Hypergraph Type Theory and Requirements Engineering for Software Development in a CyberPhysical Context

Nathaniel Christen

August 31, 2019

Abstract

This chapter will explore the integration of several methodologies related to source code analysis and software Requirements Engineering. The chapter will review Semantic Web and general graph-based representations of source code, alongside applied type theory (for expressing programming languages' type systems) and systematic accounts of foundational programming elements such as functions/procedures, function calls, and inter-procedure information flows. The principal relatively new representational device described here involves a theory of "channels" which ties together models associated with lambda calculi, type theory, and graph-based code representation. The proposed techniques support documentation and verification of procedural, data type, and holistic specifications — implementational assumptions on procedures and/or modeling assumptions on types. For concrete examples, an accompanying open-source data set (at https://github.com/scignscape/ntxh) demonstrates code libraries concretizing techniques outlined here.

Some seek to encourage reductions in consumption of energy and material goods, or to support changes in purchasing behavior. Others seek to use software capabilities to build smarter (lower impact) infrastructure. However, there is a lack of common understanding of the fundamental concepts of sustainability and how they apply, and a need for a common ground and consistent terminology. As such, persistent misperceptions occur, as researchers and practitioners disagree over whether we're even asking the right questions ... We lack a coherent framework with sound theoretical basis that can provide a well-understood trans-disciplinary basis for sustainability design. — *The Karlskrona Manifesto* [14, p. 5]

It is possible to look at CyberPhysical systems at the level of individual devices. Each device has its own mechanical properties, generates its own kind of data, and requires specially designed software to interpret and understand that data. Often users view the data generated by CyberPhysical devices on phone "apps" or specialized touch-screens. As devices proliferate, so does the diversity of software which users use to access device data.

CyberPhysical systems can also be seen in a more holistic way. As CyberPhysical networks proliferate, we can envision a rise in technologies that merge and integrate data from many kinds of devices, from many different vendors. This eventuality has already been contemplated, including in this volume. Teixera *et. al.*, for example, argue that

> Overall, the increase in sensors, devices, and appliances, in our homes, has transformed it into a rather complex environment with which to interact. This characteristic cannot be merely addressed by a matching set of device-dependent applications, turning the smart home into a set of isolated interactive artifacts. Hence, there is a strong need to unify this experience, blending this diversity into a unique interactive ecosystem. This can be tackled, to a large extent by the proposal of a unique, integrated, ubiquitous distributed smart home application capable of handling a dynamic set of sensors and devices and providing the different house occupants (e.g., children, teenagers, young adults, and elderly) with natural and simple ways of controlling and accessing information. However, the creation of such application presents a challenge, particularly due to the need to support natural and adaptive forms of interaction beyond a simple home dashboard application.

In this chapter, I will refer to an "application capable of handling a dynamic set of sensors and devices" as a *hub application*.

Hub applications must receive data from many kinds of devices. They must also respond properly to data once it is received. For each kind of device, there must then be a corresponding software component, part of the hub application, which is specifically capable of understanding data generated by *that particular* device. For sake of discussion, I will call such device-specific software components a *hub library*. We may assume that hub applications will feature many hub libraries, and that implementing hub libraries will become an integral step in the process of deploying CyberPhysical instruments. Hub libraries need to bridge the low-level realm of CyberPhysical signals (and the networks that carry them) with the high-level realm of software engineering: GUI components, data validation, fluid and responsive User Experience, and so forth.

Hub applications, in short, serve as central loci organizing collections of hub libraries. In this role they have a significance beyond just supplying a User Interface to CyberPhysical data. Hub libraries would need to manifest *data models* conveying technical details about devices; for instance, scientific and mathematical details about the proper range and dimensions of data fields. Hub libraries would provide a concrete artifact that engineers could consult to obtain information about device properties, data models, and expected behavior. Compared to "a set of isolated interactive artifacts", I believe the centralized architecture of hub applications and hub libraries is more conducive to rigorous, concrete representation of CyberPhysical devices as technical products. This centralized focus can make CyberPhysical systems more secure and trustworthy — hub applications can be used for testing and protyping devices and their supporting code, even before devices are brought to market.

This chapter is not explicitly about hub applications; here I will examine coding and code documentation techniques which are applicable in many contexts. However, CyberPhysical hubs are a good case study in programming contexts where Requirements Engineering is an intrinsic architectural feature. I write this chapter, then, from the perspective of a programmer creating a "hub library" for some form of CyberPhysical input. This programmer needs to express in code the physical and computational details specific to the device's signals; its functionality and capabilities. The hub library should serve as a reference point, a proxy for the device itself, in that engineers may study the library as an indirect way of coming to understand the device. Given these requirements, hub libraries need an especially rigorous development methodology, one that emphasizes strict documentation and verification of coding requirements.

The first two sections in this chapter will discuss hub applications and CyberPhysical systems on a more practical level: what are representative examples of the data structures and coding requirements that hub libraries will need to encapsulate? I will then, in the final two sections, turn to computer code at a more theoretical level, outlining certain representational paradigms, such as Directed Hypergraphs, which I believe can yield more expressive and comprehensive models of coding structures and requirements.

1 Hub Applications and Gatekeeper Code

To begin, I will speak in general terms about hub applications and about the unique coding challenges which derive from CyberPhysical technologies' unique networking and safety requirements. 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.

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" (characterized by more complex Graphical User Interface components, and by the ability to work with "local" data — data stored on users' computers or

accessible on a local network — instead of or in addition to data acquired from a web service).

From a software engineering point of view, I believe we should conceptualize hub sofware as native, desktop-style applications which leverage native GUI features. Hub applications therefore embody a fundamentally different User Experience than other kinds of CyberPhysical access points, like touch screens or phone apps.

To cite a concrete example, Teixeira *et. al.* consider "smart appliance" refrigerators which are aware of the door being left open, and even may track the expiration date of items inside. Consider then how we would design a User Interface networking with a "smart" regrigerator. A simple indicator showing whether doors are open is straightforward, but an interface listing food items in a refrigerator is much more complicated. Assuming (for the sake of this rather futuristic discussion) that a refrigerator can detect signals emanating from food items (or, more precisely, their containers), we can envision a list of items (maybe with their expiration dates) presented as a GUI component, maybe with one food item per line (lines perhaps showing a picture, text description, dates, etc.). This would require cross-referencing numeric codes, which might be broadcast by the items *inside* the refrigerator, against a database that would load images, descriptions, and any other GUI content relevant to that item. One question is then where this database would be hosted, and how it would be updated (insofar as food companies develop new products fairly often; any static database could quickly get outdated). Food companies (or some middleware agent) would have to agree on a common format so that the refrigerator's access software can integrate data from many brands. Since a refrigerator can hold many items, the GUI would also need enough screen space (and maybe a multi-level design) for users to comfortably browse many artifacts; perhaps a line-by-line window supplemented with separate dialog windows for each item. It would be difficult to provide this advanced a GUI via a phone app or an in-kitchen touch-screen.

On the other hand, a phone app interfacing with that same data would be constrained by its limited screen space and interactive modalities. For example, with no room on-screen to show a complete list of food items — and no obvious user gesture to navigate between individual and multi-item views — such an app might choose to list only those items nearing their expiry date. In general, when adapting to phone-like usage patterns (limited screen, brief but frequent user engagement), designers have to offer compact but curated snippets of information, in lieu of comprehensive access into a data space. That is, software can try to anticipate which pieces of data carry the most user interest — expiry dates are most likely important to users when items are near perishing. This means that data mining, Artificial Intelligence, and other techniques for anticipating users' needs becomes proportionately more consequential: if the whole GUI design is premised on AI, then AI ceases to be just a useful tool, augmenting software's analytic reach — it becomes instead a make-or-break User Experience necessity. If an app cannot practically share *all* data, it has to guess what data users most want to see.

Conversely, if we assume that hub applications would prototypically adopt the "look and feel" of native desktop applications, then they can present a responsive, comprehensive overview of data structures to their users. Applications can take advantage of more screen space, with secondary application windows and other interactive features that we associate with native GUI components. This arguably can render AI less important, because it is easier for users to interact with the software on their own terms. As Teixera *et. al.* put it, "some authors argue that the number of interactions between users and the smart home must be kept to a minimum", but "to remove obstacles in the adoption of smart home systems ... preserving the autonomy of the user may seem like the most sensible course of action". The more data that can be shown, the less need for software to filter information on users' behalf.

Hub applications are therefore examples of what Teixera *et. al.* call "user-centric" design. In this guise, 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.

Once software receives device data, it needs to marshal the 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 confirm 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?

Because CyberPhysical devices are intrinsically *networked* — whether over special wireless networks or the World Wide Web — there is an enlarged "surface area" for vulnerability. Moreover, because they are often worn by people or used in a domestic setting, they tend carry personal (e.g., location) information, making network security protocols especially important ([12], [42], [71], [112], [113], [120]). In brief, 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.

It is helpful at this point to distinguish cyber *security* from *safety*. When these concepts are separated, *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 300 fatalities (plus the worldwide grounding 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 [83] and the Fukushima nuclear reactor meltdown [130]. 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 commensurately to agree on a protocol for encoding the numbers 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?

In general, 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? Continuing the blood-pressure example, how would a software procedure that *does* check the integrity of the systolic/diastolic pair indicate for the overall system model that it performs that particular verification? Conversely, how would a procedure which does *not* perform that verification indicate that this verification must be performed elsewhere in the system, to guarantee that the procedure's assumptions are satisfied?

These questions are important not only for objective, measurable assessments of software quality, but also for people's more subjective trust in the reliability of software systems. In the modern world we allow software to be a

determining factor in systems' behavior, 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:[1]

> 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.

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.

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

The problem, as software becomes an increasingly central feature of the biomedical ecosystem, is that no commensurate oversight framework exists in the software 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 software errors threaten personal and/or property damages.

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. 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. With respect to medications, policy makers can stipulate that pharmaceuticals should 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 should

be a software-design issue before it becomes a policy or contractual issue. Software engineering and programming language design needs its own evaluative guidelines; its own analogs to double-blind trials and five-sigma confidence. It is at the region 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.

1.1 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 t. This type can then be associated with a type (say, t') of GUI components which visually display values of type t. A simple data structure may have GUI representation via small "widgets" embedded in other components (consider a thermometer icon to display temperature). Conversely, if $\mathcal{S}$ has many component parts, its corresponding GUI type may need to span its own application window, with a collection of nested textual or graphical elements. There may also be a type (say, t'') representing 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 localize 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. [70], [95], [96]) and the theory of Hypergraph Categories ([33], [52], [53], [75]).

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.

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 behave 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 commence. In short, *fragile* and *gatekeeper* code often has to be paired off: for each segment of fragile code which *makes* assumptions, there should be a corresponding 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 systolic and diastolic blood pressure, returning an unsigned integer. If this code were called with malformed data wherein systolic and diastolic are 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 "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 configured such 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 maintenance. 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 complexity 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 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.

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 procedure), 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 obser-

vations (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. This permits us to formulate 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 procedure's own implementation. A related idea is overloading fragile procedures: 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 artifacts. 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.

1.2 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-conscious 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. [45, 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, expressive 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 [93] and [87], [121]:

> 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. [93, 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 (potentially) 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 Lambda 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 Lambda 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 §3 and §4.

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 Groups" 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 Groups 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 Groups 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.

2 Case Studies

To motivate the themes I will emphasize going forward, this section will examine some concrete data models which are used or proposed in various CyberPhysical contexts. I hope this discussion will lay out parameters on device behavior or shared data to illustrate typical modeling patterns and their corresponding safety or validation requirements. As an initial overview, the following are some examples of data profiles that might be wedded to deployed CyberPhysical devices (my comments here are also summarized in Table 1 on page 19):

Heart-Rate Monitor 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. Interpreting heart-rate data depends on whether the person is resting or exercising. Therefore, a usable data structure might join a beats-per-minute dimension with a field indicating (or measuring) exertion, either a two-valued discrimination between "rest" and "exercise" or a more granular sampling of a person's movement cotemporous with the heart-rate calculations.

