to verify that the declared **int(0,100)** type is being used correctly? Moreover, would such choices lead to behind-the-scenes, perhaps auto-generated code which is hard to wrangle for reflection and development tools? Are the benefits of automated gatekeeping worth the risk of codewriters' mental disconnect with language internals? Given these questions, it is reasonable for a language designer to *allow* certain sorts of types to be described, but programmers must explicitly implement them for the types to be actualized and available for use in programs. Therefore there is a difference between

[6] In this discussion I mostly skip over the technical detail that in C++, at least, one cannot actually take the address of a constructor function; but this is related to C++ "constructors" having dual roles of allocating memory and initialization: we *can* take the address of an initialization function that would be analogous to a "pure" value constructor.

a *described* type and an *actual* type, and the key criterion of actual types is an addressable value constructor. So the crucial step for type-theoretic design promoting desired software qualities, like safety and reliability, is to successfully pair the *description* of types which have desirable levels of specification and granularity, with the *implementation* of types that realize the design patterns promised by their description. In some cases the description must become more complex and nuanced to make implementation feasible.

Range-bounded types are a good example of types which can be succinctly *described* but face complications when being concretely implemented. They are therefore a good example of the potential contrast between *possible* and *actual* types. I will examine this distinction in more detail and then return to range-bounded types as a demonstrative example.

3.3 Described Types and Actual Types

The notion of "type systems" I adopt here sees types not logical abstractions but as engineered artifacts (as are languages themselves). A logical description of a plausible type — say, the type of all functions that take equal-sized lists of integers — may not correspond to a type that can be concretely implemented in a given programming languages and environment. There may be *contingently* uninhabited types, types which cannot be inhabited *in some particular computing environment* because there are no constructors implemented; or because the environment has no compile-time or run-time mechanism for enforcing the requirements stipulated by the type — insofar as they are used either to describe values or as an element in typing judgments.[7]

In this context, then, a *type* is conceptually a set of guarantees on function-call resolution (for overloaded function symbols) and gatekeeping (for preventing code from executing with unwarranted assumptions), and a type can only be inhabited if those guarantees can be met. In particular, the "witness" to a type's being inhabited is always one (or more) functions — either a constructor (a "value constructor") which creates a value of the type from other values or from character-string literals; or, for function-like types, an implemented procedure.

A consequence of this framing is that defining what exactly constitutes a type — via an expression notating a type description and a corresponding type implementation — depends intrinsically on defining what constitutes a *function*, and particularly an *implementation* or *function body*. Moreover, since functions are implemented in terms of other functions, another primordial concept is a function *call*. Reasoning abstractly about functions needs to be differentiated from reasoning about available, implemented functions.[8] Consider function pointers: what is the address of $f \circ g$ if that expression is interpreted in and of itself as evaluating to a functional value? This suggests that a composition operator does not work in function-like types quite like arithmetic operators in numeric types (which is not unexpected insofar as functional values, internally, are more like pointers than numbers-with-arithmetic).[9] To put it differently, an **address-of** operator *may* be available for $f \circ g$ if it is available for **f** and **g**, but this depends on language design; it is not an abstract property of type systems.

A similar discussion applies to "Currying" — the proposal that types $\mathcal{T}_1 \to \mathcal{T}_2 \to \mathcal{T}_3$ and $\mathcal{T}_1 \to (\mathcal{T}_2 \to \mathcal{T}_3)$ are equivalent, in that fixing one value as argument to a binary function yields a new unary function. Again, since the Curried function is not necessarily implemented, there is a *modal* difference between $\mathcal{T}_1 \to \mathcal{T}_2 \to \mathcal{T}_3$ and $\mathcal{T}_1 \to (\mathcal{T}_2 \to \mathcal{T}_3)$. Languages *may* be engineered to silently Curry any function on demand, but purported $\mathcal{T}_1 \to \mathcal{T}_2 \to \mathcal{T}_3$ and $\mathcal{T}_1 \to (\mathcal{T}_2 \to \mathcal{T}_3)$ equivalence is not a *necessary* feature of type systems.

To the extent that both mathematical and programming concepts have a place here, we find a certain divergence in how the word "function" is used. If I say that "there exists a function from $\mathcal{T}_1$ to $\mathcal{T}_2$", where $\mathcal{T}_1$ and $\mathcal{T}_2$ are (not necessarily different) types, then this statement has two possible interpretations. One is that, mathematically, I can assume the existence of a $\mathcal{T}_1 \Rightarrow \mathcal{T}_2$ mapping by appeal to some sort of logic; the other is that a $\mathcal{T}_1 \Rightarrow \mathcal{T}_2$ function actually exists in code. This is not just a "metalanguage" difference pro-

[7] Similar issues are sometimes addressed by a *modal* type theory (cf., e.g., [**?**]) where (in one interpretation) a *logical* assertion about a type may be *possible* but not necessary (the modality ranging over "computing environments", which act like "possible worlds").

[8] As a case in point, a common functional-programming idiom is to treat the composition of two unary functions as itself a function-typed value to pass to contexts expecting other function-typed values. In my perspective here, $f \circ g$ may be a *plausible* value, but it is not an *actual* value without being implemented, whether via a code graph (spelling out the equivalent of $\lambda x.fgx$) or some indirect/behavioral description (analogous to **inc** represented as ⟨**&add, 1**⟩).

[9] Of course, languages are free to implement functions behind the scenes to expand (say) $f \circ g$, but then $f \circ g$ is just syntactic sugar (even if its purpose is not just to neaten source code, but also to inspire programmers toward thinking of function-composition in quasi-arithmetic ways).

jected from how the discourse of mathematical type theory is used to different ends than discourses about engineered programming languages, which are social as well as digital-technical artifacts. Instead, we can make the difference exact: when a function-value is keyed to a procedure, it is bound to a segment of code subject to analysis and to alternative representations (such as code graphs).

Initializing function-values from code-segments is, roughly, analogous to initializing simpler types from source-code literals. Typically — see item 1 on page 27 — procedures are defined from aggregate code-spans with semantically and syntactically regulated internal structure. I assume that code is written in a specific language and that, in the context of that language, any sufficiently complete code-span can be compiled to an Abstract Semantic Graph (or similar graph-like Intermediate Representation). The logic of this representation may vary among languages and/or type systems — optimal graph representation of source code is an active area of research, not only for compiler technologies but also for code analyzers and "queries" and for code deployment on the Semantic Web. In this chapter, I assume that an adequate graph representation will be isomorphic to a Directed Hypergraph. So, assume that there is a "code-DH" type such that every code-span suitable for compilation into a function-implementation is a value of that type. Assume also that every source code literal is a **code**-DH graph with one hypernode and no edges. In that case, initializing carriers from literals is one example of initializing carriers from code-graph instances more generally, including initializing "function" values.

This approach — using DH or CH graphs as the formal ground of type-theoretic statements — influences how we can analyze types. For every "function-like" type, instances of the type are given through implementations suitable to graph representation. Many nodes in these graphs represent values which the function receives from and/or passes to other functions. Therefore, assertions about functions' behavior often take the form of assertions about values functions receive from other functions, and conditions for their properly sending values to other functions in turn. Insofar as we seek to express conditions on functions' behavior through a type system, we can then interpret type systems as *leveraging* taxonomies of node-to-node relations — modeled via Source Code Ontologies — to define notations where descriptions of functions' *behavior* can be interpreted or converted to descriptions of types themselves.

Notice that one single literal may initialize carriers of multiple types; the number "0" could become a signed or unsigned integer, a float, etc. Similarly, a code-span can be compiled multiples times, as in C++ templates. So, there is not necessarily a one-to-one correspondence between code-graphs and function values. Nevertheless, we can assume that each function implementation is uniquely determined by the function type together with its function-body implementation-graph, whose potential "template parameters" are fixed according to the produced type. So, each function *implementation* is fully determined by a type-and-code-graph pair. This formally expresses how *implemented functions* are different phenomena than what we might call "functions" in mathematics. If there is a meaningful type Category than we must have nontrivial morphisms, which I would argue should be those that abstractly capture type-level semantics, such as predefined conversion operators or the "initialization" morphism. But morphisms are not affixed to the code-graph and value-constructor machinery, though of course some morphisms may coincide with implemented functions.

Given this distinction, we can start to explore why some advanced constructs in Dependent Type Theory, or other features of very expressive type systems, may be hard to implement in practice. I suggested earlier in this section that "range-bounded" types are a good case-study in implementational complications that can befall described types; I will now return to that discussion and pursue the "ranged-type" case further.

3.4 *Range-Bounded Types, Value Constructors, and Addressability*

Consider a function **f** which takes values that must be in a specified range ⟮**r**⟯ (say, an integer between ⟮**0,100**⟯). By extension, suppose we want to overload **f** based on whether its argument (say, **x**) falls inside or outside ⟮**r**⟯. This is not hard to achieve if **f**'s *type* internally references a *fixed* ⟮**r**⟯. Let **T** be a symbol that quantifies over the typeclass of types with magnitude/comparison operators, and **ranged**<**T**> be a type formed from **T** and two **T**-values, implementing the semantics of closed intervals over **T**: so **ranged**<**T, t1, t2**> is a "type-expression" mapped to a type constructor yielding a single type (not a type family, typeclass, or higher-order type).

