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

Nathaniel Christen[1]

[1]Linguistic Technology Systems, Inc.

September 5, 2019

Abstract

This chapter will explore the integration of several methodologies related to source code analysis and software Requirements Engineering. The chapter will review 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 new representational device described here involves a theory of "channels" which permits graph-based code models to be integrated with type theories and lambda calculi structured around modern programming paradigms. The proposed techniques support documentation and verification of procedural, data type, and holistic specifications — implementational assumptions on procedures and/or modeling assumptions on types, along with larger-scale inter-type relationships. This chapter will use real-world CyberPhysical case studies to illustrate data models which call for advanced code-documentation techniques, insofar as CyberPhysical software should prioritize safety and reliability. 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 study CyberPhysical systems at the level of individual devices. Each device has its own mechanical properties, generates its own kind of data, and may require its own software to interpret and understand that data. As devices proliferate, so does the diversity of software via which users may access whatever data they generate.

CyberPhysical systems may 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 and many different vendors. Such an 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.

I will use the term *hub application* to describe software meeting the requirements of what Teixera *et. al.* call an "application capable of handling a dynamic set of sensors and devices".

Hub applications must receive data from many kinds of devices. They must also respond properly to data once it is received. Each kind of device should have a corresponding software component built around data generated by *that particular* device. For sake of discussion, I will call such device-specific software components a *hub library*. 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. 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 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 provide a concrete artifact that engineers may consult to obtain information about device properties and expected behavior. Compared to (as Teixera *et. al.* put it) "a set of isolated interactive artifacts", I believe hub applications and hub libraries, with a centralized architecture, are more conducive to rigorous, concrete representation of CyberPhysical devices as technical products. This rigor can make CyberPhysical systems more secure and trustworthy — hub applications may be used for testing and prototyping devices and their supporting code, even before they 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, 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.

I claim, also, that hub libraries are operationally similar to a different genre of software components: code libraries providing access to scientific data sets. In practice, most CyberPhysical devices are products of Research and Development cycles that emanate from scientific and technological advances. Research data generated during an R&D phase may therefore be an originating source for data models which ultimately govern the deployed CyberPhysical software. As a result, code libraries which systematically access research data may be seen as ancestral versions of hub libraries — even though hub libraries prototypically work with real-time input, whereas data sets are curated information spaces that are frozen and time, and potentially reused in multiple research projects.

In essence, libraries for accessing research data sets — which may be called "dataset applications" — are analogous to hub libraries whose real-time networking logic is subtracted out; their input data comes from static files, not from any kind of decentralized or wireless networks. Hub libraries actually have two roles: they are low-level drivers attuned to network signals, and also high-level processors transforming raw data into analyzable and visualizable representations. Hub libraries can inherit the logic for this second, high-level role from R&D data-set libraries. This means that published data sets, and their accompanying code, are an important foundation for establishing data models and coding practices that may propagate through CyberPhysical systems' subsequent deployment phases.

So, although I claimed to write this chapter from the perspective of a programming writing a hub library, it is more literally accurate to say that I am writing from the perspective of a programmer composing dataset applications — because that *is* my perspective in real life. I contend that dataset applications are a reasonable proxy for hub libraries, so that these perspectives will coincide for many practical purposes.

I will orient this chapter's discussion toward the C++ programming language, which is arguably the most central point from which to consider the integration of concerns — GUI, device networking, analytics — characteristic of CyberPhysical hub software. C++ is unique in having extensive resources traversing various programming domains, like native GUI components alongside low-level networking and

logically rigorous data verification. For this reason C++ is a reasonable default language for examining how these various concerns interoperate.

The demo code for this chapter (itself composed mostly in C++) includes several re-published data sets in various technical and CyberPhysical domains (bioacoustics, speech samples, and parsed language samples) together with "dataset applications" to access research data and model its properties. I hope these demonstrations serve to illustrate how future CyberPhysical data sets might be organized. I have also supplemented the data sets with code bases operationalizing many of the theoretical paradigms I present in the second half of this chapter. For example, the code provides a modest but demonstrative scripting platform via a hypergraph-based Intermediate Representation based on (what I call) Channel Algebra. A few code samples are drawn from the demo and published here to illustrate some basic patterns in these hypergraph structures; interested readers may find that work discussed in much greater detail in the documentation for the demo code.