Accelerometers These devices measure object's or people's rate of movement (see [8], [18], [79], [77], etc.), and therefore can be paired with heart-rate sensors to quantify how heart rate is affected by exercise (likewise for other biometric instruments, such as those calculating respiration rate).

Outside the biomedical context, accelerometers are important for Smart Cities (or factories, and so forth) for modeling the integrity of buildings, bridges, and industrial areas or structures (see e.g. [124], [133]).

An accelerometer presents data as voltage changes in two or three directional axes, data which may only produce signals when a change occurs (and therefore is not continuously varying), and which is mathematically converted to yield information about physical objects' (including a person's) movement and incline. Mechanically, that is, accelerometers actually measure *voltage*, from which quantitative reports of movement and incline can be derived. Accelerometers are classified as *biaxial* or *triaxial* depending on whether they sample forces in two or three spatial dimensions.

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-dimensional relationships.

Remote Medical Diagnosis An emerging application of CyberPhysical technology involves medical equipment deployed outside conventional clinical settings — in remote areas with little electricity, refugee camps, temporary ad-hoc medical units (established to contain potential epidemics, for instance), and so forth. These settings have limited diagnostic capabilities, so data is often transmitted to distant locations in lieu of on-site laboratories.

A good case-study derives from "medical whole slide imaging" (MWSI) [13], where a mobile phone attached to an ordinary microscope, by subtle modifications of camera position and microscope resolution, allows many views to be made on one slide. Positional data (the configuration of the phone and microscope) then merges with image segmentation computations characteristic of conventional whole slide imaging (see, e.g., [49]), and diagnostic pathology in general, which is concerned with isolating medically significant image features and identifying diagnostically significant anomalies (such as cell shapes suggesting cancer).

Segmentation, in turn, generates multiple forms of geometric data: in [73], for instance, segments are identified as approximations to ellipse shapes, and features are tracked across scales of resolution, so geometric data merges ellipse dimensions with positional data (in the image) and a metric of feature persistence across scales. (Features which are detectable at many scales of resolution are more likely to be empirically significant rather than visual "noise"; calculating cross-scale "persistence" is an applied methodology within Statistical Topology — see e.g. [43], [84], [110]). Merged with MWSI configuration info and patient data, the whole data package integrates geometric, CyberPhysical, and health-record aspects.

Speech Sampling Audio sensors can be used to isolate different people's speech episodes (see Raju Alluri and Anil Kumar Vuppala, this volume; and Ravi Kumar Vuddagiri *et. al.*, this volume). Feature extraction cancels background noise and partitions the foreground audio into different segments, individuated (potentially) by differences between speakers as well as each speaker's conversation turns. Such data can then be employed in several ways. António Teixera's chapter (*et. al.*, this volume) discusses speech-activated User Interfaces for software, while the previous two chapters mentioned above present methodology for estimating speakers' emotional states and identifying samples' spoken language or dialect, respectively.

The data profile germane to an audio processor will be determined by the system's overarching goals. For example, [32] describes techniques for measuring emotional stress via heart-rate signals. Combined with speech-derived data, a system might accordingly be designed around emotional profiles, merging linguistic and biometric evidence. For those use-cases, programming would emphasize signs of emotional changes (reinforced by both metrics), and secondarily isolating times and locations, which factor into proper sofware responses to users' moods.

On the other hand, a voice-based User Interface might similarly model speakers' identity and location, but perform Natural Language Processing to translate speech patterns into models of user requests. Conversely, the use-case in Vuddagiri *et. al.* in this volume, where speech data is parsed for language classification (viz., matching voices to the language or dialect spoken) as part of a "smart city" network, calls for different features. The priority here is not necessarily identifying individual speakers, but potentially tagging samples to obtain a geospatial model of language-use in a given area.

Bioacoustic Sampling Similar to speech sampling (at least up to the point where acoustical analysis gives way to syntax and semantics), audio samples can be used to track and identify species (Todor Ganchev, this volume; and Boulmaiz, *et. al.*, this volume). Here again feature extraction foregrounds certain noise patterns, but the main analytic objective is to map audio samples to the species of the animal that produced them. Sensor networks can then build a geospatial/temporal model of species' distribution in the area covered by the network: which species are identified, their prevalence, their concentration in different smaller areas, and so forth.

These measurements can be employed in the study of species populations and behavioral patterns, and can also add data to urban-planning or ecological models. For example, precipitous decline of a species in some location can signal environmental degradation in that vicinity.

Data sets such as those accompanying [103] (the smallest, labeled CLO-43SD, is profiled within this chapter's data set) provide a good overview of data generated during species identification: in addition to audio samples themselves (in WAV format), the data set includes NPY (Numerical Python) files representing different spectral analysis methods applied to the bird songs, as well as a metadata file summarizing species-level data (such as the count of samples identified for each species). Every species also acquires a 4-letter identifier then used as part of the WAV and NPY file names, so the file names themselves serve a classifying role, semantically linking the sample to its species. These three levels of information are a good example of the contrast in granularity — and the mechanisms of information acquisition — between raw CyberPhysical input (the audio files), midstream processing (the spectral representations), and summarial overviews (species counts and labels; other avian data sets might also recognize geospatial coordinates obtained via noting sensor placement, as a further metadata dimension).

Facial Recognition Given a frontal (or, potentially, partial) view, software can rather reliably match faces to a preexisting database or track faces across different locales ([27], [41], [68], [72], [82], etc.). The most common methodology depends on normalizing each foreground image segment (corresponding to one face) into a rectangle, whose axes then establish vector components for any features inside the segment. Feature extraction then isolates anatomical features like eyes, nose, mouth, and chin, quantifying their position and distances, yielding a collection of numeric values which can statistically identify a person with relatively small error rates.

Given privacy concerns, enterprise or government use of this data is controversial: should analyses be performed on every person, or only on exceptional circumstances (crime investigation, say)? Can facial-recognition outcomes be anonymized so that faces would be tracked across locations but not tied to specific persons without extra (normally inaccessible) data? When and by whom should face data be obtainable, and under what legal or commercial circumstances? Should stores be allowed to use these methods to prevent shoplifting, for example? What about searching for a missing or kidnapped child, or keeping tabs on an elderly patient? When does surveillance cross the line from benevolent (protecting personal or public safety) to privacy-invasive and authoritarian?

Of course, there are many other examples of Cyber-Physical devices and capabilities that could be enumerated. But these cases illustrate certain noteworthy themes. One observation is that a gap often exists between how devices physically operate and how they are conceptualized: accelerometers, for instance, mechanically measure voltage, not acceleration or incline; but their data exposed to client software is constructed to be used as vectors indicating persons' or objects' movement. Moreover, multiple processing steps may be needed between raw physical inputs and usable software-facing data structures. Such processing may generate a large amount of intermediate data; for instance, feature extraction from audio or image samples can yield numeric aggregates with tens or hundreds of different fields. Further processing usually reduces these structures to narrower summaries: an audio sample might be consolidated to a spatial location and temporal timestamp, along with a mapping to an individual person speaking (perhaps along with a text transcription), or human language spoken, or animal species. Engineers then have to decide what level of detail to expose across a software network. Another issue is integrating data from multiple sources: most of the more futuristic scenarios envision multi-modal Ubiquitous Computing spaces where, e.g., speech and biometric inputs are cross-referenced.

Different levels of data resolution also intersect with privacy concerns: simpler data structures are more likely to employ private or sensitive information as an organizing instrument, heightening security and surveillance concerns. For example, a simple facial-recognition system would match faces against known residents of or visitors to the relevant municipalities. This is less technologically challenging than anonymized systems which would persist more mid-processing data in order to complete the algorithmic cycle — matching faces to concrete individuals — only under exceptional circumstances; of course, though, it is also a greater invasion of privacy.

Analogously, syncing speech technology with personal health data would be simplified by directly matching speaker identifications to biosensor devices wearers. Again, though, using personal identities as an anchor for disparate data points makes the overall system more vulnerable to intrusive or inappropriate use. In total, security concerns might call for more complex data structures wherein shared data excludes the more condensed summaries wherever they may expose private details, and rely more on multipart, mid-level processing structures. Rather than organize face-recognition

around a database of persons, for example, the basic units might be numeric profiles paired with probabilistic links notating that a face detected at one time and place matches a face analyzed elsewhere, but without that similarity being anchored in a personal identifier.

Other broad issues raised by these CybePhysical case-studies include (1) testing and quality assurance and (2) data interoperability. In the case of testing, many of the scenarios outlined above (and throughout this volume) require complex computational transformations to convert raw physical data into usable software artifacts. In [6], for example, the authors present technology to measure heart rate from a distance, based on subtle analysis of physical motions associated with blood circulation and breathing. The analytic protocols leverage feature extraction from wireless signal patterns. As with feature extraction in audio and image-analysis (e.g. face recognition) settings, algorithms need to be rigorously tested to guard against false inferences or erroneous generated data. This implies that analytic code needs to be developed in a software ecosystem which is rigorously structured in documenting algorithmic inputs, outputs, and parameters. In [6] the ultimate goal is to introduce heart and breathing monitors within a Smart Home environment, with computations performed on embedded Operating Systems. However, testing and prototyping of the technology should be conducted in a desktop Operating System environment so as to generate or leverage test data, document algorithmic revisions, and in general prove the system's trustworthiness in a controlled setting (including a software environment which transparently windows onto computational processes) before this kind of network is physically deployed.

With respect to data integration, notice how projects mentioned here often anticipate pooled or overlapping information. For instance, Smart Homes are envisioned to embed sensors analyzing speech, biomedical data-points like heart rate (like I cited last paragraph), atmospheric measurements (temperature, say) and appliance or architectural states (windows, doors, or refrigerator doors being open, ovens or stove burners being turned on, heaters/coolers being active, and so forth). In some cases this data would be cross-referenced, so that e.g. a voice command would close a window or turn off a stove. Analogously, [107] (one of whose co-authors, Nouredine Doghmane, is also a coauthor of this volume's chapter on bird species) describes a combination of face-recognition and speech analysis for "multi-modal biometric authentication"; here again a component supplying image-processing data and one supplying speech metrics will need to transmit data to a hub where the two inputs can be pooled. Or, as I pointed out in the case of Mobile Whole Slide Imaging, image-segmentation, CyberPhysical, and personal-health information fields may all be integrated into a holistic diagnostic platform.

Overall, future CyberPhysical systems may be integrated not only with respect to their empirical domain but in term of the environs where they are deployed — Smart Homes, Smart Cities (or factories or industrial plants), hospitals and medical offices, schools and children's activities centers, refugee or displaced-persons camps/campuses, and so forth. I'll take Smart Homes as a case in point. We can imagine future homes/apartments provisioned with a panoply of devices evincing a broad spectrum of scientific backgrounds, from biology and medicine to ecology and industrial manufacturing.

2.1 How Internet of Things Interoperability Affects Data Modeling Priorities

So, let's imagine the following scenario: homeowners have a choice of applications that they may install on their in-home computers, supplied by multiple vendors or institutions, which access the myriad of Smart Home devices they've installed around the property. CyberPhysical products are engineered to interoperate with such hub applications — and therefore with third-party components — as well as with their own "in house"-implemented offerings, such as phone apps.

In this eventuality, individual devices are no longer situated in proprietary circuits linking device signals to apps and databases, for example, where customers purchase each app, and its associated input instrument(s), in isolation. Devices are not only being connected to their own product suite. Instead, technology inside the home is charged with pooling data from many kinds of devices into a comprehensive Smart Home platform, where users can see a broad overview, access disparate device data from a central location, and where cross-device data will be merged into aggregate models: e.g., cross-referencing speech and biometric inputs. In this scenario, devices must be designed to broadcast data to third-party software platforms. Smart Home "hub" applications are likely to be often upgraded; likewise, home owners/renters would likely buy or replace devices fairly often, so the precise configuration of data senders and receivers will dynamically evolve. These givens call for a modular, flexible architecture where a central software hub is poised

to receive data from different devices as they come "online", i.e., exposed on the Smart Home internal network. Hub applications should seamlessly adjust to devices joining and also exiting the network.