For any type-with-intervals **T** and **T**-range ⟮**r**⟯, a compiler (for instance by specializing a template) can produce a

value constructor that takes **T**-values and tests (or coerces) them such that the constructor only returns a value in ⟨**r**⟩. This can then be the input type for a function which requires ⟨**r**⟩-bounded input. What to do when a values *fails* the range-test is another question which I set aside for now. We can similarly define a "non-range" type which only accepts values *not* in ⟨**r**⟩; and, since these two types (the in-range and the out-of-range) are different, we can overload **f** on them. So, assuming we accept ⟨**r**⟩ being fixed, we can achieve something semantically — "overload-wise" — like dependent typing. Of course, Dependent Types as a programming construct are more powerful than this: an example of "real" dependent typing would be something like an **f** which takes *two* arguments: the first a range, and the second a value within the range. We want the type system to be engineered allowing the condition on the second argument to be verified as part of *typechecking*.

Using the ⟨**r**⟩-type as before, the type of **f**'s second parameter would then be **T** restricted to the ⟨**r**⟩ interval, but here ⟨**r**⟩ is not fixed in **f**'s declaration but rather passed in to **f** as a parameter; the type of the second parameter depends on the *value* of the first one. Unless we know *a priori* that only a specific set of ⟨**r**⟩s in the first parameter will ever be encountered, how should a compiler identify the value constructor to use for **x**? This evidently demands either that a value constructor be automatically created at runtime, for each ⟨**r**⟩ encountered — so, i.e., that the compiler has to insert some runtime mechanism which creates and calls a value constructor for **x** before **f**'s body is entered — or else that a single value constructor is used for all **x**s, regardless of ⟨**r**⟩. As I argued, each (concretely) inhabitable type has at least one associated acyclic value constructor which is unique to that type: a type-plus-code-graph pair. This allows one code graph to be mapped to multiple value constructors. But it does not allow one value constructor to service many types, although we can implement functions that would be semantically analogous to such a value constructor.

We could certainly write a function that takes a range and a value and ensures that the value fits the range — perhaps by throwing an exception if not, or mapping the value to the closest point in the range. Such a function would provide common functionality for a family of constructors each associated with a given range. But a function (**Cf**, say) providing "common functionality" for value constructors is not necessarily itself a value constructor. If we'd want to treat such a function as a *real* value constructor we'd have to add contextual modifiers: **Cf** is a value constructor for range-type ⟨**r**⟩ when ⟨**r**⟩ as a range is supplied as one parameter. The value constructor itself would have to be dependently typed, its result type varying with the value of its arguments — but a result-type-polymorphic value constructor is no longer an actual value constructor; at best we can say it is a function which can dynamically *create* value constructors. In the present case, Currying **Cf** on any given ⟨**r**⟩ probably does yield a bonafide value constructor, but a function which when Curried yields a value constructor is not, or at least not necessarily, a value constructor itself.

It appears that language designers — at least considering pureblood Dependent Types — have two options: either modify the notion of value constructor such that one *true* value constructor is understood as a possible constructor for multiple types, and on behalf of which type it is constructing is something dynamically determined at runtime; or value constructors are allowed to be transient values created and recycled at runtime. This is not just an internal-implementation question because value-constructors also need to be *exposed* for reflection (which in turn involves some notion of addressability: the most straightforward reflection tactic is to maintain a map of identifiers to function addresses). Either option complicates the relation between types' realizability and their value constructor: instead of each inhabitable type having at least one value constructor which is itself a value, and as such itself results from a value constructor taking a code graph, we have to associate types either with dynamically created temporary value-constructor values or we have to map value constructors not to singular values but to a compound structure. For example, if the purported value constructor for a range type **T**⟨**r**⟩ is to be the "common functionality" base function *plus* a range-argument to be passed to it — some sort of ⟨**&Cf,** $\mathbf{r}_1$**,** $\mathbf{r}_2$⟩ compound data structure, again by analogy to **inc** and ⟨**&add, 1**⟩ — then the "value" of the value constructor no longer has a single part, but becomes a function-and-range pair. Let me dub this the "metaconstructor" problem: what are allowable *value constructors for the value constructors* of allowable types?

If we ignore templates, a reasonable baseline assumption is that "metaconstructors" must only accept one sort of argument: code graphs. That is, for each metaconstructor — again, a value constructor whose result is a value constructor whose result is a value of some type **T** — there must be exactly one code-span notating the value constructor's implementation. As I just outlined, dependent typing can complicate the

picture because metaconstructors then have to have possible alternative signatures: e.g. the value constructor for "integers between zero and one hundred (inclusive)" has to combine a "common functionality" function body with another part that specifies the desired ⟨**0, 100**⟩ range. If we *don't* ignore templates, we can speculate that each actual metaconstructor is a specialization of a template, so each one goes back to the one-argument-code-graph signature — but we then have an entire family of metaconstructors (or possible metaconstructors) which share functionality and differ only according to a criterion that varies over values of a type. Consider just the simpler case of integer ranges with lower bound zero: for any i of an integer type (64-bit unsigned, say) there is a reasonable type of **int**s $\leq i$. The collection of "reasonable" types formed in this manner is therefore co-extensive with **int** itself. But on both philosophical and practical grounds, we may argue that "reasonable" types are not the same thing as types *full stop*.

Philosophically, programming types lie at the intersection of mathematics and human concepts: a datatype typically avatars in digital environments some human concepts. There are particular arithmetic intervals that have legitimate conceptual status: let's say, ⟨**0, 100**⟩ for percentages; the maximum speed of a car; the dial range of a thermostat. So, conceptually, we can implement an abstract family of range types which might be concretized for a handful of conceptually meaningful specializations. Moreover, we can conceptualize a general-purpose structure which is a range ⟨**r**⟩ together with a range-bounded value, but then we are conceptualizing *one* type, not a whole family of types. So the basic "Ontology" of Dependent Types — of whole type-families indexed over values of some other type — does not correspond with the nature of concepts: while there is a *reasonable type* for intervals ⟨**0, n**⟩ for any n, there is not necessarily a corresponding *concept*, *a priori* (similarly, we have a capacity to conceptualize any number — assuming it has some distinct conceptual status, like "the first nontrivial Fourier coefficient of the j-function" — but reasonably we do not have a distinct concept for every number).

Meanwhile, practically, it is not computationally feasible to have an exponential explosion in the order of *actual* types — such as, one unique type for each 64-bit integer. For example, it is reasonable for a language engine to assume that most function values support an address-of operator. This is one property whereby function values differ from, say, integers: we cannot take the address of the number **5** (by contrast, we *could* form a pointer to a file-scoped C function that just trivially returns **5**). But allowing type families to be indexed on 64-bit integers *and* providing a distinct address for each such type's value-constructors would be mathematically equivalent to providing a unique address for each 64-bit integer.

A reasonable language, conversely, may have "non-addressable" function values: for example, suppose a lambda passed to a function is defined just via an operator, like an $f \circ g$. Say, sorting two lists of strings on a comparison which calls a "to lowercase" function before invoking a less-than operator. This could be notated with a lambda block, but some languages may allow a more "algebraic" expression, something meaning "lower-case then less-than", with the idea that function values can be composed by rough analogy to numbers being added (see item 3 on page 28). In this case, the *value constructor for the function type* does not take a code graph, at least not one visible near the $f \circ g$ expression, just as the value constructor for **x** in **x** = **y** + **z** is hidden somewhere in the "**y** + **z**" implementation. A language can reasonably forbid taking the address of (or forming pointers to) "temporary" function values derived algebraically from other functions. Indeed, the concepts of "constructed from a code graph" and "addressable" may coincide: a compiler may allocate long-term memory for just those function-implementations it has compiled from code-graphs.

But value-constructors are not just any function-value: they have a privileged status vis-à-vis types, and may be invoked whenever an appropriately-typed value is used. Allowing large type families (like one type for each **int** — similar to "inductive typing" as discussed by Edwin Brady in the context of the Idris language [**?**, p. 14]) effectively forces a language to accept non-addressable value-constructors. Conversely, forcing value constructors to be addressable prohibits "large" type families — like types indexed over other (non-enumerative) types — at least as *actual* types. A language engine may declare that value constructors, in short, cannot be "temporary" values. This apparently precludes full-fledged Dependent Types, since dependent-typed values invariably require in general some extra contextual data — not just a function-pointer — to designate the desired value constructor at the point where a value, attributed to the relevant dependent type, is needed. It may be infeasible to add the requisite contextual information at every point where a dependent-typed value has to be constructed — unless, perhaps, a description of the context can be packaged and carried around with the value, sharing the value's lifetime.

A value can, indeed, actually be an aggregate data structure including functions to call when the value is created or modified — behaving as if it were dependently typed — but this is more a matter of one type supporting a range of different behaviors, rather than a family of distinct types. A single range-plus-value type can behave *as if* it were actually instantiating a type belonging to a family where every possible range corresponds to a different type — at least with respect to value constructors and accessors, which can implement hidden gatekeeping code. But the type is still just one type from the point of view of overloading: the behavioral constraints are code evaluated behind-the-scenes at runtime, and cannot in themselves be a basis for compile-time overload decisions. In other words, they are more like *typestate* than type families.