The first two sections in this chapter, however, are less theoretical and less code-oriented. These sections discuss hub applications and CyberPhysical systems on a more practical level: what are representative examples of data structures and coding requirements that hub libraries will need to encapsulate? I will then, in the final several 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 difficulties 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 many different kinds of devices. Hub applications also present technical challenges that are different from other kinds of software, 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, and where data is usually stored on remote servers) and "native applications" (characterized by more complex GUI 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 software 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 a refrigerator "that notifies the user if the door stays accidentally open" and moreover "knows what is inside the refrigerator and [its] expiration dates". Consider then how we would design a User Interface networking with "smart" refrigerators. A simple indicator showing whether doors are open is straightforward, but an interface listing the items inside is much more complicated. Once a refrigerator can detect signals emanating from food items/containers, we can envision a list of items presented as a GUI component, perhaps one food item per line (each line, say, showing a picture, price, text description, and expiration date). This would require cross-referencing numeric codes, which might be broadcast by the items *inside* the refrigerator, against a database that would load the images, descriptions, dates, and prices.

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.

On the other hand, a phone app interfacing with that same data would be constrained by its lesser screen real estate and limited interactive modalities. For example, with no room on-screen to show a complete list of items — and no obvious 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 (smaller screen, brief but frequent user engagement), designers have to offer compact but curated snippets of information. That is, software is forced to anticipate what info carries the most user interest — expiry dates are probably most 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.

Conversely, if we assume that hub applications will adopt the "look and feel" of native desktop applications, then they can present more holistic information — taking advantage of larger screens, with secondary application windows and other interactive features that we associate with native GUI components. It is a reasonable hypothesis that this renders AI less important: the more data that can be shown, the less need for software to filter information on users' behalf. 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". In that spirit, investments in AI solutions might be redirected to HCI, securitization, data transparency, and other software virtues which customers may value more than "smart" software that actually strips them of control.

Hub applications could therefore exemplify 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 people 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 data in the different contexts of GUI components, database storage, and analytic review, to confirm proper operation of devices and their handler code.

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 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], [41], [72], [116], [117], [124]). 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 [85] and the Fukushima nuclear reactor meltdown

[134]. 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 terms 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 have to 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 licensed, 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 scrutinizing medical practices; commercial activities (like marketing) are regulated, and operational transparency (like reporting adverse outcomes) is mandated, more so than in most other sectors of the economy. This level of oversight *causes* the public to trust that clinicians' recommendations are usually correct, or that medicines are usually beneficial more than harmful.

The problem, as software becomes an increasingly central feature of the biomedical ecosystem, is that no commensurate oversight framework exists in the software world. Biomedical IT regulations tend to be ad-hoc and narrowly domain-focused. For example, code bases in the United

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

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. 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 identify coding standards which are believed, on firm theoretical grounds, 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 $\mathfrak{t}$. This type can then be associated with a type (say, $\mathfrak{t}'$) of GUI components which visually display values of type $\mathfrak{t}$. 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, $\mathfrak{t}''$) representing $\mathfrak{t}$-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/or 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 implementations for receiving new data from external sources or signals. These are places where data "enters the system".[2] Conversely, other code points localize the software's capabilities to initiate

[2] A simple example is, for desktop applications, the preliminary code which runs

external effects.[3] The functions which leverage such capabilities reveal data "leaving the system". Distinguishing where data "enters" and "leaves" the system from where data is transferred *inside* the application helps ensure that incoming data and external effects are properly vetted.[4]

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.

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.

[3] 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.

[4] Several mathematical frameworks have been developed to 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. [71], [98], [99]) and the theory of Hypergraph Categories ([31], [51], [52], [77]).

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 were inverted, the difference would be a negative number, which (under binary conversion to an unsigned integer) would come out as a potentially extremely large positive number (as if the patient had blood pressure in, say, the tens-of-thousands). This nonsensical outcome indicates that the basic calculation is fragile. We then have three options: test "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).