All of this calls for carefully designed protocols, where devices not only expose data but do so in a manner conducive to centralized aggregation. The role of a software hub should be not only to receive data, but to transform multi-domain signals into a common graphical presentation. It should also wrangle received data into a common format permitting integration algorithms — e.g. syncing speech and biometrics — to operate properly. Received device data must therefore be systematically mapped to appropriate transform procedures and GUI components. This is important because we are no longer considering data models from the viewpoint of devices' own capabilities (viz., their specific physical measurements and parameters). More technically, the key libraries associated with devices are no longer merely low-level drivers or IoT signal processors. Instead, the technology stack would include a software-prioritizing intermediate semantic layer acting between "smart objects" and corresponding hub applications. The data models at this semantic mid-layer, in short, would no longer be device-centric. Instead, data models would now be assessed in a software-centric milieu: how do we route device data to proper interpretive procedures? How do we consolidate device data into GUI presentations?

Current literature on CyberPhysical data sharing has focused primarily on establishing common formats and Ontologies for CyberPhysical information: our energies have been invested in standardized representational paradigms. But common representational formats is only a minimal foundation for robust software ecosystems. I would argue that engineers have overemphasized the virtues of standardized representations in general, driven perhaps by the press surrounding mechanisms like XML or RDF. While there is nothing wrong with widely-adopted formats, how data is *encoded* is, in essence, tangential to the primary goal of networked software — which is to route shared data to the proper procedures and HCI protocols. In the case of a Smart Home, once we commit to aggregating devices via a software hub, the key organizing principle is a mesh of procedures implemented in the hub application that can pool all relevant devices into one information space. What needs to be standardized then are not so much data *formats*, but in fact data-handling *procedures*.

To put it differently, device manufacturers would now be dealing with an ecosystem in which hub applications receive and aggregate their data, affording users access points to and overviews of device data and state. Hub applications may be provided by different companies and iterations, their inner workings opaque to devices themselves. What can be standardized, however, are the *procedures* implemented within hub software to receive and properly act upon device data. Software might guarantee, for example, that so long as devices are supplying signals in their documented formats, the software has capabilities to receive the signals, unpack the data, and internally represent the data in a manner suitable for device particulars. Devices can then specify what kind of internal representations are appropriate for their specific data, essentially specifying conditions on software procedures and data types.

In short, the key units of mutual trust and verification among and between CyberPhysical devices and CyberPhysical software are not, in theory, data structures themselves, but instead *procedures* for processing relevant data structures. Robust CyberPhysical ecosystems can be developed by reinforcing procedural alignment wherever possible, including by curating substantial collections of reusable software libraries, either for direct application or as prototypes and testing tools. Suppose many CyberPhysical sensors were paired with open-source code libraries which illustrate how to process the data each device broadcasts. Commercial products could use those libraries directly, or, if they want to substitute closed-source alternatives, might be required to document that their data management emulates the open-source prototypes. Test suites and testing technology can then be implemented against the open-source libraries and reused for stress-testing analogous proprietary components. This appears to be the most likely path to ensuring interoperable, high-quality CyberPhysical technology that serves the ultimate goal of integrated Smart Home (and Smart City, etc.) solutions.

That hypothesis notwithstanding, there are a lot more academic papers on CyberPhysical Ontologies or common signal/message formats, like CoAP and MQTT ([7], [38], [57], [62], [64], [102], etc.) than there are open-source libraries which prototype device data, its validation, parameters, and proper transformations.[2] A good case-in-point can be found

[2]The rationale for emphasizing standard data formats is probably that these formats constrain any procedure which operates on the data, so standardization in data representation indirectly leads to standardization in data-management procedures, or what I am calling "procedural alignment". This accommodates the fact that shared data may be used in many different software environments — components implemented in different programming languages and coding styles. However, more detailed guarantees can be engineered by grounding standardization on procedures rather than data representations themselves. To accommodate multiple programming languages and paradigms, data models can be supplemented with a "reference implementation" which prototypes proper behavior vis-à-vis con-

in [115, pages 4 ff.] using C structures to model CoAP metadata: while it is reasonable, even expected, for low-level driver code to be implemented in C, this code should also be the basis of data models implemented in a language like C++ where dimensions and ranges can be made explicit in the data types. Nevertheless, many engineers will instead focus on describing the more nuanced data modeling dimensions through Ontologies and other semantic specifications wholly separate from programming type systems. This has the effect of scattering the data models into different artifacts: low-level implementations with relatively little semantic expressiveness, and expressive Ontologies which, due to a lack of type-level correspondence between modeling and programming paradigms, are siloed from implementations themselves.

Conversely, in lieu of "data-centric" Ontologies whose mission is to standardize how information is mapped to a *representation*, in this chapter I will consider "Procedural" Ontologies: ones which focus on procedural capabilities and requirements that indicate whether software components are properly managing (e.g., CyberPhysical) data. The idea is that proving procedural conformance should be a central step, and an organizing groundwork, for showing that software intended for production deployment is trustworthy and complies to technical and legal specifications.

In practical terms, the above discussion mentioned the CLO-43SD (avian) data set and Vuddagiri *et. al.*'s chapter, which (as noted below) builds off the AP17-OLR "challenge" corpus. I will also be referring to recent CoNLL corpora. So, together, these constitute three representative examples of data sets tangibly applicable to CyberPhysical and/or NLP Research and Development: one audio-based (for species identification); one audio/speech; and one linguistic. Based on the idea that R&D data sets should germinate deployment data models, we should look to data sets like these to provide semantic and type-theoretic encapsulations of their information spaces and analytic methods (e.g., spectral waveform analysis, or Dependency Grammar parsing). Accompanying materials for this chapter provide profiles of the three aforementioned data sets, which consolidate their various files and formats into a streamlined — and procedure-oriented, "software-centic" — representational paradigm. The demo code presents examples of procedures and data types targeting these data sets, as well as an architecture for deploying data sets in a procedure-oriented and software-centric manner, in terms of how files and the information they contain are organized, and in terms of employment of data-publishing standards such as the "Research Object" model ([15], [16], [23], [50], [126]).

At present, to make these issues more concrete with further case-studies, in this introductory discussion I will examine in more detail the specific case of speech and language data structures.

2.2 Linguistic Case-Study

Establishing data models for deployed technology is certainly part of the Research and Development cycle, which means that data profiles tend to emerge within the scientific process of formulating and refining technical and algorithmic designs. This is particularly true for complex, computationally nuanced challenges such as image segmentation or (to cite one above example) measuring heartbeats and breathing patterns via subtle waveform analysis. We can assume that every R&D phase will itself leave behind an ecosystem of testing data and code which can be decisive for consolidating data models, directly or indirectly influencing production code for systems (even if their deployment and commercialization is well after the R&D period).

In the case of speech and language technology, a research-oriented data infrastructure has been systematically curated, in several subdisciplines, by academic or industry collaborations. The Conference on Natural Language Learning (CoNLL), for example, invites participants to develop Natural Language Processing techniques targeted at a common "challenge" dataset, updated each year. These data sets, along with the data formats and code libraries which allow software to use that data, thereby serve as a reference-point for Computational Linguistics researchers in general. Similarly, this volume's chapter on Language Identification describes research targeting a multilingual data set (labeled AP17-OLR) curated for an annual "Oriental Language Challenge" conference dedicated to language/dialect classification for languages spoken around East Asia (from East Asian language families and also Russian).

Technically, curated and publicly accessible data sets are a different genre of information space than real-time data generated by CyberPhysical technology (e.g. voices picked

formant data; components in different languages can then emulate the prototype, serving both as an implementation guide and a criterion for other developers to accept a new implementation as trustworthy. There are several examples of a reference implementation used as a standardizing tool, analogous to an Ontology, such as the LIBRETS (Real Estate Transaction Standard) library, servers and clients for FHIR (Fast Healthcare Interoperability Resources), and clients for DICOM (Digital Imaging and Communications in Medicine), e.g. for Whole Slide Imaging (see https://www.orthanc-server.com/static.php?page=wsi).

up by microphones in a Smart Home). That is to say, software developed to access speech and language data sets like the CoNLL's or the Oriental Language Challenge has different requirements than software responding to voice requests in real time — or other deployment use-cases, such as medical transcription, or identifying dialects spoken in an urban community. However, data models derived from publicly shared test corpora *do* translate over to realtime data: we can assume that R&D data sets are collections of signals or information granules which are structurally similar to those produced by operating CyberPhysical devices. As a result, portions of the software targeting R&D data sets — specifically, the procedures for acquiring, transforming, validating, and interactively displaying individual samples — remain useful as components or prototypes for deployed product. Code libraries employed in R&D cycles should typically be the basis for data models guiding the implementation of production software.

To make this discussion more concrete, I will use the example of CoNLL data sets. This chapter's demo includes samples from the most recent collection of CoNLL files and conference challenge tasks (at the time of writing) as well as demo code which operates on such data via techniques described in this chapter. The CoNLL format is representative of the kinds of linguistic parsing requisite for using Natural Language content (such as speech input) in CyberPhysical settings.

The chapter by Teixera *et. al.*, which I cited earlier, considers voice-activated CyberPhysical interfaces; in this context Natural Language segments become the core elements in translating user queries to actionable software responses. The proposed systems analyze speech patterns to build textual reconstructions of speakers' communications, then parses the text as Natural Language content, before eventually (if all goes well) interpreting the parsed and analyzed text as an instruction the software can follow. Text data can then be supplemented with metrics measuring vocal patterns, syntactic and semantic information, speaker's spatial location, and other information that can help interpret speakers' wishes insofar as software can respond to them.

The authors discuss, for instance, the possibility of annotating language samples (after speech-to-text translation) with Dependency Grammar parses.[3] Textual content can also be annotated with models of intonation, stress patterns, and other acoustic features (because the original inputs are audio-based) that can help an NLP engine to properly parse sentences (for instance by noting which words or syllables are vocally emphasized). So, in the context of integrated Smart Home hub software (continuing the above discussion), this is the kind of data which would be transmitted to a hub application. We can assume that audio processing as well as NLP technology would supply intermediary processing somewhere between acoustic devices and the centralized application.

Different kinds of linguistic details require different data models. Dependency parses, for instance, are often notated via some version of a specialized CoNLL format, which textually serializes parse and lexical data, usually one word per line. The most recent standard (dubbed CoNLL-U) recognizes ten fields for each word, identifying, in particular, Parts of Speech and syntactic connections with other words (see e.g. [28], [61], [66], [89], [117]). As a custom format, CoNLL-U requires its own parser, such as the UDPipe library for C++ (a slightly modified version of this library is published with this chapter's data set). So, for hub applications, a reasonable assumption is that these programs compile in the UDPipe library or an alternative with similar capabilities, in order for them to handle parsed NLP data.

Suppose, then, that a Smart Home speech-technology product suite bundles audio-capture devices with software that can perform dependency parsing, perhaps after training against users' speech and language patterns.[4] The NLP components — those which actually generate parses, as opposed to merely reading them — can be bundled with the acoustic devices, so that complex NLP code is isolated in its own software environment. Smart Home applications would not then compile NLP capabilities directly; instead, NLP features would be provisioned by a distinct program receiving audio input and generating text and parse transcriptions, which would subsequently be sent to hub applications, in lieu of raw device data. Let us assume that this architecture is in effect.

A hub application will, then, periodically receive a data package comprising an audio sample along with text transcriptions and NLP-generated, e.g., CoNLL-U data. To make sense of linguistic content, the software would presumably pair the NLP-specific information with extra details,

[3] Adequately describing Dependency Grammar is outside the scope of this chapter, but, in a nutshell, Dependency Grammar models syntax in terms of word-to-word relations rather than via phrase hierarchies; as such, Dependency parses yields directed, labeled graphs (node labels are words and edge labels are drawn from an inventory of syntactic inter-word connections), which are structurally similar to Semantic Web graphs. See also [1], [78], [91], [92], [101], [104], [129], etc.; variants include Link Grammar [58], [90] and Extensible Dependency Grammar [36], [37], [56].