Consider a function to remove the *n*th value from a **list**. For this to work properly, the *n* has to be less than the size of **list**; i.e., it has to be in the range **⟨0, list.size()-1⟩**. The relevant range-expression *looks* like the example I used earlier — **int⟨0,100⟩** — but in place of a *fixed* **0-100** range, here we have a range that can potentially be different each time the function is called (assuming each **list** can be a different size). So while it may be a well-formed type-expression to say that *n* has *type* **int⟨0, list.size()-1⟩**, the net result is that *n*'s type is then not known until runtime. Since its type is not known, nor is the proper value constructor to call when *n* has to be provided to a **lambda** channel, at least not *a priori*. Instead, the value constructor has to be determined on-the-fly. As such, the "constructed" value constructor acts like a kind of supplemental function called prior to the main function being called. But there are several ways of arranging for such gatekeeping functions to be called, apart from via explicitly declaring types whose value constructors implement the desired functionality. In the current example, there are ways to ensure that a gateekeping function is called whose runtime checks mimic the **int⟨0, list.size()-1⟩** value constructor without actually stipulating that *n*'s *type* is **int⟨0, list.size()-1⟩**. Some of these involve typestate, which I will now review briefly. Other options will be discussed later in the chapter.

3.5 *Dependent Types and Typestate*

Typestates are finer-grained classifications than types. However, just as it is "reasonable" to consider each range as its own type — in the sense that a coherent TXL should allow range-value types, even if in practice a language may limit how such types are actualized — it is also "reasonable" to factor typestates into types as well. A canonical example of typestate is restricting how functions are called which operate on files. A single "file" type actually covers several cases, including files that are open or closed, and even files that are nonexistent — they may be described by a path on a filesystem which does not actually point to a file (perhaps in preparation for creating such a file). Instead of *one* type covering each of these cases, we can envision *different* types for nonexistent, closed, or open files. With these more detailed types, constraints like "don't try to create an already-existing file" or "don't try to modify a closed or nonexistent file" are enforced by type-checking.

While this kind of gatekeeping is valuable in theory, it raises questions in practice. Reifying "cases" — i.e., *typestates* like open, closed, or nonexistent — to distinct *types* implies that a "file" value can go through different types between construction and destruction. If this is literally true, it violates the convention that types are an intrinsic and fixed aspect of typed values. It is true that, as part of a type cast, values can be reinterpreted (like treating an **int** as a **float**), but this typically assumes a mathematical overlap where one type can be considered as subsumed by a different type for some calculation, *without this changing anything*: any integer is equally a ratio with unit denominator, say. "Casting" a closed file to an open one is the opposite effect, using disjunctures between types to capture the fact that state *has* changed; to capture a trajectory of states for one value — which must then have different types at different times, since this is the whole point of modeling successive states via alternations in type-attribution.

An alternative interpretation is that the "trajectory" is not a single mutated value but a chain of interrelated values, wherein each successive value is obtained via a state-change from its predecessor. But a weakness of this chain-of-values model is that it assumes only one value in the chain is currently correct: a file can't be both open and closed, so if one value with type "closed file" is succeeded by a different value with type "opened file", the latter value will be correct only if the file was in fact opened, and the former otherwise — but a compiler can't know which is which, *a priori*. Or, instead of a "chain" of differently-typed values we can employ a single general "file" type and then "cast" the value to an "open file" type when a function needs specifically an *open* file, and so forth. The effect in that case is to insert the cast operator

as a "gatekeeper" function preventing the function receiving the casted value from receiving nonconformant input. Again, though, the compiler cannot make any assumptions about whether the "casts" will work (e.g., whether the attempt to open a file will succeed).

A good real-world example of the overlap between Dependent Types and typestate (also grounded on file input/output) comes from the "Dependent Effects" tutorial from the Idris (programming language) documentation [**?**]:

> A practical use for dependent effects is in specifying resource usage protocols and verifying that they are executed correctly. For example, file management follows a resource usage protocol with ... requirements [that] can be expressed formally in [Idris] by creating a **FILE_IO** effect parameterised over a file handle state, which is either empty, open for reading, or open for writing. In particular, consider the type of [a function to open files]: This returns a **Bool** which indicates whether opening the file was successful. The resulting state depends on whether the operation was successful; if so, we have a file handle open for the stated purpose, and if not, we have no file handle. By case analysis on the result, we continue the protocol accordingly. ... If we fail to follow the protocol correctly (perhaps by forgetting to close the file, failing to check that open succeeded, or opening the file for writing [when given a read-only file handle]) then we will get a compile-time error.

So how does Idris mitigate the type-vs.-typestate conundrum? Apparently the key notion is that there is one single **file** *type*, but a more fine-grained type-*state*; and, moreover, an *effect system "parametrized over" these typestates*. In other words, the *effect* of **file** operations is to modify *typestates* (not types) of a **file** value. Moreover, Dependent Typing ensures that functions cannot be called sequentially in ways which "violate the protocol", because functions are prohibited from having effects that are incompatible with the potentially affected values' current states. This elegant syntheses of Dependent Types, typestate, and Effectual Typing brings together three of the key features of "fine-grained" or "very expressive" type systems.

But the synthesis achieved by Idris relies on Dependent Typing: typestate can be enforced because Idris functions can support restrictions which *depend* on values' current typestate to satisfy effect-requirements in a type-checking way. In effect, Idris requires that all possible variations in values' unfolding typestate are handled by calling code, because otherwise the handlers will not type-check. An analogous tactic in a language like C++ would be to provide an "open file" function only with a signature that takes two callbacks, one for when the **open** succeeds and a second for when it fails (to mimic the Idris tutorial's "case analysis"). But that C++ version still requires convention to enforce that the two callbacks behave differently: via Dependent Types Idris can confirm that the "open file" callback, for example, is only actually supplied as a callback for files that have actually been opened. A better C++ approximation to this design would be to cast files to separate types — not only typestates — after all, but only when passing these values to the callback functions; this is similar to the approach I endorse here to approximate Dependent Typing via (sometimes hidden) channels rather than constructs (like typed-checked typestate-parametrized effects) which require a language to implement a full Dependent Type system.

In the case of Idris, Dependent Types are feasible because the final "reduction" of expressions to evaluable representations occurs at runtime. In the language of the Idris tutorial:

> In Idris, types are first class, meaning that they can be computed and manipulated (and passed to functions) just like any other language construct. For example, we could write a function which computes a type [and] use this function to calculate a type anywhere that a type can be used. For example, it can be used to calculate a return type [or] to have varying input types.

More technically, Edwin Brady (and, here, Matúš Tejiščák) elaborate that

> *Full-spectrum* dependent types ... treat types as first-class language constructs. This means that types can be built by computation just like any other value, leading to powerful techniques for generic programming. Furthermore, it means that types can be parameterised on values, meaning that strong, explicit, checkable relationships can be stated between values and used to verify properties of programs at compile-time. This expressive power brings new challenges, however, when compiling programs. ... The challenge, in short, is to identify a phase distinction between compile-time and run-time objects. Traditionally, this is simple: types are compile-time

> only, values are run-time, and erasure consists simply of erasing types. In a dependently typed language, however, erasing types alone is not enough [**?**, page 1].

To summarize, Idris works by "erasing" some, but not all, of the extra contextual detail needed to ensure that dependent-typed functions are used (i.e., called) correctly (see also [**?**], and [**?**, page 210]). This means that a lot of contextual detail is *not* erased; Idris provides machinery to join executable code and user specifications onto *types* so that they take effect whenever affected types' values are constructed or passed to functions. Despite a divergent technical background, the net result is arguably not vastly different from an Aspect-Oriented approach wherein constructors and function calls are "pointcuts" setting anchors upon source locations or logical run-points, where extra code can be added to program flow (see e.g. [**?**], [**?**], [**?**]). Recall my contrast of "internalist" and "externalist" paradigms, sketched at the top of this chapter: Aspect-Oriented Programming involves extra code added by external tools (that "modify" code by "weaving" extra code providing extra features or gatekeeping). Implementations like Idris pursue what often are in effect similar ends from a more "internalist" angle, using the type system to host added code and specifications without resorting to some external tool that introduces this code in a manner orthogonal to the language proper. But Idris relies on Haskell to provide its operational environment; it is not clear how Idris's strategies (or those of other Haskell and ML-style Dependent Type languages) for attaching runtime expressions to type constructs would work in an imperative or Object-Oriented environment, like C++ as a host language.

This discussion emanated from Idris examples based on file-management typestate, but it generalizes to other cases, such as range-delimited types or states. Practically, working with scenarios like range-bounded values — which in principle exemplifies programming tasks where Dependent Types can be a useful idiom — in practice arguably ends up more like typestate management as exemplified by gatekeeping vis-à-vis, say, files (only call *this* function if *that* file is open). A range-interval is typestate-like in that the practical intent is to affix certain gatekeeper functions to an accompanying value so that it will never be incorrectly used — for instance, that it will never be modified to lie outside its assigned range. Conforming to a range restriction is more like a typestate than a type: indeed, a range-plus-value pair has an obvious covering via two typestates (the value is either in or out of its closed interval).[10] So the semantics of Dependent Types can practically be captured via a typestate framework — and perhaps vice-versa, since typestates can be seen as type families indexed over (possibly enumerative) types; file typestates as a family indexed over ⟨**closed**, **open**, **nonexistent**⟩, for example.