Also, engineers may appreciate the flexibility to upgrade how improper data is handled *throughout the application*. Suppose it is decided to log, and periodically review, all instances of malformed parameters "rejected" by gatekeeper code. Every **if…then…else** block could potentially then need to be paired with logging code, and it would be time-consuming and error-prone to verify that such protocol is obeyed everywhere, throughout a large, complex code base. Isolating gatekeeper procedures, and labeling them as such, would make it both easier to find all gatekeeping logic and to modify the protocol for handling failed validations.

In essence, factoring *gatekeeper* and *fragile* code into separate procedures exemplifies the programming maxim of "separation of concerns": it makes the overall system more flexible and easier to maintain. However, such separation creates a new problem: 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.

Because there is no one best "gatekeeping protocol", these issues should be studied holistically, defining a range of options that can be weighed at the planning and prototyping stages — well before most of the serious production code is implemented.

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 assuming proper functioning (and therefore what validations can check that the instrument is working as intended). For actuators, it should be similarly understood what range of values guarantee safe, correct behavior. For any device then we can construct a *profile* — an abstract, mathematical picture of the space of "normal" values associated with proper device performance. Gatekeeping code can then ensure that data received from or sent to devices fits within the profile. Defining device profiles, and explicitly notating the corresponding gatekeeping code, should therefore be an essential pre-implementation planning step for CyberPhysical software hubs.

1.2 Fragile Code

Fragile code is code that makes assumptions stronger than the programming language on its own can guarantee. Where safety and quality is a priority, fragile code needs gatekeeping code to ensure that these assumptions are warranted.

Fragile code is not necessarily a harbinger 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 that the *totality* is robust. Defining and declaring specifications is then an intrinsic part of implementing code bases which are both robust and maintainable.

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

Most programming language will provide some basic (typically relatively coarse-grained) specification semantics, usually through type systems and straightforward code observations (like compiler warnings about unused or uninitialized variables). For sake of discussion, assume that all languages have distinct compile-time and run-time stages (though these may be opaque to the codewriter). We can therefore distinguish compile-time tests/errors from run-time tests and errors/exceptions. This permits us to formulate questions such as: 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 principle 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 separated out from the main (fragile) procedure. 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 — i.e., to compilers and runtimes). 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.

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.

These limitations then produce a split between tools *in the language itself* and those maintained as separate projects *analyzing* code in a given language. They raise the question, which has no simple answer, of what should be guaranteed *by the language* and what should be tested externally. I will now examine this question in greater detail.

1.3 Core Language vs. External Tools

Because of programming languages' engineering limitations, such as I just outlined, software projects should not necessarily rely on core-language features for responsible, safety-conscious programming. 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. [44, 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. 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 [96] and [90], [125]:

> 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 prin-

> ciple, 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. [96, 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 their 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. I then 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.

Prior to launching into that mostly theoretical discussion, however, I will examine real-world CyberPhysical data with a little more specificity. My goal in the following pages is to describe the sorts of details that need to be expressed in CyberPhysical data models and therefore verified in CyberPhysical code. Whereas my subsequent analyses of code representation will address *how* code can demonstrate its underlying data model, my preliminary discussion will motivate *why* code needs to do so in the first place.

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], [17], [79], [81], 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. [128], [137]).

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., [48]), 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 [74], 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. [42], [86], [114]). 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 Alluri and Vuppala, this volume; and 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. The two chapters just mentioned present methodology for estimating speakers' emotional states and identifying samples' spoken language or dialect, respectively; while the chapter by Teixera *et. al.* (which I quoted earlier) discusses speech-activated User Interfaces.

The data profile germane to an audio processor will be determined by the system's overarching goals. For example, [30] 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 software 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 (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 [106] (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, which thereby themselves serve a classifying role, semantically linking each 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 ([26], [40], [68], [73], [84], 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 CyberPhysical 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 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. 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, 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, [111] (one of whose co-authors 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 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 — rather than just networking with their own proprietary access points, like phone apps.