[4] Users in this context meaning homeowners or other people expected often to be in the home: renters, children, health aides, and so on.

such as, the identity of the speaker (if a Smart Home system knows of specific users), where and when each sentence or request was formulated, and perhaps the original audio input (allowing functionality such as users playing back instructions they uttered in the past). A relevant data model might thereby comprise: (1) CoNLL-U data itself; (2) location info, such as spatial position and which room a speaker is found in; (3) timestamps; (4) speaker info, if available; (5) audio files; and maybe (6) extra acoustical or intonation data. Extra data could include annotations based on how conversation analysts notate speech patterns, or might be waveform features derived from initial processing of speech samples.

This data model would presumably translate to multiple data types: we can envision (1) a class for UDPipe sentences obtained from CoNLL-U; (2) a class for audio samples; (3) speaker and time/location information; plus versions of these classes appropriate for GUIs and database persistence. And, in addition, these data requirements for speech and text samples only considers obtaining a valid parse for the text; to actually react to speech input, an application would need to map lexical data to terms and actions the software itself, in the context of its own capabilities, can recognize. For instance, *close the window* would map to an identifier for which window is intended (inferred perhaps from speaker location) and a *close* operation, which could be available via actuators embedded in the window area. All of the objects that users might semantically reference in voice commands therefore need their own data models, which must be interoperable with linguistic parses. So along with data types specific to linguistic elements we can consider "bridge" types connecting linguistic data (e.g., lexemes) to data types modeling physical objects themselves.

Likewise, we can anticipate the procedures which speech and/or language data types need to implement: correctly decoding CoNLL-U files; mapping time/location data points to a spatial model of the Smart Home (which room is targeted by the location and also if that point is close to a door, window, appliance, and so forth; and perhaps matching the location to a 3D or panoramic-photography graphics model for visualization); audio-playback procedures, along with interactive protocols for this process such as users pausing and restarting playback; procedures to map identified speakers to user profiles known to the Smart Home system. The audio devise makers and NLP providers — assuming those products are delivered as one suite separate and apart from the Smart Home hub — can mandate that hub applications demonstrate procedural implementations that satisfy these requirements as a precondition for accessing their broadcast data. Conversely, hub applications can stipulate the procedural mandates they are prepared to honor as a guide to how devices and their drivers and middleware components should be configured for an integrated Smart Home ecosystem.

The essential point here is that procedural requirements and validation becomes the essential glue that unifies the diverse Smart Home components, and allows products designed by different companies, with different goals, to become interoperable. Once again, procedural alignment and predictability is more important than standardized data formats.

We can also consider representative criteria for testing procedures; for instance, preconditions that procedures need to recognize. In CoNLL-U, individual words can be extracted from a parse-model, but the numeric index for the word must fall within a fixed range (based on word count for the relevant sentence). In audio playback, time intervals are only meaningful in the context of the length of the audio sample in (e.g.) seconds or milliseconds. Similarly, features extracted from an audio sample (of human speech or, say, a bird song) are localized by time points which have to fit within a sample window; and image features are localized in rectangular coordinates that need to fit within the surrounding image. Therefore, procedures engaged with these data structures should be checked to ensure that they honor these ranges and properly respond to faulty data outside them. This is an example of the kind of localized procedural testing which, cumulatively, establishes software as trustworthy.

I will discuss similar procedural-validity issues for the remainder of this section before developing more abstract or theoretical models of procedures, as formal constructions, subsequently in the chapter.

2.3 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 procedures 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 procedures may make assumptions legislated by the relevant device's specifications, to the extent that using any one procedure too broadly constitutes a system error. Furthermore, CyberPhysical devices may exhibit errors due to mechanical malfunction, hostile attacks, or one-off errors in electrical-computing operations, causing

Data Type	Dimensions/Fields	Requirements
Blood Pressure	Beats per Minute; resting/active (boolean) or exercise level (scalar)	Systolic more than Diastolic; plausible resting/active range
Accelerator	Incline; Movement in 2 or 3 Dimensions	Biaxial or Triaxial
Audio Sample	Binary data/file and/or (waveform analysis) feature set	Length (in time) (1D)
Audio Feature	Time-interval inside sample and/or spectral numerics	Subinterval of sample (1D)
Image	Matrix of values in a colior model: RGB, RGBA, HSV, etc.	Matrix Dimensions (2D)
Image Segment	Rectangular Coordinates; geometric characterization (e.g. ellipse dimensions); scales of resolution where detectable	Subregion of image (2D)
Bioacoustic Sample	Audio Sample plus species identifier; geospatial and timestamp coordinates	Well-formed space-time coordinates
Speech Sample	Audio Sample plus text transcription; spacetime coodinates; identify language/ dialic and/or speaker	Valid dialect and/or speaker identifier
Dependency-Parsed Text	Text transcription plus parse serialization; audio metadata	Valid and accessible metadata
Lexical Text Component	Index into parse serialization; Part of Speech tag; lexical/semantic classification	Valid index

Table 1: Example Data Profiles

performance anomalies which look like software mistakes even if the code is entirely correct (see [46] and [99], 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.

Summarizing the case studies from earlier in this section, Table 1 identifies several details about dimensions, parameters of operation, data fields, and other pieces of information relevant to implementing procedures and data types capturing CyberPhysical data. These types may derive from CyberPhysical input directly or may model artifacts constructed from CyberPhysical input midstream, such as audio or image files, or text transcriptions representing speech input. The summaries are not rigorous data models, but are just suggestive cues about what sort of details engineers should consider when formalizing data models. In general, detailed models should be defined for any input source (including those transformed by middleware components, such as NLP engines), thereby profiling both CyberPhysical devices and also "midstream" artifacts such as audio or image files — i.e., aggregates, derived from CyberPhysical input, that can be shared between software components (within hub applications and/or between hubs and middleware).

These data profiles need to be integrated with CyberPhysical code from a perspective that cuts across multiple dimensions of project scale and lifetime. Do we design for biaxial or triaxial accelerometers, or both, and may this change? Is heart rate to be sampled in a context where the range considered normal is based on "resting" rate or is it expanded to factor in subjects who are exercising? These kinds of questions point to the multitude of subtle and project-specific specifications that have to be established when implementing and then deploying software systems in a domain like Ubiquitous Computing. It is unreasonable to expect that all relevant standards will be settled *a priori* by sufficiently monolithic and comprehensive data models. 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 procedures 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 (see also, as a case study, [11, pages 6-10]). 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.

"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.

I have argues that hub *libraries* — intermediaries between CyberPhysical devices and hub applications — are the key components where diverse requirements may be exercised. Hub libraries therefore need capabilities for documenting coding assumptions and requirements, such that their corresponding applications garner users' trust and acceptance. Hub applications, in short, would be deemed trustworthy insofar as their hub libraries are properly engineered. These, then, are the practical concerns driving the code-documentation proposals I will develop in the next two sections. Hub libraries are an environment where these techniques may be especially applicable.

3 Directed Hypergraphs and Generalized Lambda Calculus

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

Note that almost all non-trivial software is in some sense "procedural"; the total package of functionality provided by each software component is distributed among many individual, interconnected procedures. 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 alternative 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 functional types, specified by tuples of input 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 respective 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

— applying 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 these 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.

3.1 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 (compare 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 ([9] has similar interesting examples in the context of code optimization). This kind of tinkering with equations is of course a 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 repeatedly substituting nested expressions for free symbols? This question 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 unknowns 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.[5]

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 John von Neumann's formal prototypes for digital computers.

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 computing 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

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

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., [47], [65], [122]).

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 interchangeable, but it is important to represent 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 different 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.

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, procedures can "return" values not just by providing a final result but by throwing exceptions, writing to files or pipes, and so forth. To represent these myriad "channels of communication" computer scientists have invented a menagerie of extensions to λ-Calculus — a noteworthy 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 to 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 in the body of procedure implementations. For example, procedures 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.

My strategy for unifying multiple λ-calculi depends in turn on hypergraph code representations, which is a theme in the umbrella of graph-based data modeling, to which I now turn.

3.2 Directed Hypergraphs and "Channel Abstractions"

A *hypergraph* is a graph whose edges (a.k.a. "hyperedges") can span more than two nodes ([59, e.g. p. 24], [81], [86]; [88], [98], [108], [109]). 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).[6]

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

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

inferentially equivalent if any inference justified via edges incident to the first hypernode can be freely combined with inferences justified via edges incident to the second hypernode. Equatable nodes are not necessarily inferentially equivalent.

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

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

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

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

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

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

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

12. Call a *product-type transform* of a hypernode to be a different hypernode whose hypovertices are tuples of values equatable to those from the first hypernode, typed in terms of product types (i.e., tuples). For example, consider two different representations of semi-transparent colors: as a 4-vector **RGBT**, or as an **RGB** three-vector paired with a transparency magnitude. The second representation is a product-type transform of the first, because the first three values are grouped into a three-valued tuple. We can assert the requirement in most contexts that **CH**s 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 **CH**s. 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 **CH**s are flexible in part by morphing naturally into these various forms.

Notice that information present *within* a hypernode can also be expressed as relations *between* hypernodes. For example, consider the information that I (Nathaniel), age 46, live in Brooklyn as a registered Democrat. This may be represented as a hypernode with hyponodes $\langle \lceil \textbf{Nathaniel} \rceil, \lceil \textbf{46} \rceil \rangle$, 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

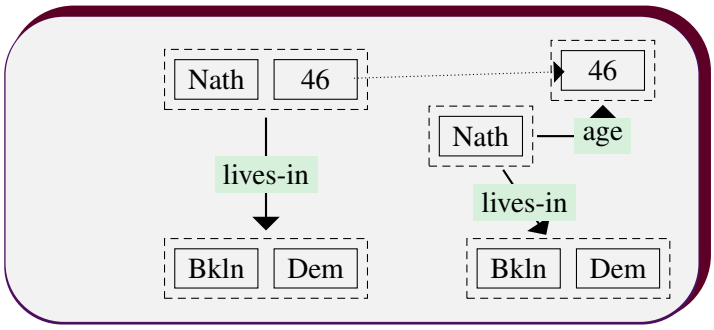

Diagram 1: Unplugging a Node.

a new edge, whose tail has a single hyponode ⌈45⌉ 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) ⟨⌈**Nathaniel**⌉, ⌈**46**⌉, ⌈**Brooklyn**⌉, ⌈**Democrat**⌉⟩ 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 ⟨⌈**Nathaniel**⌉, ⌈**46**⌉⟩ 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.

The two-tiered **DH** structure is also a factor when integrating serialized or shared data structures with runtime data values. In the demo **DH** library, for example, it is assumed that each node can be associated with a runtime, binary data allocation (practically speaking, a pointer to user data). Hypernodes' internal structure can therefore be represented *either* via hyponodes explicit in the graph content *or* by internal structure in the user data (or some combination). Graph deserialization can then be a matter of mapping hyponodes to fields in the "internal" data allocations, before then mapping

Sample 1: Initializing Hypernodes

```
caon_ptr<RE_Node> RE_Graph_Build::make_new_node(
  caon_ptr<RE_Function_Def_Entry> fdef)
{
 caon_ptr<RE_Node> result = new RE_Node(fdef);
 RELAE_SET_NODE_LABEL(result, "<fdef>");
 return result;                                    1
}
...
caon_ptr<RE_Node> RE_Graph_Build::
  new_function_def_entry_node(RE_Node& prior_node,
  RE_Function_Def_Kinds kind,
  caon_ptr<RE_Node> label_node)
{
 caon_ptr<RE_Function_Def_Entry> fdef = new        1
   RE_Function_Def_Entry(&prior_node,
   kind, label_node);
 caon_ptr<RE_Node> result = make_new_node(fdef);
 fdef->set_node(result);
 return result;
}
...
caon_ptr<RE_Node> RE_Graph_Build::create_tuple(
  RE_Tuple_Info::Tuple_Formations tf,
  RE_Tuple_Info::Tuple_Indicators ti,
  RE_Tuple_Info::Tuple_Formations sf,
  bool increment_id)
{
 int tuple_id = increment_id?++tuple_entry_count_:0;
 caon_ptr<RE_Tuple_Info> tinfo = new RE_Tuple_Info(
   tf, ti, tuple_id);
 caon_ptr<RE_Node> result = new RE_Node(tinfo);    1
 return result;
}
...
caon_ptr<RE_Node> RE_Markup_Position::
  check_implied_lambda_tuple(
  RE_Function_Def_Kinds kind)
{
...
 if(caon_ptr<RE_Call_Entry> rce =
   current_node_->re_call_entry())
 {
...
  caon_ptr<RE_Node> fdef_node = graph_build_->     2
    new_function_def_entry_node(
     *last_pre_entry_node_, kind);
  last_pre_entry_node_->delete_relation(
    rq_.Run_Call_Entry, current_node_);
  current_function_def_entry_node_ = fdef_node;
  caon_ptr<RE_Node> tuple_info_node = graph_build_->
2  create_tuple_node(
    RE_Tuple_Info::Tuple_Formations::Indicates_Input
    , RE_Tuple_Info::Tuple_Indicators::Enter_Array,
    , RE_Tuple_Info::Tuple_Formations::N_A );
  caon_ptr<RE_Node> entry_node =
    rq_.Run_Call_Entry(current_node_);              3
 ...
  fdef_node << fr_/rq_.Run_Call_Entry >>            4
    current_node_;
  current_node_ << fr_/rq_.Run_Data_Entry >>        4
    tuple_info_node;
  tuple_info_node << fr_/rq_.Run_Data_Entry >>      4
    entry_node;
...}}}
```

inter-hypernode relations to the proper hypervertex-relations. Code sample 1 demonstrates the pattern of hypervertex construction as C++ objects that get wrapped in new nodes (❶-❷), along with obtaining nodes already registered in a runtime graph (❸) and then inserting the new nodes (with stated relationships) alongside prior ones into the runtime graph (❸).