Adding validation code at runtime — to dynamically enforce typestate constraints — fits the profile of Aspect-Oriented Programming more than type-system expressiveness, however. I propose therefore to outline a framework which in its engineering works effectively by code-insertion but expresses a more type-theoretic orientation. To begin, recall that the implementational problem with Dependent Types is maintaining entire families of value constructors; but we can perform computations to assess whether a value meeting stated criteria *could* be constructed without actually constructing the value. In this spirit, assume that value constructors can potentially be associated with *preconstructors* which run before the constructor proper and assess whether the proposed construction is reasonable. These preconstructors may be binary-valued but may also have a larger result type, such as a typestate enumeration. Given a range-bounded type, for example, the preconstructor can classify a value as *in range*, *too small*, or *too large*, returning an enumerated value which the actual constructor then uses to refine its behavior. The key point about preconstructors is that they can be used even without a value constructor present to receive its value: instead, the preconstructor can test that a value of a narrowly defined type *could* be constructed, even if the actual value used belongs to a broader type.

Similar to preconstructors, I will also introduce a concept of *co-channels*, which are "behind the scenes" channels that create values from functions' channels, but whose values are passed to functions implicitly; they are not visible at function-call sites. I will return to the discussion of Preconstructors and Co-channels after developing a theory of Channel Complexes more substantially.

4 Channels and Carriers

Suppose one procedure calls a second. From a high level perspective, this has several consequences that can be semantic-graph represented — among others, that the calling procedure depends on an implementation of the callee being available —

[10] Although a more detailed range typestate might have a few more enumeration values: representing "on the border" or distinguishing "too large" from "too small".

but at the source code level the key consequence is that a node representing source tokens which designate functional values enters into different semantic relations (modeled by different kinds of edge-annotations) than nodes marking other types of values and literals. Suppose we have an edge-annotation that x is a value passed to f; this graph is only semantically well-formed if f's representatum has functional type (by analogy to the well-formedness criteria of $\lambda x.fx$).

This motivates the following: suppose we have a Directed Hypergraph, where the nodes for each hyper-edge represent source-code tokens (specifically, symbols and literals). Via the relevant Source Code Ontology, we can assert that certain edge-annotations are only possible if a token (in subject or object position) designates a value passed to a function. From the various edge-annotation kinds which meet this criteria, we can define a set of "channel kinds".

This implicitly assumes that symbols "hold" values; to make the notion explicit, I will say that symbols are *carriers* for values. Carriers do not necessarily hold a value at every point in the execution of a program; they may be "preinitialized", and also "retired" (the latter meaning they no longer hold a meaningful value; consider deleted pointers or references to out-of-scope symbols). A carrier may pass through a "career" from preinitialized to initialized, maybe then changing to hold "different" values, and maybe then retired.[11] I assume each carrier is associated with a single type throughout its career, and can only hold values appropriate for its type.[12]

In short, *carriers* embody the contrast between abstract or mathematical type theory and practical languages' type systems. Instead of the notion of a *typed value* — an instance of a type — we can focus on carriers which are tangible elements of source code and also (during runtimes) binary resources. Carriers evince different states; in those states where they hold a concrete value, carriers play a conceptual role analogous to type-instances in formal type theory. On the other hand, carriers can have other states which are orthogonal to type systems: carriers holding *no* value, for example, are different carriers that *are* holding values of a "" type.

Having introduced the basic idea of "carriers" I will now consider carrier operations in more detail, before then expanding on the theory of carriers by considering how carriers group into channels.

4.1 Carriers and Careers

In this theory, carriers are the basic means by which values are represented within computer code, including representing the "communication" of values between different parts of code source, such as calling a function. The "information flow" modeled by a function-call includes values held by carriers at the function-call-site being transferred to carriers at the function-implementation site. This motivates the idea of a "transfer" of values between carriers, a kind of primitive operation on carriers, linking disparate pieces of code. It also illustrates that the symbols used to name function parameters, as part of function signatures, should be considered "carriers" analogous to lexically-scoped and declared symbols.

Taking this further, we can define a *channel* as a list of carriers which, by inter-carrier transfers, signify (or orchestrate) the passage of data into and out of function bodies (note that this usage varies somewhat from process calculi, where a channel would correspond roughly to what is here called a single carrier; here channels in the general case are composed of multiple carriers). I'll use the notation $\looparrowright$ to represent inter-carrier transfer: let $\mathfrak{C}_1$ and $\mathfrak{C}_2$ be carriers, then $\mathfrak{C}_1 \looparrowright \mathfrak{C}_2$ is a transfer "operator" (note that $\looparrowright$ is non-commutative; the "transfer" happens in a fixed direction), marking the logical moment when a value is moved from code-point to code-point. The $\looparrowright$ is intended to model several scenarios, including "forced coercions" where the associated value is modified. Meanwhile, without further details a "transfer" can be generalized to *channels* in multiple ways. If $\mathfrak{C}_1$ and $\mathfrak{C}_2$ are carriers which belong to two channels (χ_1, χ_2), then $\mathfrak{C}_1 \looparrowright \mathfrak{C}_2$ elevates to a transfer between the channels — but this needs two indices to be concrete: the notation has to specify which carrier in χ_1 transfers to which carrier in χ_2. For example, consider the basic function-composition

[11] Because "unitialized" carriers and "dangling pointers" are coding errors, within "correct" code, carriers and values are bound tightly enough that the whole carrier/-value distinction might be considered an artifact of programming practice, out of place in a rigorous discussion of programming languages (as logicomathematical systems, in some sense). But even if the "career" of symbols is unremarkable, we cannot avoid in some contexts — within a debugger and/or an IDE (Integrated Development Environment, software for writing programs), for example — needing to formally distinguish the carrier from the value which it holds, or recognize that carriers can potentially be in a "state" where, at some point in which they are relevant for code analysis or evaluation, they do not yet (or do not any longer) hold meaningful values. Consequently, the "trajectory" of carrier "lifetime" — from being declared, to being initialized, to falling "out of scope" or otherwise "retired" — should be integrated into our formal inventory of programming constructs, not relegated to an informal "metalanguage" suitable for discussing computer code as practical documents but not as formal systems.

[12] The variety of possible careers for carriers is not directly tied to its type: a carrier which cannot change values (be reinitialized) is not necessarily holding a CONST-typed value.

$f \circ g$: **(f.g)(x) = f(g(x))**. The analogous "transfer" notation would be, say, **g:return**$_1 \looparrowright$**f:lambda**$_1$: here the first carrier in the **return** channel of **g** transfers to the first carrier in the **lambda** channel of **f** (the subscripts indicate the respective positions).

Most symbols in a function body (and corresponding nodes in a semantic graph) accordingly represent carriers, which are either passed in to a function or lexically declared in a function body. Assume each function body corresponds with one lexical scope which can have subscopes (the nature of these scopes and how they fit in graph representation will be addressed later in this section). The *declared* carriers are initialized with values returned from other functions (perhaps the current function called recursively), which can include constructors that work on literals (so, the carrier-binding in source code can look like a simple assignment to a literal, as in **int i = 0**). In sum, whether they are passed *to* a function or declared *in* a function, carriers are only initialized — and only participate in the overall semantics of a program — insofar as they are passed to other functions or bound to their return values.

Furthermore, both of these cases introduce associations between different carriers in different areas of source code. When a carrier is passed *to* a function, there is a corresponding carrier (declared in the callee's signature) that receives the former's value: "calling a function" means transferring values between carriers present at the site of the function call to those present in the function's implementation. Sometimes this works in reverse: a function's return may cause the value of one of its carriers to be transferred to a carrier in the caller (whatever carrier is bound to the caller's return value).

Let $\mathfrak{C}_1$ and $\mathfrak{C}_2$ be two carriers. The $\looparrowright$ operator (representing a value passed from $\mathfrak{C}_1$ to $\mathfrak{C}_2$) encompasses several specific cases. These include:

1. Value transfer directly between two carriers in one scope, like **a = b** or **a := b**.

2. A value transferred between one carrier in one function body when the return value of that function is assigned to a carrier at the call site, as in **y = f(x)** when **f** returns with **return 5**, so the value **5** is transferred to **y**.

3. A value transferred between a carrier at a call-site and a carrier in the called function's body. Given **y = f(x)** and **f** declared as, say, **int f(int i)**, then the value in carrier **x** at the call-site is transferred to the carrier **i** in the function body. In particular, every node in the called function's code-graph whose vertex represents a source-code token representing symbol **i** then becomes a carrier whose value is that transferred from **x**.

4. A value transferred between a **return** channel and either a **lambda** or **sigma** channel, as part of a nested expression or a "chain of method calls". So in **h(f(x))**, the value held by the carrier in **f**'s **return** channel is transferred to the first carrier in **h**'s **lambda**. An analogous **return**$\looparrowright$**sigma** transfer is seen in code like **f(x).h()**: the value in **f**'s **return** channel becomes the value in **h**'s **sigma**, i.e., its "**this**" (we can use $\looparrowright$ as a notation between *channels* in this case because we understand the Channel Algebra in force to restrict the size of both **return** and **sigma** to be at most one carrier).