In this eventuality, 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. Devices must be designed to broadcast data to third-party software platforms, and software hubs must 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 an intermediate semantic layer acting between "smart objects" and corresponding hub applications.

The data models at this semantic midlayer, moreover, would no longer be device-centric. Instead, they would 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? This means that common representational formats for wireless data, or standardized IoT Ontologies — however much these may technically embody semantic middleware — are insufficient as *high-level*, *software-centric* semantic layers.

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], [37], [56], [62], [64], [105], etc.) than there are open-source libraries which prototype device data, its validation, parameters, and proper transformations.[5] A good case-in-point can be found

[5] 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 — compo-

in [119, 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 with technical and legal specifications.

In practical terms, the above discussion mentioned the CLO-43SD (avian) data set and then the chapter by Vuddagiri *et. al.*, 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-centric" — 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], [22], [49], [130]).

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

nents 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 conformant 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 products. 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.

This volume's chapter by Teixera *et. al.* considers voice-activated CyberPhysical interfaces; here 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.[6] 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). 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. [27], [61], [66], [92], [121]). 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.[7] 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, 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

[6] 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], [80], [94], [95], [104], [107], [133], etc.; variants include Link Grammar [58], [93] and Extensible Dependency Grammar [35], [36], [55].

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

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. (Those extra details 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.

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; in particular, 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 performance anomalies which look like software mistakes even if the code is entirely correct (see [45] and [102], 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, pa-

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 color 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

rameters 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 profiles, 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 argued 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 in this chapter, 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 for such a comprehensive theory. One perspective 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, third, procedures can be studied via mathematical frameworks such as Lambda Calculus, allowing 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 — relatively simple geometric expressions — mathematicians began to ask foundational 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.[8]

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 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., [46], [65], [126]).

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

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

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], [83], [89]; [91], [101], [112], [113]). 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 **DH**s 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 **DH**s 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).[9]

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 **DH**s which (for reasons to emerge) I will call "Channelizable". Channelizable Hypergraphs (**CH**s) have these properties:

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

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

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

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

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

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

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

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

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

10. Each edge, separate and apart from the **CH**'s actual graph structure, is associated with a distinct hypernode, called its *annotation*. This annotation cannot (except via a proxy) be associated with any other hypernode (it cannot be a head- or tail-set in any hypernode). The first hyponode in

[9] 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 **DH**s meet this condition. Any graphs discussed here are assumed to meet this condition.

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 ⟨⌈**Nathaniel**⌉, ⌈**46**⌉⟩, connected to a hypernode with hyponodes ⟨⌈**Brooklyn**⌉, ⌈**Democrat**⌉⟩, 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 ⌈**Nathaniel**⌉ and it acquires 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.

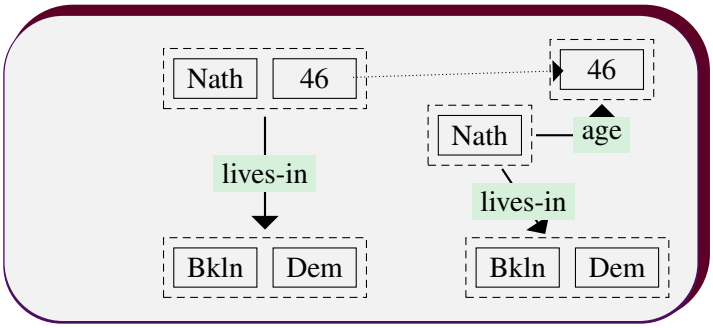

Diagram 1: Unplugging a Node.