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.

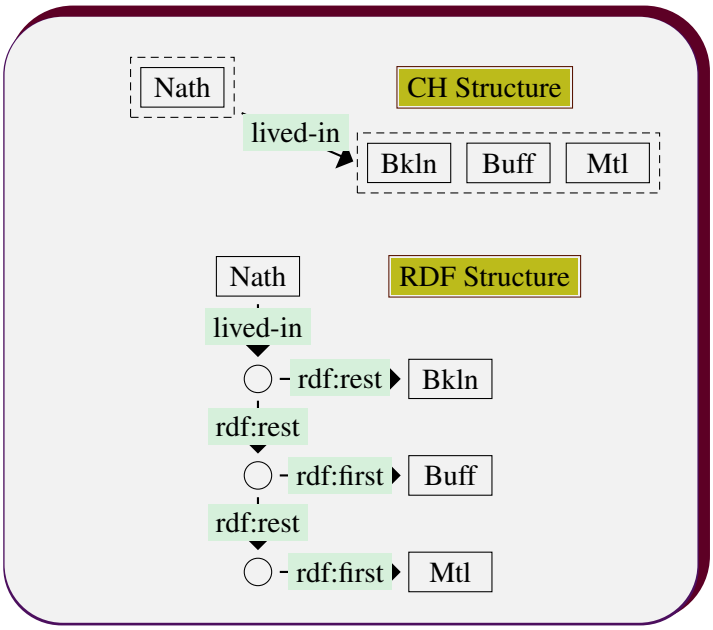

Diagram 2: CH vs. RDF Collections.

3.3 Channelized Hypergraphs and RDF

The Resource Description Framework (RDF) models information via directed graphs ([63], [34], and [35] 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.

For example, imagine a representation of the fact "(A/The person named) Nathaniel, 46, has lived in Brooklyn, Buffalo, and Montreal" (shown in Diagram 2 as both a CH and in 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" [20, 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 [125]). 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**⌉, ⌈**46**⌉⟩ 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 my 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 46-year-olds named Nathaniel, so *that* blank node plays a key semantic role (analogous to the quantifier in "*There is* a Nathaniel, age 46..."); it provides an unambiguous nexus so that further predicates can be attached to *one specific* 46-year-old Nathaniel rather than any old ⟨⌈**Nathaniel**⌉, ⌈**46**⌉⟩. 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? Moreover, defining hierarchies of material composition or structural organization — and so by extension, potentially, distinct scales of modeling resolution — has been identified as an intrinsic part of domain-specific Ontology design (see [10], [21], [22], [39], [48], [97], [105], [106], or [116]). Semantic Web advocates have not however promoted multitier structure as a feature *of* Semantic models fundamentally, as opposed to criteriology *within* specific Ontologies. To the degree that this has an explanation, 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 (it may be useful to see CH hypernodes as roughly analogous to C structs — which have no *a priori* uniqueness mechanism).

Also, RDF types are less intrinsic to RDF semantics than in CH (see [94]). 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 $\langle \lceil \mathsf{Nathaniel} \rceil, \lceil \mathsf{46} \rceil \rangle$ 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 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 46 in one graph is indistinguishable from 46 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 2018 goal totals". One way to block these inferences is to prevent nodes like "the number 46" 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 between "declarative knowledge" and "procedural knowledge" (see e.g. [59, 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, such as 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.

Despite their differences, the Semantic Web, on the one hand, and Hypergraph-based frameworks, on the other, both belong to the overall space of graph-oriented semantic models. Hypergraphs can be emulated in **RDF**, and **RDF** graphs can be organically mapped to a Hypegraph representation (insofar as Directed Hypegraphs with annotations are a proper superspace of Directed Labeled Graphs). Semantic Web Ontologies for computer source code can thus be modeled by suitably typed **DH**s as well, even while we can also formulate Hypergraph-based Source Code Ontologies as well. So, we are justified in assuming that a sufficient Ontology exists for most or all programming languages. This means that, for any given procedure, we can 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 Calculi with Source Code Ontologies. The essential construct in λ-calculi 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, linking 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 by-reference or by-value captures. 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, captures, 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.

3.4 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 t_2 from a type t_1 — note how this is different from a morphism which maps *values of* t_1 to 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).[7] 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 [25]. 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 $\mathbb{T}$ may to be "closed" under products: if t_1 and t_2 are in $\mathbb{T}$ then $t_1 \times t_2$ must be as well. Analogously, $\mathbb{T}$ supports function-like types if it is closed under a kind of "functionalization" operator: if the $t_1 \times t_2$ product can be mapped onto a function-like type $t_1 \rightarrow t_2$.

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 t_1 and t_2 are in $\mathbb{T}$ then so is the combination of t_1 and 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-like 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 **DH**s 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

[7] 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).

further.

3.4.1 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 λ-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 $\not\searrow_{\mathbf{LAM}}.xy$.

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 that may share values. Here are several examples:

- In most Object-Oriented languages, any procedure can have at most one **this** ("message receiver") object. Let $\not\searrow_{\mathbf{SIG}}$ model a "Sigma" channel, as in "Sigma Calculus" (written as ς-calculus: see e.g. [2], [26], [51], [131], etc.). We then have the requirement than any procedure's $\not\searrow_{\mathbf{SIG}}$ 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 cannot 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 "$\not\searrow_{\mathbf{SIG}}$" 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' $\not\searrow_{\mathbf{SIG}}$ 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 — **lambda+return**, say — therefore models the basic λ-Calculus and, simultaneously, the minimal theory of function-like types (for Cartesian Closed type Categories). In essence: a procedure's signature, or type-expression, can be seen as a "sum of channels", or an inter-channel operation. Notionally, a procedure is, in the

simplest conceptualization, the unification of an input channel and an output channel. So a "channel sum" creates the basic foundation for a procedure, analogous to how input and output graph elements yield the foundations for morphisms in Hypergraph Categories. More complex channel combinations and protocols can then model more complex variations on λ-Calculi and programming language type systems.

3.4.2 Channelized Type Systems

Collectively, to summarize my discussion 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 procedures, 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 λ-Calculus can be described as a χ-language for the algebra defined by a minimal **lambda+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 **DH**s whose type system is Channelized. We can think of channel constructions as operators which combine groups of types into new types. Once we assert that a **CH** is Channelized, we know that there is a mechanism for describing some Hypergraphs or subgraphs as "procedure 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 [74], [76], [80], [123], [127], [128]). 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 **TXL**s. Formal declarations of channel axioms — e.g., restrictions on channel sizes, alone or in combination — are examples of terms that should be representable in a **CXL**. However, whereas the **CXL** expressions I have described 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 employing 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.

3.5 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.[8]

Types are, then, conceptually distinct from sets of values, and type theory should be developed with no appeal to sets or to set theory. We therefore need a mechanism to specify and individuate types — e.g., under what circumstance do two different symbols or two different **TXL** terms designate "the same" type? Here I will resolve questions about type-identity via functional resolutions: functions are overloaded insofar as one symbol can designate multiple functional values, which in turn are distinguished by type differences somewhere in their signature. Consequently, types themselves are differentiated insofar as they engender distinct signatures. If and only if two expressions designate the same type, they can be interchanged in a signature without altering the function-like type which the signature demarcates.

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.[9] 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).[10] 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 $t_1 \to t_2 \to t_3$ and $t_1 \to (t_2 \to t_3)$ 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 $t_1 \to t_2 \to t_3$ and $t_1 \to (t_2 \to t_3)$. Languages *may* be engineered to silently Curry any function on demand, but purported $t_1 \to t_2 \to t_3$ and $t_1 \to (t_2 \to t_3)$ 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

[8] Similar issues are sometimes addressed by a *modal* type theory (cf., e.g., [54]) 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").

[9] 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**⟩).

[10] 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).

how the word "function" is used. If I say that "there exists a function from t_1 to t_2", where t_1 and 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 $t_1 \Rightarrow t_2$ mapping by appeal to some sort of logic; the other is that a $t_1 \Rightarrow t_2$ function actually exists in code. This is not just a "metalanguage" difference projected 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 41 — 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 a code-graph — say, an Abstract Semantic Graph (or similar graph-like Intermediate Representation), generalizing from an Abstract Syntax Tree. 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 every code-span suitable for compilation into a function-implementation can be isolated as a separate graph or subgraph, logically adjoined to one or more functional values (viz., those functions instanced in procedures described by the graph). Assume also that every source code literal is equivalent to a **code-DH** graph with one hypernode and no edges. In that case, initializing values from literals is one example of initializing values from code-graph instances more generally.

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, most 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 values (or, in my terminology, 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 then 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.[11]

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.

3.6 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

[11] Here I depart from the usual account of Hypergraph Categories, because I assume that functions between types are not, in the general case, recognized as morphisms in types qua objects in a type Category. This is partly because Category-theory morphisms have no analogs to "channels" other than **lambda** and **return**.

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**⟩. Ordinarily, we think of value constructors as procedures which yield values of their intended types. Given the bijection (once types are fixed) between procedure implementations and code-graphs, each (concretely) inhabitable type would then have at least one value constructor which is unique to that type: a type-plus-code-graph pair. Conversely, one code-graph can engender multiple procedures by varying carrier types. This allows one code graph to be mapped to multiple value constructors. But, this generalization does not (without supplemental logic) support a parallel generalization where one value constructor would 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 derived from 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 "com-

mon functionality" base function *plus* a range-argument to be passed to it — some sort of ⟨**&Cf, r**$_1$**, 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 be obtained from one sort of origin: code graphs. That is, for each metaconstructor — again, a de-facto 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.[12] 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 (see e.g. [19, p. 4]): 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 an anonymous function passed to a procedure is defined 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 an anonymous-function block, but some languages may allow a more "algebraic" expression, something meaning "lowercase then less-than", with the idea that function values can be composed by rough analogy to numbers being added (see item 3 on page 41). 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

[12] I am not specifically assuming some sort of on-the-fly compilation where new functions could be created from source code at runtime. That is, referring to "metaconstructors" as value-constructors may be suggestive more than literal, technical terminology. On the other hand, if a language *can* dynamically read code-graphs, this usage should be understood more literally: we might assume that there is a specific type representing code-graphs, and values of that type are then inputs to value-constructors yielding procedure types.

to "inductive typing" as discussed by Edwin Brady in the context of the Idris language [24, 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.