Let $\mathfrak{C}_1 \rightarrowtail \mathfrak{C}_2$ be the special case of $\looparrowright$ corresponding to item (3): a transfer effectuated by a function call, where $\mathfrak{C}_1$ is at the call site and $\mathfrak{C}_2$ is part of a function's signature. If f_1 calls f_2 then $\mathfrak{C}_1$ is in f_1's context, $\mathfrak{C}_2$ is in f_2's context, and $\mathfrak{C}_2$ is initialized with a copy of $\mathfrak{C}_1$'s value prior to f_2 executing. A *channel* then becomes a collection of carriers which are found in the scope of one function and can be on the right hand side of an $\mathfrak{C}_1 \rightarrowtail \mathfrak{C}_2$ operator.[13]

To flesh out Channels' "transfer semantics" further, I will refer back to the model of function-implementations as represented in code graphs. If we assume that all code in a computer program is found in some function-body, then we can assume that any function-call operates in the context of some other function-body. In particular, any carrier-transfer caused by a function call involves a link between nodes in two different code graphs (I set aside the case of recursive functions — those which call themselves — for this discussion).

So, to review my discussion in this section so far, I started with the process-algebraic notion of inter-procedure combinations; but whereas process calculi enlarge this notion by distinguishing different syntheses of procedures (such as, concurrent versus sequential), I have focused instead on the

[13] In general, we might say that two carriers are "convoluted" if there is a value passed between them or if their values somehow become interdependent (as an example of interdependence without direct transfer, consider tests that that two lists are the same size, or two numbers monotone increasing, as a runtime disambiguation of dependent-typed polymorphic implementations). Depending on context, convolution can refer to a structure in program graphs in the abstract or to an event in program execution: modeling a program as a series of discrete points in time — each point inhabited by a small change in program state — two carriers are convoluted at the time-point where a value-transfer occurs (or the steps toward some kind of gatekeeping check get initiated).

communication of values between procedures. The semantics of inter-procedure value-transfers is unexpectedly detailed, because it has to recognize the possibility of nontrivial copy semantics, casts, overloading, and perhaps "gatekeeping". Furthermore, in addition to these semantic details, analysis of value-transfers is particularly significant in the context of Source Code Ontologies and RDF or Directed Hypergraph representations of computer code. This is because code-graphs give us a rigorous foundation for modeling computer programs as sets of function-implementations which call one another. Instead of abstractly talking about "procedures" as conceptual primitives, we can see procedures as embodied in code-graphs (and function-values as constructed from them, which I emphasized last section). "Passing values between" procedures is then explicitly a family of relationships between nodes (or hypernodes) in disparate code-graphs, and the various semantic nuances associated with some such transfers (type casts, for example) can be directly modeled by edge-annotations. Given these possibilities, I will now explore further how the framework of *carriers* and *channels* fits into a code-graph context.

4.2 *Channel Complexes, Code Graphs, and Carrier Transfers*

For this discussion, assume that f_1 and f_2 are implemented functions with code graphs Γ_1 and Γ_2, respectively. Assume furthermore that some statement or expression in f_1 involves a call to f_2. There are several specific cases that can obtain: the expression calling f_2 may be nested in a larger expression; f_2 may be called for its side effects alone, with no concern to its return value (if any); or the result of f_2 may be bound to a symbol in f_1's scope, as in **y = f(x)**. I'll take this third case as canonical; my discussion here extends to the other cases in a relatively straightforward manner.

A statement like **y = f(x)** has two parts: the expression **f(x)** and the symbol **y** to which the results of **f** are assigned. Assume that this statement occurs in the body of function f_1; **x** and **y** are then symbols in f_1's scope and the symbol **f** designates (or resolves to) a function which corresponds to what I refer to here as f_2. Assume f_2 has a signature like **int f(int i)** As such, the expression **f(x)**, where **x** is a carrier in the context of f_1, describes a *carrier transfer* according to which the value of **x** gets transferred to the carrier **i** in f_2's context.

In the terms I used earlier, f_2's signature represents a channel "package" — which, in the current example, has a **lambda** channel of size one (with one carrier of type **int**) and a **return** channel of size one (f_2 returns one **int**). Considered in the context of carrier-transfers between code graphs, a Channel Package can be seen as a description of how two distinct code-graphs may be connected via carrier transfers. When a function is called, there is a channel complex which I'll call a *payload* that supplies values to the channel *package*. In the concrete example, the statement **y = f(x)** is a *call site* describing a channel *payload*, which becomes connected to a function implementation whose signature represents a channel *package*: a collection of transfers $\mathfrak{C}_1 \rightarrowtail \mathfrak{C}_2$ together describe an overall transfer between a *payload* and a *package*.

More precisely, the **f(x)** example represents a carrier transfer whose target is part of f_2's **lambda** channel, which we can notate $\mathfrak{C}_1 \overset{\text{lambda}}{\rightarrowtail} \mathfrak{C}_2$. Furthermore, the full statement **y = f(x)** shows a transfer in the opposite direction: the value in f_2's ***return*** channel is transferred to the carrier **y** in the *payload*. This relation, involving a return channel, can be expressed with notation like $\mathfrak{C}_2 \overset{\text{return}}{\rightarrowtail} \mathfrak{C}_1$. The syntax of a programming language governs how code at a call site supplies values for carrier transfers to and from a function body: in the current example, binding a call-result to a symbol always involves a transfer from a ***return*** channel, whereas handling an exception via code like **catch(Exception e)** transfers a value from a called function's ***exception*** channel. The syntactic difference between code which takes values from **return** and **exception** channels, respectively, helps reinforce the *semantic* difference between exceptions and "ordinary" returns. Similarly, method-call syntax like **obj.f(x)** visually separates the values that get transferred to a "**sigma**" channel (**obj** in this case) from the "ordinary" (**lambda**) inputs, reinforcing Object semantics.

To consolidate the terms I am using: we can interpret both function *signatures* and *calls* in terms of channels. Both involve "carrier transfers" in which values are transferred *to* or *from* the channels described by a function signature. The distinction between functions' "inputs" and "outputs" can be more rigorously stated, with this further background, as the distinction between channels in function signatures which receive values *from* carriers at a call site (inputs), and those *from which* values are obtained as a procedure has completed (outputs).

A Channel Expression Language (CXL) can describe

channels both in signatures and at call-sites. The aggregation of channels generically described by CXL expressions I am calling a *Channel Complex*. A Channel Complex representing a function *signature* I am calling a *Channel Package*, whereas complexes representing a function *call* I am calling a *Channel Payload*. Input channels are then those whose carrier transfers occur in the payload-to-package direction, whereas output channels are the converse.

In addition to the payload/package distinction, we can also understand Channel Complexes at two further levels. On the one hand, we can treat Channel Complexes as found *in source code*, where they describe the general pattern of payload/package transfers. On the other hand, we can represent Channel Complexes *at runtime* in terms of the actual values and types held by carriers as transfers are effectuated prior to, and then after, execution of the called function. Accordingly, each Channel Complex may be classified as a *compile-time* payload or package, or a *runtime* payload or package, respectively. The code accompanying this chapter includes a "Channel Complex library" — for creating and analyzing Channel Complexes via a special Intermediate Representation — that represents complexes of each variety, so it can be used both for static analysis and for enhanced runtimes and scripting.

This formal channel/complex/package/payload/carrier vocabulary codifies what are to some degree common-sense frameworks through which programmers reason about computer code. This descriptive framework (I would argue) more effectively integrates the *semantic* and *syntactic* dimensions of source code and program execution. Computer programs can be understood *semantically* in terms of λ-Calculi combined with models of computation (call-by-value or by-reference, eager and lazy evaluation, and so forth). These semantic analyses focus on how values change and are passed between functions during the course of a running program. From this perspective, source code is analyzed in terms of the semantics of the program it describes: what are the semantic patterns and specifications that can be predicted of running programs on the basis of source code in itself?

At the same time, source code can also be approached *syntactically*, as well-formed expressions of a formal language. From this perspective, correct source code can be matched against language grammars and, against this background, individual code elements (like tokens, code blocks, expressions, and statements) — and their inter-relationships — may be identified.

The theory of Channel Complexes straddles both the semantic and syntactic dimensions of computer code. Semantically, carrier-transfers capture the fundamental building blocks of program semantics: the overall evolving runtime state of a program can be modeled as a succession of carrier-transfers, each involving specific typed values plus code-graph node-pairs, marking code-points bridged via a transfer. Meanwhile, syntactically, how carriers belong to channels — the carrier-to-channel map fixing carriers' semantics — structures and motivates languages' grammars and rules. In particular, carrier-transfers induce relationships between code-graph nodes. As a result, language grammars can be studied through code-graphs' profiles insofar as they satisfy RDF and/or DH Ontologies.

In sum, a DH and/or Semantic Web representation of computer code can be a foundation for both semantic and syntactic analyses, and this may be considered a benefit of Channel Complex representations even if they only restate what are established semantic patterns mainstream programming language — for example, even if they are restricted to a **sigma-lambda-return-exception** Channel Algebra modeled around, say, C++ semantics prior to C++11 (more recent C++ standards also call for a "**capture**" channel for inline "lambda" functions).