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 inter-

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;
...}}}
```

nal 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 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.

3.3 Channelized Hypergraphs and RDF

The Resource Description Framework (RDF) models information via directed graphs ([32], [33], and [63] 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" [19, p. 6]. That is, the specification for RDF list-type data structures invites us to consider that they *may* be regarded integral units

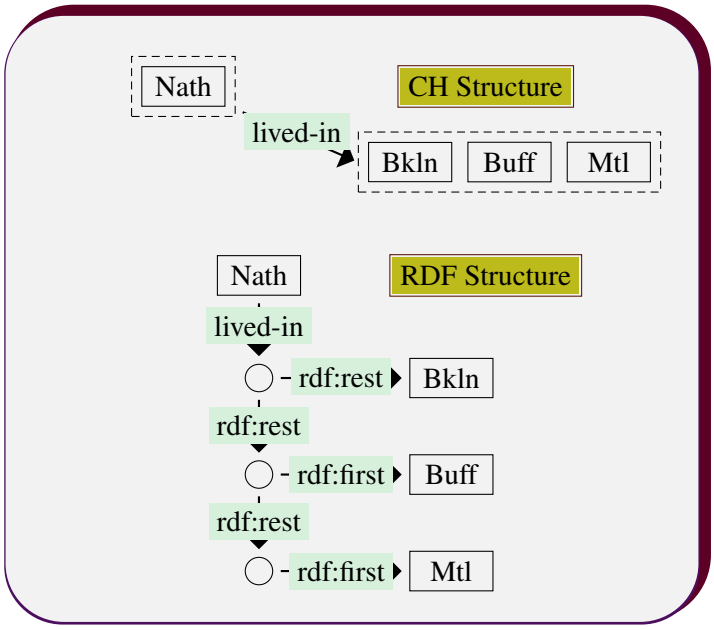

Diagram 2: CH vs. RDF Collections.

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 [129]). 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], [20], [21], [38], [47], [100], [108], [110], or [120]). 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 do-

mains 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 **struct**s — which have no *a priori* uniqueness mechanism).

Also, RDF types are less intrinsic to RDF semantics than in CH (see [97]). The foundational elements of CH are value-tuples (via nodes expressing values, whose tuples in turn are hypernodes). Tuples are indexed by position, not by labels: the tuple $\langle \lceil \textbf{Nathaniel} \rceil, \lceil \textbf{46} \rceil \rangle$ 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 forming "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).[10] 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).

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

In 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 [24]. 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 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 $\not\Delta$**LAMBDA**$.xy$.

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 **sigma** model a "Sigma" channel, as in "Sigma Calculus" (written as ς-calculus: see e.g. [2], [25], [50], [135], etc.). We then have the requirement than any procedure's **sigma** 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 "**sigma**" 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' **sigma** channel. The pairing of **return** and **sigma** 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 those 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 type Categories that are Cartesian Closed). 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. Channel formulae and signatures describe norms source-code norms which could also be expressed via 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 [75], [78], [82], [127], [131], [132]). Given the relations between **RDF** and Directed Hypergraphs (despite differences I have discussed here), Channel Algebras can also be seen as adding to Ontologies governing Directed Hypergraphs. Such is the perspective I will take for the remainder of this chapter.

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

Similarly, we can then develop for Channel Algebras a *Channel Expression Language*, or **CXL**, which can indeed

be integrated with appropriate TXLs. 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 §5 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.

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, sometimes functional values are denoted via inter-function operators (like the composition $f \circ g$) rather than by giving an explicit implementation. We can say that functions defined via operators (like $\circ$) 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". On the other hand, code-graphs capture the implementations through which function-like types are (mostly) populated with concrete values.

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. Moreover, channels can be seen as structures on *graphs*, as well as runtime information flows, so that channels are applicable for both static and dynamic program analysis.

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 concerning Requirements Engineering via code models.

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-like values is to define operators on function-like types themselves, which transform function-like values to other function-like 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-like values via code-graphs. 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-like values are internally represented. We *might* treat all initializations of these values as via code-graphs (in practice, compiled down via an Abstract Syntax Tree or graph to some Intermediate Representation or byte-code). 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. Alternatively, 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).

4.1.1 Addressability and Implementation

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-like 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 produce multiple addressable values. 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. So there is no single, universal "Type Expression Language". One role of any given TXL is to model what its corresponding language recognizes as a type, or — better — a *potential* type. A TXL expression which designates a (unique) type is well-formed if it unambiguously describes a type that *could* exist. Such an expression does not, however, implement the type on its own, or mandate that the type be implemented; it would merely affirm that the type so designated is implementable within the target language.