In a nutshell: without restricting their expressiveness, Dependent Types can only be achieved by modifying foundational assumptions about the relationships between types and the procedures which construct their instances. We then have a choice between simply accepting an expanded model of value-construction or devising strategies for emulating Dependent Types within the confines of a code model that preserves, in particular, value-constructor addressability. The techniques I discuss in this chapter are adjusted to the latter decision.

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 **int⦅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 gatekeeping 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.

3.7 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).

In short, typestate forces us to modify some basic assumptions about the relationship between types and values: either values can change types mid-stream, or a lexical scope can subsume a sequence of carrier which share the same symbol-name (and maybe the same type) but differ in state (some holding values unrelated to actual program state). This situation can be juxtaposed with the "metaconstructor problem", i.e., how Dependent Types force a rethink on basic value-constructor theory. As with constructors, here I will advocate for techniques that emulate typestate without unduly expanding the scope of a Channelized type theory.

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 [69]:

> 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 [114, 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 [31], and [44, page 210]). This means that much 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 runpoints, where extra code can be added to program flow (see e.g. [60], [85], [132]). 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 analogous requirements manifest as type-states). In practice, working with scenarios such as range-bounded values — which in principle exemplifies programming tasks where Dependent Types can be a useful idiom — 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 range restrictions 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).[13] So the semantics of Dependent Types can practically be captured via a typestate framework — and perhaps

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

vice-versa, since typestates can be seen as type families indexed over (possibly enumerative) types; e.g., file typestates as a family indexed over ⟨**closed**, **open**, **nonexistent**⟩.

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 Groups more substantially.

4 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.

4.1 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**$\circ$**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).

4.2 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 #includes). 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 (mathematically, 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.[14]

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 mini-

[14] 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.

mize 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 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.

4.3 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.[15]

Types are, then, conceptually distinct from sets of values, and type theory should be developed with no appeal to sets or to set theory. We therefore need a mechanism to specify and individuate types — e.g., under what circumstance do two different symbols or two different TXL terms designate "the same" type? Here I will resolve questions about type-identity via functional resolutions: functions are overloaded insofar as one symbol can designate multiple functional values, which in turn are distinguished by type differences somewhere in their signature. Consequently, types themselves are differentiated insofar as they engender distinct signatures. If and only if two expressions designate the same type, they can be interchanged in a signature without altering

[15] Similar issues are sometimes addressed by a *modal* type theory (cf., e.g., [54]) 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").

the function-like type which the signature demarcates.

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.[16] 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).[17] 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 $t_1 \to t_2 \to t_3$ and $t_1 \to (t_2 \to t_3)$ 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 $t_1 \to t_2 \to t_3$ and $t_1 \to (t_2 \to t_3)$. Languages *may* be engineered to silently Curry any function on demand, but purported $t_1 \to t_2 \to t_3$ and $t_1 \to (t_2 \to t_3)$ 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 t_1 to t_2", where t_1 and 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 $t_1 \Rightarrow t_2$ mapping by appeal to some sort of logic; the other is that a $t_1 \Rightarrow t_2$ function actually exists in code. This is not just a "metalanguage" difference projected 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 41 — 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 a code-graph — say, an Abstract Semantic Graph (or similar graph-like Intermediate Representation), generalizing from an Abstract Syntax Tree. 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 every code-span suitable for compilation into a function-implementation can be isolated as a separate graph or subgraph, logically adjoined to one or more functional values (viz., those functions instanced in procedures described by the graph). Assume also that every source code literal is equivalent to a **code-DH** graph with one hypernode and no edges. In that case, initializing values from literals is one example of initializing values from code-graph instances more generally.

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, most 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 re-

[16] 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**⟩).

[17] 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).

ceive 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 values (or, in my terminology, 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 then 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.[18]

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.

4.4 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⟩. Ordinarily, we think of value constructors as procedures which yield values of their intended types. Given the bijection (once types are fixed) between procedure implementations and code-graphs, each (concretely) inhabitable type would then have at least one value constructor which is unique to that type: a type-plus-

[18] Here I depart from the usual account of Hypergraph Categories, because I assume that functions between types are not, in the general case, recognized as morphisms in types qua objects in a type Category. This is partly because Category-theory morphisms have no analogs to "channels" other than **lambda** and **return**.

code-graph pair. Conversely, one code-graph can engender multiple procedures by varying carrier types. This allows one code graph to be mapped to multiple value constructors. But, this generalization does not (without supplemental logic) support a parallel generalization where one value constructor would 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 derived from 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, r**$_1$**, 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 be obtained from one sort of origin: code graphs. That is, for each metaconstructor — again, a de-facto 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.[19] 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

[19] I am not specifically assuming some sort of on-the-fly compilation where new functions could be created from source code at runtime. That is, referring to "metaconstructors" as value-constructors may be suggestive more than literal, technical terminology. On the other hand, if a language *can* dynamically read code-graphs, this usage should be understood more literally: we might assume that there is a specific type representing code-graphs, and values of that type are then inputs to value-constructors yielding procedure types.

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 (see e.g. [19, p. 4]): 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 an anonymous function passed to a procedure is defined 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 an anonymous-function block, but some languages may allow a more "algebraic" expression, something meaning "lowercase then less-than", with the idea that function values can be composed by rough analogy to numbers being added (see item 3 on page 41). 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 [24, 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.

In a nutshell: without restricting their expressiveness, Dependent Types can only be achieved by modifying foundational assumptions about the relationships between types and the procedures which construct their instances. We then have a choice between simply accepting an expanded model of value-construction or devising strategies for emulating Dependent Types within the confines of a code model that preserves, in particular, value-constructor addressability. The techniques I discuss in this chapter are adjusted to the latter decision.

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 gatekeeping 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.

4.5 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).

In short, typestate forces us to modify some basic assumptions about the relationship between types and values: either values can change types mid-stream, or a lexical scope can subsume a sequence of carrier which share the same symbol-name (and maybe the same type) but differ in state (some holding values unrelated to actual program state). This situation can be juxtaposed with the "metaconstructor problem", i.e., how Dependent Types force a rethink on basic value-constructor theory. As with constructors, here I will advocate for techniques that emulate typestate without unduly expanding the scope of a Channelized type theory.

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 [69]:

> 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 [114, 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 [31], and [44, page 210]). This means that much 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. [60], [85], [132]). 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 analogous requirements manifest as type-states). In practice, working with scenarios such as range-bounded values — which in principle exemplifies programming tasks where Dependent Types can be a useful idiom — 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 range restrictions 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).[20] 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; e.g., file typestates as a family indexed over ⟨**closed**, **open**, **nonexistent**⟩.

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 Groups more substantially.

5 Conclusion

Both type theory and Semantic Web style Ontologies pose fundamental questions about data modeling — about how digitized data structures can capture the nuances and detail of human and scientific concepts. Ideally, data models are expressive enough to represent concepts and artifacts, drawn from our cultural and scientific domains, without any sense of conceptual mismatch or simplification; but at the same time work in a software ecosystem, where data structures have sufficient predictability and classification that they are

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

amenable to algorithms and mutations to accommodate different software roles, such as database and GUI presentations. Software users should not feel as if they have to wrangle real-world data into awkward formats, in order to introduce information they deem worthy of being shared and studied into a digital ecosystem. On the other hand, digital resources should have enough structure and planning to be accessible to high-quality software, with optimized User Interface design and responsiveness, as well as trustworthy safety and privacy features.

Achieving all of these goals involves a certain balancing act, where data repositories are modeled via expressive, fine-grained prototypes without becoming too unstructured, or too heterogeneous, for rigorous software implementations. The technical terrain of Ontology-based or type-theoretic modeling can therefore be seen as a drive to expand models' expressiveness as far as possible, but without losing models' underlying formal rigor and tractability. In terms of data models, this can be reflected in the evolution from fixed-field structures (like spreadsheets and relational databases) to labeled-graph Ontologies to Hypergraphs and other multi-scale graph paradigms. Parallel to the emergence of Semantic Web technology there is also a body of research in Scientific Computing, where expressiveness translates to modeling strategy which encapsulate scientific theories and workflows — cf. Object-Oriented simulations ([118], [119] being a good case-study) and such formats or approaches as Conceptual Space theory (in science and linguistics) and Conceptual Space Markup Language ([3], [4], [5], [40], [55], [67], [100], [111] — a perspective integrating Hypergraphs and Conceptual Space theory, taking its departure from recent research on Hypergraph categories as a syntax for Conceptual Space semantics, is available as [30] in the demo). Meanwhile, in type theory, a similar impetus leads from the simple type systems of Typed Lambda Calculus through to Dependent Types, typestate, effect systems, Object-Orientation, and other features of modern programming environments.

Whatever their features, data models are ultimately only as usable as the software that receives them. Applications may be receiving CyberPhysical measurements "in real time" or affording access to archived research data sets, but in each case the structured formats of shared and/or persisted information must be transformed into interactive, usually GUI-based presentations for applications to qualify as productive viewers onto the relevant information space. This is how we should understand the criterion of expressiveness: expressiveness at the modeling level is a means to an end; the ultimate goal is "expressive" software, i.e., software whose layout, visual presentations, and interactive features/responsiveness render applications effective vehicles for interfacing with complex, nuanced digital content. Ultimately, then, data models are effective to the extent that they promote effective software engineering for the applications that transform modeled data into user-facing digital content.

On the other hand, this leaves room for differences in what is prioritized: data models can be targeted at a narrow, specialized set of software end-points, or can be designed flexibly to work with a diversity of software products, in the present and going forward. Broader application-scope is desirable in theory, but practically speaking a data model which is open-ended enough to work with a range of software components is potentially too provisional, or insufficiently detailed, to promote the highest-quality software.

Information Technology in the last one or two decades seem to have favored general-purpose data models — or at least serialization techniques — which exist in isolation from applications that work with them. Canonical examples would be JSON, XML, and RDF. Conceptually, however, data models' most important manifestation are in the software components where they are shared — sent (perhaps indirectly via a generated archive) and received. To the degree that multi-purpose formats like XML are beneficial, there merits are in part that developers can anticipate the code that generated and/or will receive the data: while programmers do not necessarily just write code off of an XML sample (or corresponding Document Type Declaration), any XML document or DTD gives us a rough idea of what its client code would look like.

Nevertheless, for robust software engineering we should aspire to something more rigorous than that. In effect, we should consider documentation of components which send and/or receive data structures to be an intrinsic aspect of rigorous data modeling itself: description of the procedures which construct, serialize/deserialize, validate, and transform data structures, particularly those procedures supplying functionality determinative of their components' ability to be part of a conformant data-sharing network. In this sense data and code modeling coincide. In particular, characterization of individual procedures — their types, assumptions, and requirements — is an essential building-block of data models generally. Data structures can be indirectly systematized in terms of the procedures which act upon them.

With this background, the demo proposes a notion of "Procedural Hypergraph Ontology" which extends (or diverges from) conventional Semantic Web Ontology partly by orienting toward Hypergraphs, but more substantially

by centering on this procedural dimension: the role of an Ontology being to describe components' procedural interface as well as their targeted data structures.

In particular, the demo presents both a hypergraph serialization format and methodology for generating interface descriptions, based on channel complexes. The demo code shows a compilation process which works with channel groups, branching off into a runtime engine which actually evaluates channel packages and, separately, algorithms to compile information about procedure signatures and function calls. This last capability can be a point for embedding more detailed Interface Definition metadata, including via the non-standard channel protocols I have discussed in this chapter. Both static data structures and compiled channel groups translate to a Hypergraph format, which thereby serves as a common denominator between code and data.

Rigorous procedural documentation, or Interface Definition, can then serve several different roles in application development, including testing, Requirements Engineering, and GUI design. In particular, a formal review of important procedures exposed by a software component allows front-end development to identify which operational features need to covered by interactive User Interface components; for example, what units of functionality should be linked to buttons, context menus, and other responsive design elements.

Extending Interface Definition outward to front-end GUI layers then presents the challenge of integrating rigorous data-modeling paradigms with quality front-end software affordances. Whether or not by technical necessity or just entrenched culture, GUI frameworks are still predominantly built in Procedural or Object-Oriented 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.