At the same time, one of my claims in this chapter is that more complex Channel Algebras can lead to new tactics for introducing more expressive type-theoretic semantics in mainstream programming environments. As such, most of the rest of this section will explore additional Channel Kinds and associated Channel Complexes which extend, in addition to merely codifying, mainstream languages' syntax and semantics.

4.3 *Channels for Dependent Types and Larger-Scale Code Structures*

This chapter has focused on modeling specifications captured entirely via a type system, so at this point I'll return to that topic in the specific context of Dependent Types. Consider first a function which must receive a signed integer no greater than **100**. This is a dependent type in the mild sense that the type expression depends on a value: for the range criterion to be TXL-expressible we must assume that the relevant TXL includes type expressions whose elements can be values which are not themselves types. For instance **ranged_lte** can

be a type class such that **ranged_lte<V>** is a type-expression (with **V** a value, not a type), designating types whose inhabitants all compare $\leq$ to **V**. As I have alluded to, we do not have an *a priori* set-theoretic machinery at the implementation level; our ability to express criteria like "the set of unsigned **int**s $\leq$ **100**" depends on an explicit type implementation — and, in particular, a value constructor. Assuming we have generic programming, we can implement **ranged_lte<V>** via a constructor/initializer which takes an arbitrary (hitherto unchecked) **int** and verifies that it falls in the range $\leq$ **V**. This leaves unspecified what to do if the check fails — throw an exception? Silently covert the input to **V**, maybe logging a warning? But that choice can be left to the discretion of the type implementation; the fixed criterion is that any carrier purporting to hold **ranged_lte<V>** values guarantees that any initialized value has been range-checked.

Note that this checking may be mostly runtime: we might not actually prohibit code like **ranged_lte<100> x = 101** since the value constructor will be called with **101** (and have to decide what to do with it) only at runtime. Compilers can do basic arithmetic — they can warn, say, that in **int x = 0.5** the decimal is truncated to an integer — but compilers for most languages allow only primitive number-related checks; certainly they cannot enforce axioms such as an "insert" operation on a list increasing the list's size by one. That is, enforcing restrictions like range-checks via gatekeeper value constructors does not realize the vision of compile-time type-checking to sniff out anomalies, but it does serve the goal of guaranteeing properties of values prior to functions being called so that these do not have to be checked in the implementation itself.

Type classes like **ranged_lte<V>** also have the feature that they will only be instantiated in a handful of concrete types; they are simpler than real dependent-type constructs such as, consider, a possible function taking two *increasing* integer values: **f(x, y)** where **x** $<$ **y**. How can we express the **x** $<$ **y** condition within **f**'s signature, assuming the signature can only express semantics pertinent to **f**'s type attribution? On the face of it, we know that the desired "increasing" condition is equivalent to **y** having a type like **ranged_gt<x>**, where this "**x**" is the **x** preceding **y** in **f**'s signature. But using such directly as **y**'s type-attribution means that from the perspective of **f**'s own type-attribution, **y** does not have a single, fixed type; its type varies according to the value of **x**. As a consequence, the proper value-constructor for **y** cannot be known at until runtime. Moreover, for reasons I reviewed above in the context of "indexed" types, **y**'s value constructor would have to be some temporary value (you couldn't have directly-addressable constructors indexed over non-enumerative types like **int**). Here again we encounter a "metaconstructor" problem: in order for the **x** $<$ **y** condition to be modeled *directly* by **y**'s type-attribution we would need the constructor for **y**'s value-constructor to be some operation that produces a temporary function-value — not simply the compilation of a code-graph to a non-temporary implementation that can be directly-addressable via a function-pointer.

These issues go away if, instead of working with a function taking *two* integers, we instead consider a function taking *one* value which is a monotone-increasing pair (something like **int f(mi_pair pr)**). A type like **mi_pair**, based on ordered pairs x, y of **int**s, solves the metaconstructor problem for **y** because **x** and **y** are no longer distinct **f** parameters with distinct value-constructors; they are subsumed into one pair, whose own value-constructor can check the **x** $<$ **y** condition. The *requirements* for the original (two-valued) **f** may then be *described* as **x** and **y** being convertable into a pair **pr** which is an instance of **mi_pair** (so that **x** $<$ **y**). This *description* is not a *type*, but elevating the description to type-level can be at least approximated with a signature like **f(int x, int y, mi_pair pr = mi_pair(x, y))**, which when called as **f(x, y)** will silently call the **mi_pair** constructor. This is only approximate because it allows anomalies like **f(x, y, mi_pair(0, 1))**, defeating the purpose — how well this "hidden values" technique (similar to what is sometimes called "passkey" parameters) approximates dependent-typed protocols depends on how well client code can be forbidden from calling the three-argument form directly.

A Channel-based solution is to introduce a *hidden* channel of values which are initialized from values in the other channels (rather than passed in from the call site). A variation on this theme is to construct hidden values that classify values within types more precisely than types themselves. Such finer classification is a feature of "typestate" systems — the distinction between closed and open files, for instance, being a variation in typestate within an overall **file** type. For typical typestates, it is possible in principle to implement types whose constructors only accept values in the associated state — e.g., a constructor for an "open file" type that takes **file** values but only returns a value if the passed file is open. Despite their greater guarantees, these narrower types may have limited usefulness because it is presumably impossible to know at compile-time if (say) a file is (going to be) open. On the

other hand, creating an open-file value is a way to gatekeep specifications that a function body only be entered if the relevant file is open. Placing tests in value-constructors is a way to express them through a vehicle more central to a type system and TXL.

Aside from typestate as exclusionary — preventing function bodies from running without specific criteria being met — typestates can also "overload" functions on finer-grained criteria (relative to the type system itself). Consider a function which works on two lists, but needs different implementation depending on which is larger, or both equal-sized. Functions cannot typically be overloaded based on such runtime criteria alone, but via "hidden" channels we can assign them different signatures: one takes a type whose constructor only succeeds if it is passed two lists of increasing size; another's hidden type only gets initialized if the sizes are *decreasing*; and a third only if the sizes are equal. But notice that these three "hidden" types can also be interpreted as typestates of a pair-of-lists type; every **pair<list<...>>** value can be sorted into one of three states (size-increasing, or decreasing, or equal). That is: apply passkey techniques like those discussed above for a hidden **mi_pair** argument, only to test an "increasing" condition not on values but on list-sizes. Suppose we have three list-pair functions overloaded on variants of **mi_pair**, which collectively span the size-comparison typestates of a "pair-of-lists". A compiler — or perhaps some supplemental code-generation tool — could plausibly default-implement an associated function (visible to client code, without the hidden channel) that delegates to one of these overloads, automating the size-comparison check without extra code either at the call site or in the implementations. More than just a device for inserting runtime checks at useful "pointcuts" (using Aspect-Oriented terms), such compiler enhancements, working through hidden channels, allow automatic dispatching to implementations who may declare but not test assumptions and make type-systems allowing these functions closer to bonafide dependent-type engines.

These examples do not exhaust the topic of type-system-level articulation and enforcement of runtime specifications, but they perhaps pose ideas that with suitable variation can apply in other contexts and unify subjects I have discussed: if a larger class of specifications can be integrated within type systems, then perhaps a larger class of source code features (blocks, statements, etc.) can be interpreted as typed values.

4.3.1 Channelized-Type Interpretations of Larger-Scale Source Code Elements

By intent, Channel Algebras provide a machinery for modeling function-call semantics more complex than "pure" functions which have only one sort of input parameter (as in lambda abstraction) — note that this is unrelated to parameters' *types* — and one sort of (single-value) return. Examples of a more complex paradigm come from Object-Oriented code, where there are two varieties of input parameters ("lambda" and "sigma"). Consider method calls in Object-Oriented languages: these typically have a special syntax with one distinguished carrier. This carrier is in a sense privileged: the type of the "**this**" carrier establishes the class to which function belongs, influencing when the function may be called and how polymorphism is resolved. Moreover, "chaining" method calls means that the result of one method becomes an object that may then receive another method (the following one in the chain). Such chaining allows for an unambiguous function-composition operator: since functions in general take multiple arguments, there is no single operator to pass the result of one function to another; but since most methods have one return value and one **this** object, it is straightforward to define a method-chain operator.

Another case-study is offered by exceptions. The option to "throw" an exception can be considered an alternative kind of output channel. A function throws an exception instead of returning a value. As a result, **return** and **exception** channels typically evince a semantic requirement (which earlier — see the notations on page **??** — I sketched as an algebra stipulation): when functions have both kinds of channels, only one may have an initialized carrier after the function returns. Usually, thrown-exception values can only be bound to carriers in **catch(...)** formations — once held in a carrier they can be used normally, but carriers in **exception** channels themselves can only transfer values to other carriers in narrow circumstances (this in turn depends on things like code blocks, which in turn will be reviewed below). So **exception** channels are not a sugared form of ordinary returns, any more than objects are sugar for functions' first parameter; there are axiomatic criteria defining possible formations of **exception** and **return** channels and carriers, criteria which are more transparently rendered by recognizing **exception** and **return** as distinct channels of communication available within function bodies.