As a concrete example, consider a type described in English as: "the type inhabited by functions which take, as one parameter, a Unicode string, and, as the second parameter, an unsigned integer less than the length of the string". A TXL version of this specification would only be valid if the requirements thereby described can be satisfied, in the target language, via type-checking alone.

For a more in-depth 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> (assume this will be implemented as a list of integers). 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>, then the compiler can derive a "specialization" of the template for a specific T — or the programmer can specialize MyList on int (or any other 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.

Templates and specialization add complexity to discussions about types, because compilers may automatically instantiate concrete types from templated code *unless* programmers supply specializations which deviate from the template. As a result, in a local segment of a source file it may be impossible to know whether or not the code concretizing a templated type is automatically generated from a template. Another complication is that compilers may derive *default implementations* of types' constructors, unless these are coded explicitly. Taking these two considerations together, it can be difficult in a code base to, given a type, find which code-segments correspond to that types' constructors.

As an analytic device, here I assume that every implementable type can be associated with a procedure I will call a *co-constructor*, whose role is to wrap constructor-calls in a readily identifiable code body. Co-constructors are "ordinary" procedures in the sense that they are "addressable". Specifically, addressable procedures have these properties:

1. You can take their address (assuming we are dealing with a language that supports function pointers in the first place).
2. They have a corresponding (possibly templated) location in source code (and therefore a code-graph). For co-constructors, this location can be marked as such — it should be straightforward to identify all co-constructor implementations in a code base.
3. They can be exposed to scripting engines and runtime reflection; so co-constructors enable type-instances to be created via scripts and other runtime-introspection capabilities.

Operationally, co-constructors are similar to *factory procedures* or *object factories* (see e.g. [29, esp. pages 32-35], [34, esp. pages 35-36], [70], [87]), which similarly delegate to constructors but can be used in contexts where constructors cannot, e.g. where it is necessary to address the factory through a pointer (note that in C++ you may not take the address of an actual constructor).

Insofar as co-constructors are *addressable*, they provide an indirect mechanism for designating their corresponding type. I will use the term *preconstructor* to mean a function-pointer holding the address of a co-constructor, or some similar data structure which uniquely identifies a co-constructor. A preconstructor thereby holds a compact value which is associated with exactly one type. A valid preconstructor, in particular, serves as proof that a given type is

implemented — it confirms the existence of at least one fully implemented constructor for that type, indicating that the type is *actual* and not just *potential*.

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).

However, merely describing range-delimited types' desired space of values does not provide a full implementation specification. What should happen if someone tries to construct an **int**⟮**0,100**⟯ value with the number, say, **101**? What about with values taken from an external source, like a web **API**, where it cannot be formally proven that the values fall in the proper range? These question point to implementation choices that transcend formal designations. This is why **TXL** expressions should be seen as just articulating *potential* types, because bringing types into actuality will usually call for engineering choices that transcend type theory *per se*. Once types *are* implemented, co-constructors serve as tangible witness to types' actualization, and preconstructors are convenient proxies referring to those types.[11]

Reasoning abstractly about functions and types needs to be differentiated from reasoning about available, implemented types (and functions defined on them). 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?[12] 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).[13] 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).

Since co-constructors are *addressable*, they cannot — at least not within the framework I have discussed thus far — be "temporary" function-values analogous to $f \circ g$. This means that *types* cannot be temporary values. More precisely, a type system may be constrained by the proposition that *no type can be created* whose co-constructors would have to be temporary values — or, to put it differently, no type can be created whose co-constructors are not procedures that can be mapped to source-code segments (and thereby to code-graphs).

Notice that co-constructors then are not just function-like values; co-constructors have to be in that subspace of function-like values initialized via code-graphs, rather than via some quasi-arithmetic inter-function operator like $f \circ g$. This then limits what we can do with Dependent Types, type-state, and other "expressive" type mechanisms. I will call this