Contemporary software engineering still seems caught in a paradigm split, with Functional and Object-Oriented styles seen as competitors rather than candidates for admixture, and with a profound divergence between code libraries targeting native, desktop applications and those designed for

the web ecosystem. I believe that further research in programming language design and software engineering methodology will however reveals these divisions to be preliminary, and a new generation of web/desktop and Object/Functional hybrid paradigms can emerge. Given the unique requirements of the CyberPhysical domain, perhaps CyberPhysical and Ubiquitous Sensing technology will be a momentum boost for this kind of behind-the-scenes research.

References

1 Frank Abromeit and Christian Chiarcos, "Automatic Detection of Language and Annotation Model Information in CoNLL Corpora". http://lucacardelli.name/Papers/PrimObjSemLICS.A4.pdf

2 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

3 Benjamin Adams and Martin Raubal, "A Metric Conceptual Space Algebra". https://pdfs.semanticscholar.org/521a/cbab9658df27acd9f40bba2b9445f75d681c.pdf

4 Benjamin Adams and Martin Raubal, "Conceptual Space Markup Language (CSML): Towards the Cognitive Semantic Web". http://idwebhost-202-147.ethz.ch/Publications/RefConferences/ICSC_2009_AdamsRaubal_Camera-FINAL.pdf

5 Benjamin Adams and Martin Raubal, "The Semantic Web Needs More Cognition". http://www.semantic-web-journal.net/sites/default/files/swj37_0.pdf

6 Fadel Adib, *et. al.*, "Smart Homes that Monitor Breathing and Heart Rate". http://witrack.csail.mit.edu/vitalradio/content/vitalradio-paper.pdf

7 Firas Albalas, *et. al.*, "Security-aware CoAP Application Layer Protocol for the Internet of Things using Elliptic-Curve Cryptography". In *The International Arab Journal of Information Technology*, Vol. 15, No. 3A, Special Issue 2018 https://www.researchgate.net/publication/325987571_Security-aware_CoAP_Application_Layer_Protocol_for_the_Internet_of_Things_using_Elliptic-Curve_Cryptography

8 Marco Altini, "Combining Wearable Accelerometer and Physiological Data for Activity and Energy Expenditure Estimation". https://www.marcoaltini.com/uploads/1/3/2/3/13234002/1569766907-altini.pdf

9 Kenneth R. Anderson, "Freeing the Essence of a Computation". http://repository.readscheme.org/ftp/papers/kranderson_essence.pdf

10 Gonzalo A. Aranda-Corral and Joaquín Borrego-Díaz, "Mereotopological Analysis of Formal Concepts in Security Ontologies." In *Computational Intelligence in Security for Information Systems*, Herrero Á, Corchado E., Redondo C., Alonso Á, eds. (Advances in Intelligent and Soft Computing, vol 85), Springer, Berlin, Heidelberg, 2010. https://core.ac.uk/download/pdf/158966553.pdf

11 Flávia Linhalis Arantes, "Requirements Engineering of a Web Portal Using Organizational Semiotics Artifacts". https://arxiv.org/pdf/1305.3255.pdf

12 Ronald Ashri, *et. al.*, "Torwards a Semantic Web Security Infrastructure". American Association for Artificial Intelligence, 2004. https://www.aaai.org/Papers/Symposia/Spring/2004/SS-04-06/SS04-06-012.pdf

13 Louis Auguste and Dhaval Palsana, "Mobile Whole Slide Imaging (mWSI): a low resource acquisition and transport technique for microscopic pathological specimens". https://www.researchgate.net/publication/279276605_Mobile_Whole_Slide_Imaging_mWSI_A_low_resource_acquisition_and_transport_technique_for_microscopic_pathological_specimens

14 Christoph Becker, *et. al.*, "Sustainability Design and Software: The Karlskrona Manifesto". http://www.cs.toronto.edu/~sme/papers/2015/Beckeretal-ICSE2015.pdf

15 Khalid Belhajjame, *et. al.*, "Using a suite of ontologies for preserving workflow-centric research objects". *Web Semantics: Science, Services and Agents on the World Wide Web*, 2015 https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3199184

16 Khalid Belhajjame, *et. al.*, "Workflow-Centric Research Objects: First Class Citizens in Scholarly Discourse". http://ceur-ws.org/Vol-903/paper-01.pdf

17 Eran Bellin, *et. al.*, "Democratizing Information Creation From Health Care Data for Quality Improvement, Research, and Education – The Montefiore Medical Center Experience" https://pdfs.semanticscholar.org/ad02/adebdfe8d51c6defb120aac9f6f102e16596.pdf

18 Vincas Benevičius, *et. al.*, "Finite element model of MEMS accelerometer for accurate prediction of dynamic characteristics in biomechanical applications ". https://www.jvejournals.com/article/10526/pdf

19 Jean-Philippe Bernardy, et. al., "Parametricity and Dependent Types". http://www.staff.city.ac.uk/~ross/papers/pts.pdf

20 Tim Berners-Lee, "N3Logic: A Logical Framework For the World Wide Web". 2007. https://arxiv.org/pdf/0711.1533.pdf

21 Thomas Bittner, Barry Smith, and Maureen Donnelly, "The logic of systems of granular partitions." http://ontology.buffalo.edu/smith/articles/BittnerSmithDonnelly.pdf

22 Thomas Bittner and Barry Smith "A taxonomy of granular partitions." http://qrg.northwestern.edu/papers/Files/Bittner_Smith_Taxonomy_granular_partitions.pdf

23 Philip E. Bourne, et. al., "Improving The Future of Research Communications and e-Scholarship". *Manifesto from Dagstuhl*

Perspectives Workshop 11331 http://drops.dagstuhl.de/opus/volltexte/2012/3445/pdf/dagman_v001_i001_p041_11331.pdf

24 Edwin Brady, "Idris, a General Purpose Dependently Typed Programming Language: Design and Implementation". 2013. https://pdfs.semanticscholar.org/1407/220ca09070233dca256433430d29e5321dc2.pdf

25 R. Brown, *et. al.*, "Graphs of Morphisms of Graphs". https://www.emis.de/journals/EJC/Volume_15/PDF/v15i1a1.pdf

26 Joana Campos and Vasco T. Vasconcelos, "Channels as Objects in Concurrent Object-Oriented Programming" https://arxiv.org/pdf/1110.4157.pdf

27 Wei-Lun Chao, "Face Recognition". http://disp.ee.ntu.edu.tw/~pujols/Face%20Recognition-survey.pdf

28 Christian Chiarcos and Niko Schenk, "The ACoLi CoNLL Libraries: Beyond Tab-Separated Values". https://aclweb.org/anthology/L18-1090

29 Nathaniel Christen, "Hypergraph Data Modeling and a Hypergraph Virtual Machine". Published as part of this chapter's data set. https://github.com/scignscape/ntxh/papers/hgdm/hgdm.pdf

30 Nathaniel Christen, "Channelized Hypergraphs and Conceptual Space Theory". Published as part of this chapter's data set. https://github.com/scignscape/ntxh/papers/chcs/chcs.pdf

31 David Raymond Christiansen, "Practical Reflection and Metaprogramming for Dependent Types". Dissertation, IT University of Copenhagen, 2015. http://davidchristiansen.dk/david-christiansen-phd.pdf

32 Jongyoon Choi and Ricardo Gutierrez-Osuna, "Using Heart Rate Monitors to Detect Mental Stress". http://research.cse.tamu.edu/prism/publications/bsn09_choi.pdf

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

34 Madalina Croitoru and Ernesto Compatangelo, "Ontology Constraint Satisfaction Problems using Conceptual Graphs". https://pdfs.semanticscholar.org/d05e/eb82298201d6fae0129c6d53fe16db6d4803.pdf

35 Ernesto Damiani, *et. al.*, "Modeling Semistructured Data by Using Graph-based Constraints". http://home.deib.polimi.it/schreibe/TeSI/Materials/Tanca/PDFTanca/csse.pdf

36 Ralph Debusmann, "Extensible Dependency Grammar: A Modular Grammar Formalism Based On Multigraph Description." Universität des Saarlandes, dissertation 2006.

37 Ralph Debusmann, Denys Duchier and Andreas Rossberg, "Modular Grammar Design with Typed Parametric Principles". James Rogers, ed., *Formal Grammar/Mathematics of Language 2005*, CSLI Publications, 2009. http://web.stanford.edu/group/cslipublications/cslipublications/FG/2005/debusmann.pdf.

38 Badis Djamaa, *et. al.*, "Hybrid CoAP-based Resource Discovery for the Internet of Things". https://dspace.lib.cranfield.ac.uk/bitstream/handle/1826/11602/Hybrid_CoAP-based_resource_discovery-Internet_of_Things-2017.pdf?sequence=3&isAllowed=y

39 Maureen Donnelly, *et. al.*, "A Formal Theory for Spatial Representation and Reasoning in Biomedical Ontologies". http://www.acsu.buffalo.edu/~md63/DonnellyAIMed05.pdf

40 Igor Douven, et. al., "Vagueness: A Conceptual spaces approach" https://www.researchgate.net/publication/225689962_Vagueness_A_Conceptual_Spaces_Approach

41 Yueqi Duan, et. al., "Topology Preserving Graph Matching for Partial Face Recognition". In *Proceedings of the IEEE International Conference on Multimedia and Expo (ICME)*, 2017. http://ivg.au.tsinghua.edu.cn/people/Yueqi_Duan/ICME17_Topology%20Preserving%20Graph%20Matching%20for%20Partial%20Face%20Recognition.pdf

42 Abhishek Dwivedi, et. al., "Cancellable Biometrics for Security and Privacy Enforcement on Semantic Web" https://pdfs.semanticscholar.org/7c7c/957edf8dd1dcb2c5baf315021d6fc387d030.pdf

43 Herbert Edelsbrunner and John Harer, "Persistent Homology — a Survey". https://www.maths.ed.ac.uk/~v1ranick/papers/edelhare.pdf

44 Richard A. Eisenberg, "Dependent Types in Haskell: Theory and Practice". Dissertation, University of Pennsylvania, 2017. http://www.cis.upenn.edu/~sweirich/papers/eisenberg-thesis.pdf

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

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

47 Martín Escardó and Weng Kin Ho, "Operational domain theory and topology of sequential programming languages". Information and Computation 207 (2009), pp. 411-437. https://ac.els-cdn.com/S0890540108001570/1-s2.0-S0890540108001570-main.pdf?_tid=94a23ca4-2048-44f5-8b28-50e885faaa40&acdnat=1533048081_6911dea49597f7c7e184e5e30ae3e773f

48 Sara Irina Fabrikant, "Visualizing Region and Scale in Information Spaces". https://www.semanticscholar.org/paper/VISUALIZING-REGION-AND-SCALE-IN-INFORMATION-SPACES-Fabrikant/526a09e4767ff634c4cfbc51e6f7f4ebb700096a

49 Farahani N, *et. al.*, "Whole slide imaging in pathology: advantages, limitations, and emerging perspectives". https://www.dovepress.com/whole-slide-imaging-in-pathology-advantages-limitations/-and-emerging-p-peer-reviewed-article-PLMI

50 Katrina Fenlon, "Modeling Digital Humanities Collections as Research Objects". https://drum.lib.umd.edu/bitstream/handle/1903/21860/fenlon_jcdl2019_researchObjects_final.pdf?sequence=1&isAllowed=y

51 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

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

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

54 Murdoch J. Gabbay and Aleksandar Nanevski, "Denotation of Contextual Modal Type Theory (CMTT): Syntax and Metaprogramming". https://software.imdea.org/~aleks/papers/cmtt/cmtt-semantics.pdf

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

56 Michael Gasser, "Toward Synchronous Extensible Dependency Grammar". http://openaccess.uoc.edu/webapps/o2/bitstream/10609/5643/3/Gasser_Freerbmt11_Toward.pdf

57 Riccardo Giambona, *et. al.*, "MQTT+: Enhanced Syntax and Broker Functionalities for Data Filtering, Processing and Aggregation". https://arxiv.org/pdf/1810.00773.pdf

58 Ben Goertzel, "Probabilistic Language Networks: Integrating Word Grammar and Link Grammar in the Framework of Probabilistic Logic". http://goertzel.org/ProwlGrammar.pdf