In general, extensions to λ-Calculus are meaningful

because they model semantics other than ordinary lambda abstraction. For example, method-calls (usually) have different syntax than non-method-calls, but ς-calculi aren't trivial extensions or syntactic sugar for **lambda**s; the more significant difference is that sigma-abstracted symbols and types have different consequences for overload resolution and function composition than **lambda**-abstractions. Similarly, exceptions interact with calling code differently than return values. These differences do not belong precisely to λ-Calculus, because they are consequential more in the calling context than in the called implementation — though not entirely, since (for example) throwing exceptions aborts lexical finalization. Nor do they belong precisely to process calculi, because they are most complex in the context of sequential procedures (for example, a function cannot catch exceptions thrown from functions it has spawned in new threads). As intended here, "Channel Algebra" suggests an intermediate formalization combining features of both (generalized) lambda and process calculi (or algebras). Instead of scattered λ-extensions, Channel Algebra unifies multiple expansions by endowing functions (their signatures, in the abstract, and function-calls, in the concrete) with multiple channels, each of which can be independently modeled by some λ-extension (objects, exceptions, captures, and so forth).

Specific examples of unorthodox λs (objects, exceptions, captures) suggest a general case: relations or operators between functions can be modeled as relations between their respective channels, subject to certain semantic restrictions. A *method* can be described as a function with several different channels: say, a "**lambda**" channel with ordinary arguments (as in λ-calculus); a "**sigma**" channel with a distinguished **this** carrier (formally studied via "ς-calculus"); and a **return** channel representing the return value. Because the contrast between these channels is first and foremost *semantic* — they have different meanings in the semantics of the programs where they appear — channels may therefore have restrictions governed by programs' semantics. For example, as I mentioned in the context of "method chaining", it may be stipulated that both **sigma** and **return** channels can have at most one carrier; as a result, a special channel-to-channel operator can be defined which is specific to passing values between the carriers of **return** and **sigma** channels. This operator is available because of the intended semantics of the channel system.

In general, a Channel Algebra identifies several *kinds* of channels which each have their own semantic interpretation, and accompanying axioms or restrictions. On the basis of these semantic details, channel-to-channel operators can be defined, derived from underlying carrier-to-carrier operators. A Channel Algebra in this sense is not a single fixed system, but an outline for modeling function-call semantics in the context of different programming languages and environments.

As the preceding paragraphs have presupposed, different functions may have different kinds of channels, which may or may not be reflected in functions' types (recall the question, can two functions have the same type, if only one may throw an exception)? This may vary between type systems; but in any case the contrast between channel "structures" is *available* as a criteria for modeling type descriptions. On this basis, as I will now argue, we can provide type-system interpretations to source code structures beyond just values and symbols.

4.3.2 Statements, Blocks, and Control Flow

The previous paragraphs discussed expanded channel structures — with, for example, objects and exceptions — that model call semantics more complex than the basic lambda-and-return (of classical λ-Calculus). A variation on this theme, in the opposite direction, is to *simplify* call structures: functions which lack a return channel have to communicate solely through side-effects, whose rigorous analysis demands a "type-and-effect" system. Even further, consider functions with neither **lambda** nor **return** (nor **sigma** nor, maybe, **exception**). As an alternate channel of communication, suppose function bodies are nested in overarching bodies, and can "capture" carriers scoped to the enclosing function. "Capture semantics" specifications in C++ are a useful example, because C++ (unlike most languages that support anonymous or "intra-expression" function-definitions) mandates that symbols are explicitly captured (in a "capture clause"), rather than allowing functions to access surrounding lexically-scoped with no further notation: this helps visualize the idea that captured symbols are a kind of "input channel" analogous to **lambda** and **sigma**.

I contend this works just as well for code blocks. Any language which has blocks can treat them as unnamed function bodies, with a "**capture**" channel (but not **lambda** or **return**). When (by language design) blocks *can* throw exceptions, it is reasonable to give them "**exception**" channels (further work, that I put off for now, is needed for loop-blocks, with **break** and **continue**). Blocks can then be typed as

function-like values, under the convention that function-types can be expressed through descriptions of their channels (or lack thereof).

Consider ordinary source-code expressions to represent a transfer of values between graph structures: let Γ_1 and Γ_2 be code-graphs compiled from source at a call site and at the callee's implementation, respectively. The function call transfers values from carriers marked by Γ_1 nodes to Γ_2 carriers; with the further detail of "Channel Packages" we can additionally say that the recipient Γ_2 carriers are situated in a graph structure which translates to a channel description. So the morphology of Γ_1 has to be consistent with the channel structure of Γ_2. For regular ("value") expressions, we can introduce a new kind of channel (which I'll call "lookup") acknowledging that the function called by an expression may itself be evaluated by its own expression, rather than named as a single symbol (as in a pointer-to-function call like `(*f)(x)` in C). A segment of source code represents a value-expression insofar as an equivalent graph representation comprises a Γ semantically and morphologically consistent with the provision of values to channels required by a function call — including the lookup channel on the basis of which the proper implementation (for overloaded functions) is selected. How the graph-structure maps to the appropriate channels varies by channel kind: for instance the **return** channel is not passed *to* the callee, but rather bound to a carrier as the right-hand-side of an assignment (an *rvalue*) — or else passed to a different function (thus an example of channel-to-channel connection without an intervening carrier). A well-formed Γ represents part of a function implementation's code graph, specifically that describing how a Channel Package is concretely provisioned with values (i.e., a payload).

I will use the term *call-clause* to designate the portion of a code graph, and the associated collection of source code elements, describing a Channel Payload. Term a call-clause *grounded* if its resulting value is held in a carrier (as in `y = f(x)`), and *transient* if this value is instead passed on (immediately) to another function (as in `h(f(x))`); moreover a call-clause can be *standalone* if it has no result value or this value is not used; and *multiply-grounded* if it has several grounded result values — i.e., a multi-carrier **return** channel, assuming the type system allows as much. Grounded and standalone call-clauses can, in turn, model *statements*; specifically, "assignment" and "standalone" statements, respectively.

This vocabulary can be useful for interpreting program flow. Assignment statements with no other side effects can be delayed until their grounding carrier is convoluted with some other carrier. Of course, the default choice of "eager" or "lazy" evaluation is programming-language-specific, but for abstract discussion of source code graphs, we have no *a priori* idea of temporality; of a program executing in time. This is not a matter of concurrency — we have no *a priori* idea of procedures running at the *same* time any more than of them running sequentially (cf. "detached" evaluation as on page 23). Any temporal direction through a graph is an interpretation *of* the graph, and as such it is useful to assume that graphs in and of themselves assert no temporal ordering among their nodes or edges. When modeling eager-evaluation languages, particular edge-types can be designated as forcing a temporal order or else edges can be annotated with additional temporalizing details. Without this extra documentation, however, execution order among graph elements can be evaluated based on other criteria.

In the case of statements, an assignment without side effects has temporalizing relations only with other statements using its grounding carrier. In particular, the order of statements' runtime need not replicate the order in which they are written in source code. For sake of argument, consider this the default case: Channel Algebras, in principle, model "lazy" evaluation languages, in the absence of any temporalizing factors. The actual runtime order among sibling statements — those in the same block — then depends, in the absence of further information, on how their grounding carriers are used; this in turn works backward from a function's return channel (in the absence of exceptions or effectual calls). That is, runtime order works backward from statements that initialize carriers in the return channel, then carriers used in those statements, etc.

This order needs to broken, of course, for statements with side-effects. A case in point is the expansion of "**do**-notation" in Haskell: without an *a priori* temporality, Haskell source code relies on the asymmetric order of values passed into lambda abstractions to enforce requirements that effectual expressions evaluate before other expressions (Haskell does not have "statements" per se). Haskell's **do** "blocks" can be modeled (in the techniques used here) as a series of assignment statements where the grounding carrier of each statement becomes (i.e., transfers its value to) the sole occupant of a **lambda** channel marking a new function body, which includes all the following statements (and so on recursively). There are two concepts in play here: interpreting any sequence of statements (plus one terminating expres-

sion, which becomes a statement initializing a **return** carrier) as a function body (not just those covering the extent of a "block"); and interpreting assignment statements as passing values into "hidden" lambda channels. Of course, Haskell backs this syntactic convention with monad semantics — the value passed is not the actual value of the monad-typed carrier but its "contained" value. For sake of discussion, let's call this a *monad-subblock* formation.

The temporalizing elements in this formation are the "hidden lambdas". In a multi-channel paradigm, we can therefore consider "monad-subblocks" with respect to other channels. Consider how individual statements can be typed: like blocks, statements can select from symbols in scope and can potentially result in thrown expressions, so their channel structure is something like **capture** plus **exception**. Even without hidden lambdas, observe that the runtime order of statements can be fixed in situations where an earlier statement can affect the value (via non-constant capture) of a carrier whose value is then used by a later statement. So for languages with a more liberal treatment of side-effects than Haskell, we can interpret chains of statements *in fixed order* as successively capturing (and maybe modifying) symbols which occur in multiple statements. Having discussed convoluted *carriers*, extend this to channels: in particular, say two **capture** channels are convoluted if there is a modifiable carrier in the first which is convoluted with a carrier in the second (this is an ordered relation). One statement must run before a second if their **capture** channels are convoluted, in that order.

This is approaching toward a "monad-subblock" formation using ***capture*** in place of **lambda**. To be sure, Haskell monad-subblock does have the added gatekeeping dimension that the symbol occurring after its appearance as grounding an assignment statement is no longer the symbol with a monad type, but a different (albeit visually identical) symbol taken from the monad. Between two statements, if the prior is grounded by a monad, the implementation of its **bind** function is silently called, with the subsequent (and all further) statements grouped into a block passed in to $>>=$, which in turn (by design) both extracts its wrapped value and (if appropriate) calls the passed function. But this architecture can certainly be emulated on non-**lambda** channels — a transform that would belong to the larger topic of treating blocks as function-values passed to other functions, to which I now turn.

4.3.3 Code Blocks as Typed Values

Insofar as blocks can be typed as functions (in a sense), they may readily be passed around: so loops, **if…then…else**, and other control flow structures can plausibly be modeled as ordinary function calls. This requires some extra semantic devices: consider the case of **if…then…else** (I'll use this also to designate code sequences with potential "**else if**"s), which has to become an associative array of expressions and functions with "block" type (e.g., with only **capture** and **exception** channels). We need, however, a mechanism to suppress expression evaluation. Recall that expressions are concretized channel-structures which include a **lookup** channel providing the actual implementation to call. Assume that **lookup** can be assigned a flag which suppresses evaluation. Assume also that carriers can be declared which hold (or somehow point to) *expressions* that evaluate to typed values, in lieu of holding these values directly (note that this is by intent orthogonal to a type system: the point is not that carriers can hold values whose type is designed to encapsulate potential computations yielding another type, like **std::future** in C++). Consider again the nested-expression variant of $\mathfrak{C}_1 \looparrowright \mathfrak{C}_2$: when the result of one function call becomes a parameter to another function, the value in the former's **return** carrier (assume there is just one) gets transferred to a carrier in the latter's **lambda** channel (or **sigma**, say). This handoff can be described before being effectuated: a language runtime is free to vary the order of expression-evaluation no less than of statements. The semantics of a carrier-transfer between f_2's return and f_1's lambda does not stipulate that f_2 has to *run* before f_1; language engines can provide semantics for $\mathfrak{C}_1 \looparrowright \mathfrak{C}_2$ allowing $\mathfrak{C}_1$ to hold a delayed capability to evaluate the f_2 expression. Insofar as this is an option, functions can be given a signature — this would be included in the relevant TXL — where some carriers are of this "delayed" kind. Functions like **if…then…else** can then be declared in terms of these carriers.

To properly capture **if…then…else** semantics, it is also appropriate to notate conditions on how blocks passed as function values are used. In the case of **if…then…else**, the implementation will receive multiple function values with the caller expecting *exactly one* function to be called. This constitutes a requirement on how carriers are used, analogous to mandating that a **return** (or either **return** or **exception**) carrier be initialized, or that mutable references passed to a function for initialization (as an alternative to returning multiple values) are initialized. While such requirements can potentially be described as annotations on carriers/channels,

this is a general issue outside the scope of blocks-as-function-values.

A thorough treatment of blocks-as-functions also needs to consider standard procedural affordances like **break** and **continue** statements. Since blocks can be nested, some languages allow inner blocks to express the codewriter's intention to "break out of" an outer block from an inner block. One way to model this via Channel Algebra is to introduce a special kind of return channel for blocks (called a "**break**", perhaps) which, when it has an initialized carrier, uses this channel to hold a value that the enclosing block interprets in turn: by examining the inner **break** the immediately outer block can decide whether it itself needs to "break" and, if so, whether its own **break** channel needs to have an initialized carrier. The presence of such a **break** can type-theoretically distinguish loop blocks from blocks in (say) **if…then…else** contexts.

Further discussion of code models via Channel Complexes and Channel Algebras is outside the scope of this chapter, but is demonstrated in greater detail in the accompanying code-set. Hopefully, the best way to present Channel Semantics outside the basic **lambda/sigma/return/exception** quartet is via demonstrations in live code. In that spirit, the demo code-set focuses more on practical engineering and problem-solving where Channel models can be useful, and I'll briefly review its structure and its organizing rationales in the Conclusion.

5 Conclusion

There are several tactics to practically employ techniques I have reviewed here in concrete projects. For sake of discussion, I will focus on the approach used by the demo code-set provided with this chapter. This code actually has multiple dimensions, because it has been designed to integrate with data sets accompanying other chapters in the present volume, not merely this chapter. This has led to the code being planned as follows: code published with *this* chapter directly uses its Hypergraph and Channel Complex libraries to parse Interface Definition files associated with *other* chapters. Those chapters' data sets in turn employ C++ code whose construction is influenced by Channel Algebra — notably in the design of function groups which are overloaded via types that are reasonable for "hidden channels", and in how functions are exposed to external scripting and reflection. The implementations are in normal C++, but the associated IDL annotations describe them via more sophisticated channel formations, which are mimicked (in terms of conventional input parameters) in the production code.

The "hidden" values constructed while routing between logically related functions, as seen in the various code-sets, tend to fit into two classifications. On the one hand, these may be values constructed as a test that function arguments conform to protocol: for instance, one way to check that a numeric value falls within some desired range is to cast this value to a range-checked type — whether or not the second value is actually used. Alternatively, in some places supplemental values are enumerations of typestates — allowing function bodies to achieve something resembling pattern-matching as a code-branching structure (by **switch**ing on enum values).

Manually creating residual values along these lines is admittedly cruder than pure type-level engineering, as exemplified by Idris's synthesis of Dependent Types, typestate, and effect-typing. The solutions chosen for the present context are less automated; they require deliberate choice by developers. Nevertheless, the Interface Definition technology makes these choices explicit and documented, which can help guide developers toward embracing these more fine-grained coding conventions and confirming that they have done so consistently.

It is a legitimate question whether establishing coding paradigms — in a data set management context, for example — in this convention-driven spirit is better or more maintainable than approaching data sets in a language like Haskell or Idris whose rigors are more formal than conventional. But the C++-based approach has the benefit of more immediate integration with GUI code, reflected in the Qt components published as wrappers and visualizers for data-set-specific datatypes.

In the best-case scenario, GUI code is a natural extension of programming languages' and individual languages' type systems — with rigorous mapping between value types and visual-object types that graphically display associated values. So long as Functional Programming languages remain operationally separated from GUI drivers — needing a "smoke binding" or foreign-function interface to marshal data between a C-oriented User Interface and the non-visual data (and database) management components of a project — the strongly-typed rigor of Functional Programming environments will be incomplete.

Whether or not by technical necessity or just entrenched culture, GUI frameworks are still predominantly built in procedural or OO languages like C, Java, and C++.

It seems likely that a truly integrated type system, covering User Interface as well as data management logic, will need a hybrid functional/procedural paradigm at some stratum — either the underlying GUI framework or at application-level code. So long as GUI frameworks remain committedly procedural, the most likely site for such hybrid paradigms to emerge is at the application level, and in the context of application-development SLE tools.

Implementing GUI layers in a Functional environment is usually approached from the perspective of *functional reactive* programming, which emphasizes how the interface between visual components and controller logic can be structured in terms of *event-driven* programming. In this paradigm, there is no *immediate* linkage between GUI events and the functions called in response — for example, no single function that automatically gets called when the user clicks a mouse button. Instead, events are entered into a pool wherein each event may have a varying number of handlers (including being ignored entirely). This style of programming accords well with paradigms that try to minimize the number of functions with side effects. Event-handlers are free to post new signals (these are interpreted as events) which may in turn be handled by other functions — so that signals may be routed between multiple functions entirely without side-effects. That said, most events *should* cause side effects eventually — for instance, after all, a user does not typically initiate an action (triggering an event) without intending to change something in the application data or display. But events can be routed between pure functions until an eventful handler is called, so side-effects can be localized in a proportionately small group of functions.

Moreover, certain qualities of GUI design can be expressed as logical constraints rather than as application-level state enforced by procedural code. For example, optimal design may stipulate that some graphical component must automatically be resized and repositioned to remain centered in its parent window, sustaining that geometry even when the window itself is resized. Functional-Reactive frameworks allow many of these constraints to be declared as logical axioms on the overall visual layout and properties of an application, constructing the procedures to maintain this state behind-the-scenes — which minimizes the extent of procedural code needing to be explicitly maintained by application developers.

But while Functional Reactive Programming is a strategy for providing GUI layers on a Functional code base, it can equally be treated as an Event-Driven enhancement to Object-Oriented programming. In an OO context, events and signals constitute an alternative form of non-deterministic method-call, where signal-emitting objects send messages to receiver functions — except indirectly, passing through event-pools. Indeed, as documented by the demo code, events and signals have a natural expression in terms of Channel Algebra, where both signal emitters and receivers are represented via special "Sigma" channels. On this evidence, Functional Reactive Programming should be assessed not just as a GUI strategy for Functional languages but as a hybrid methodology where Functional and Object-Oriented methodologies can be fused, and integrated.

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