59 Ben Goetzel, *et. al.*, "Engineering General Intelligence", Parts 1 & 2. Atlantis Press, 2014. http://wiki.opencog.org/w/Background_Publications

60 Dinesh Gopalani, *et. al.*, "A Type System and Type Soundness for the Calculus of Aspect-Oriented Programming Languages". Proceedings of the International MultiConference of Engineers and Computer Scientists (IMECS 2012) Vol 1, March 14-16, Hong Kong. http://www.iaeng.org/publication/IMECS2012/IMECS2012_pp263-268.pdf

61 Johannes Graën, *et. al.*, "Modelling Large Parallel Corpora: The Zurich Parallel Corpus Collection". http://corpora.ids-mannheim.de/CMLC7-final/CMLC-7_2019-Graen_et_al.pdf

62 Cenk Gündoğann, *et. al.*, "NDN, CoAP, and MQTT: A Comparative Measurement Study in the IoT". https://arxiv.org/pdf/1806.01444.pdf

63 Renzo Angles and Claudio Guttierez, "Querying RDF Data from a Graph Database Perspective". 2005. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.70.225&rep=rep1&type=pdf

64 Jussi Haikara, "Publish-Subscribe Communication for CoAP". http://kth.diva-portal.org/smash/get/diva2:1111621/FULLTEXT01.pdf

65 Masahito Hasegawa, "Decomposing Typed Lambda Calculus into a Couple of Categorical Programming Languages". Proceedings of the 6th International Conference on Category Theory and Computer Science, 1995. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.53.715&rep=rep1&type=pdf

66 Daniel Hershcovich, *et. al.*, "Universal Dependency Parsing with a General Transition-Based DAG". http://www.cs.huji.ac.il/~oabend/papers/daniel_ud_parser.pdf

67 Kenneth Holmqvist, "Dimensions of Cognition", in Jens Allwood and Peter Gärdenfors, eds., *Cognitive Semantics*, pp 153 - 171, Amsterdam, Philadelphia: John Benjamins, 1999. https://www.lucs.lu.se/spinning/categories/cognitive/Holmqvist/kenneth.pdf

68 Gary B. Huang, *et. al.*, "Towards Unconstrained Face Recognition". http://vis-www.cs.umass.edu/papers/unconstrained_face_workshop.pdf

69 Idris Development Wiki. "The Effects Tutorial". http://docs.idris-lang.org/en/latest/effects/index.html

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

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

72 Patchaiah Kalaiselvi and Sivasamy Nithya, "Face Recognition System under Varying Lighting Conditions". In *IOSR Journal of Computer Engineering*, Volume 14, Issue 3 (Sep.-Oct. 2013), pages 79-88. http://www.iosrjournals.org/iosr-jce/papers/Vol14-issue3/M01437988.pdf

73 Aslı Kale and Selim Aksoy, "Segmentation of Cervical Cell Images". http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.619.3398&rep=rep1&type=pdf

74 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

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

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

77 James Knight, *et. al.*, "Uses of Accelerometer Data Collected From a Wearable System". http://pure-oai.bham.ac.uk/ws/files/4856321/PUC_Accel.pdf

78 Lingpeng Kong, Alexander M. Rush, and Noah A. Smith, "Transforming Dependencies into Phrase Structures". http://www.aclweb.org/anthology/H01-1014

79 Hyunwoo Lee, *et. al.*, "An Enhanced Method to Estimate Heart Rate from Seismocardiography via Ensemble Averaging of Body Movements at Six Degrees of Freedom". https://www.ncbi.nlm.nih.gov/pubmed/29342958

80 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

81 Haishan Liu, *et. al.*, "Mining Biomedical Data using RDF Hypergraphs". 12th International Conference on Machine Learning

and Applications, 2013. http://ix.cs.uoregon.edu/~dou/research/papers/icmla13_hypergraph.pdf

82 Feng Lu, *et. al.*, "Adaptive Linear Regression for Appearance-Based Gaze Estimation". http://phi-ai.org/publications/papers/Adaptive%20Linear%20Regression%20for%20Appearance-Based%20Gaze%20Estimation.pdf

83 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

84 Haney Maxwell, "Persistent Homology of Finite Topological Spaces". http://www.math.uchicago.edu/~may/VIGRE/VIGRE2010/REUPapers/Maxwell.pdf

85 Katharina Mehner, *et. al.*, "Analysis of Aspect-Oriented Model Weaving". http://www.mathematik.uni-marburg.de/~swt/Publikationen_Taentzer/MMT09.pdf

86 Mark Minas and Hans J Schneider, "Graph Transformation by Computational Category Theory". https://www2.informatik.uni-erlangen.de/staff/schneider/gtbook/fmn-final.pdf

87 Gilad Mishne and Maarten de Rijke, "Source Code Retrieval using Conceptual Similarity". https://staff.fnwi.uva.nl/m.derijke/Publications/Files/riao2004.pdf

88 Bálint Molnár, "Applications of Hypergraphs In informatics: A survey and opportunities for research" http://ac.inf.elte.hu/Vol_042_2014/261_42.pdf

89 Amir More, "CoNLL-UL: Universal Morphological Lattices for Universal Dependency Parsing" http://coltekin.net/cagri/papers/more2018.pdf

90 Erwan Moreau, "From link grammars to categorial grammars" https://hal.archives-ouvertes.fr/hal-00487053/document

91 Joakim Nivre, "Dependency Grammar and Dependency Parsing." http://stp.lingfil.uu.se/~nivre/docs/05133.pdf

92 Timothy Osborne and Daniel Maxwell, "A Historical Overview of the Status of Function Words in Dependency Grammar" https://www.aclweb.org/anthology/W15-2127

93 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

94 Heiko Paulheim and Christian Bizer, "Type Inference in Noisy RDF Data". http://www.heikopaulheim.com/docs/iswc2013.pdf

95 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

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

97 John Petitot and Barry Smith, "New Foundations for Qualitative Physics". In *Evolving Knowledge in Natural Science and Artificial Intelligence*, J. E. Tiles, G. T. McKee and C. G. Dean, eds., London: Pitman Publishing, 1990, pages 231-249. http://ontology.buffalo.edu/smith/articles/qualitative_physics.pdf

98 Alexandra Poulovassilis and Mark Levene, "A Nested-Graph Model for the Representation and Manipulation of Complex Objects". Data and. Knowledge Engineering, 6, 3 (1991), pp. 205-224 http://www.dcs.bbk.ac.uk/~mark/download/tois.pdf

99 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

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

101 Siva Reddy, *et. al.*, "Transforming Dependency Structures to Logical Forms for Semantic Parsing". https://aclweb.org/anthology/Q16-1010

102 Alejandro Rodriguez, *et. al.*, "On Modelling and Validation of the MQTT IoT Protocol for M2M Communication" http://ceur-ws.org/Vol-2138/paper5.pdf

103 Justin Salamon, *et. al.*, "Towards the Automatic Classification of Avian Flight Calls for Bioacoustic Monitoring". PLOS ONE, November 2016, pages 1-26. http://www.justinsalamon.com/uploads/4/3/9/4/4394963/salamon_flightcalls_plosone_2016.pdf

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

105 Aviv Segev and Avigdor Gal, "Putting things in context: a topological approach to mapping contexts and ontologies" https://www.aaai.org/Papers/Workshops/2005/WS-05-01/WS05-01-003.pdf

106 Barry Smith and Anand Kumar, "The Ontology of Blood Pressure: A Case Study in Creating Ontological Partitions in Biomedicine". https://www.researchgate.net/publication/228961604_The_Ontology_of_Blood_Pressure_A_Case_Study_in_Creating_Ontological_Partitions_in_Biomedicine

107 Mohamed Soltane, *et. al.*, "Face and Speech Based Multi-Modal Biometric Authentication". https://www.researchgate.net/publication/228463467_Face_and_Speech_Based_Multi-Modal_Biometric_Authentication

108 John G. Stell, "Granulation for Graphs". International Journal of Signs and Semiotic Systems, 2(1), 32-71, January-June 2012. https://pdfs.semanticscholar.org/9e0f/a93a899e36dc3df62feabc004a0ecef4365d.pdf

109 John G. Stell, "Formal Concepts Analysis over Graphs and Hypergraphs". In *Graph Structures for Knowledge Representation and Reasoning*, Croitoru, M, Rudolph, S, Woltran, S and Gonzales, C, eds., Springer, 2013, pages 165-179. http://eprints.whiterose.ac.uk/78795/7/GKRLNCS_with_coversheet.pdf

110 Harry Strange, *et. al.* "Modeling Mammographic Microcalcification Clusters using Persistent Mereotopology". https://www.sciencedirect.com/science/article/pii/S0167865514001263

111 Gregor Strle, “Semantics Within: The Representation of Meaning Through Conceptual Spaces”. Univ. of Novi Gorici, dissertation, 2012.

112 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

113 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

114 Matúş Tejiščák and Edwin Brady, “Practical Erasure in Dependently Typed Languages”. https://eb.host.cs.st-andrews.ac.uk/drafts/dtp-erasure-draft.pdf

115 Ashish Patro and Suman Banerjee “COAP: A Software-Defined Approach for Managing Residential Wireless Gateways”. https://research.cs.wisc.edu/wings/projects/coap/papers/coap_spec.pdf

116 Pietro Ramellini, “Boundary Questions Between Ontology and Biology”.In *Theory and Applications of Ontology: Philosophical Perspectives*, R. Poli and J. Seibt, eds., Springer, 2010, pages 1039-1058. http://mirror.thelifeofkenneth.com/lib/electronics_archive/Theory_and_Applications_of_Ontology_Philosophical_Perspectives.pdf

117 Milan Straka, *et. al.*, “UDPipe: Trainable Pipeline for Processing CoNLL-U Files Performing Tokenization, Morphological Analysis, POS Tagging and Parsing” http://www.lrec-conf.org/proceedings/lrec2016/pdf/873_Paper.pdf

118 Alexandru Telea, “Visualisation and Simulation with Object-Oriented Networks” Dissertation, Eindhoven, 1999. http://papers.cumincad.org/data/works/att/83cb.content.pdf

119 Alexandru Telea and Jarke J. van Wijk, “VISSION: An Object Oriented Dataflow System for Simulation and Visualization” https://www.rug.nl/research/portal/files/3178139/1999ProcVisSymTelea.pdf

120 Bhavani Thuraisingham, “Security Standards for the Semantic Web”. https://pdfs.semanticscholar.org/f49c/6558265fcbfb0cbb3221af089d5deb06aa35.pdf

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

122 J. V. Tucker and J. I. Zucker, “Computation by ‘While’ Programs on Topological Partial Algebras ”. Theoretical Computer Science 219 (1999), pp. 379-420. https://core.ac.uk/download/pdf/82201923.pdf

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

124 G. R. Wetherington, Jr., *et. al*, “Two-Year Operational Evaluation Of A Consumer Electronics-Based Data Acquisition System For Equipment Monitoring ”. https://www.osti.gov/servlets/purl/1393863

125 Yurick Wilks, “The Semantic Web as the Apotheosis of Annotation, but What Are Its Semantics?”. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.131.4958&rep=rep1&type=pdf

126 Mark D. Wilkinson, *et. al.*, “The FAIR Guiding Principles for Scientific Data Management and Stewardship”. http://nrs.harvard.edu/urn-3:HUL.InstRepos:26860037

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

128 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

129 Fei Xia and Martha Palmer, “Converting Dependency Structures to Phrase Structures”. http://www.aclweb.org/anthology/H01-1014

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

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

132 Charles Zhang and Hans-Arno Jacobsen, “Refactoring Middleware with Aspects”. IEEE Transactions on Parallel and Distributed Systems Vol. 14 No. 12, November 2003. https://pdfs.semanticscholar.org/0304/5c5cc518c7d44c3f7b117ea11dfae4932a89.pdf?_ga=2.207853466.903112516.1533046888-196394048.1525384494

133 Li Zhu, *et. al.*, “Development of a High-Sensitivity Wireless Accelerometer for Structural Health Monitoring” https://www.ncbi.nlm.nih.gov/pubmed/29342102