[11] Similar issues are sometimes addressed by a *modal* type theory (cf., e.g., [53]) 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").

[12] 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**⟩).

[13] 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).

the "metaconstructor" problem: insofar as co-constructors are function-like values, they (in principle) need their own constructors — call these "metaconstructors". We can stipulate that metaconstructors — constructors of co-constructors — have to be derived from code graphs (they cannot be temporary values), but this renders certain advanced type-theoretic features inaccessible to our applied type systems. Conversely, we can accept the idea of constructors being (potentially) temporary values, but this interferes with the idea of preconstructors being referential proxies for types themselves (unless types also are, potentially, temporary constructs, which creates a new set of problems). I will now explain this choice in greater depth.

4.2 Dependent Types and Co-Constructors

To see why the metaconstructor problem determines how extensively Dependent Types are supported in a type system, consider a variation on the range ⟨**0,100**⟩ type. In lieu of a fixed range, consider a procedure taking a (variable) **T**-range ⟨**r**⟩ and a number **x** *which must be in that range*. Here **x** "depends" on ⟨**r**⟩ — its *type* is ⟨**r**⟩ itself seen as its own **T**⟨V_1,V_2⟩ type — so **x** can vary among many range-types, only being fixed at runtime. Defining a *type* for procedures meeting those *specifications* is a classic problem of Dependent Type theory.

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. Unless we know *a priori* that only a specific set of ⟨**r**⟩s in the first parameter will ever be encountered, the compiler has to be prepared for **x** being assigned any one of many different range types, depending on the **f**'s first argument. In particular, the compiler cannot know ahead of time which constructor to call for **x**. More precisely, it is impossible for the compiler to have *separate* constructors for millions of possible range types. Instead, the compiler must either "create" a constructor "on the fly" or else have some generic constructor which services many range-types, but then requires extra information to establish *which* range is desired.

Assuming we use co-constructors to wrap constructors, these two options for compiler writers correspond to the choice of *either* creating ad-hoc co-constructors *or* designing co-constructors as a compound data structure. 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 range's closest point. 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.[14] To treat such a function as a *real* value constructor we would have to add contextual modifiers: **Cf** is a value constructor for range-type ⟨**r**⟩ in the presence of numbers that specify ⟨**r**⟩ at runtime. The co-constructor for a range type **T**⟨**r**⟩ is accordingly the "common functionality" base function *plus* numbers passed to it — some sort of ⟨**&Cf, r**$_1$**, r**$_2$⟩ compound data structure, again by analogy to **inc** and ⟨**&add, 1**⟩ (see footnote 12, above). Here again though the co-constructor is a temporary data structure, created on-the-fly to model the desired value constructor for an **x** whose type (and therefore whose constructor) is not known until runtime. I contend, on examples like these, that Dependant Typing for a type system $\mathbb{T}$ is thus logically equivalent to the possibility of $\mathbb{T}$ co-constructors being temporary values.

But value constructors (and by extension co-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. Many constructors are called behind-the-scenes: in C++, the standard function-call mechanism is "pass by value", wherein values are *copied* when passed between procedures; but any copy can potentially invoke a so-called "copy constructor". Indeed, programmers use certain constructors as "hooks" to silently insert logic into normal program flow (usually this is to make complex types behave like built-in-types from client code's point of view). Allowing large type families (like one type for each **int** or each two-number range ⟨**r**⟩ — similar to "inductive typing" as discussed by Edwin Brady in the context of the Idris language [23, p. 14]) — could easily conflict with user-defined constructor overrides: users (meaning, in this context, library developers) would need not only to write their own (e.g., copy) constructors, but to hook into a complex run-time mechanism for creating constructors ad-hoc as temporary values. Conversely, forcing value co-constructors to be addressable prohibits "large" type families — like types indexed over other (non-enumerative) types (see e.g. [18, p. 4]) — at least as *actual* types. 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, per-

[14] Here I say "value constructor" to clarify that I am not commenting on *type constructors*, which derived specialized types from generic ones.

haps, a description of the context can be packaged and carried around with the value, sharing the value's lifetime.

As I will now review, this analysis in the realm of Dependent Types carries over into *typestate*, which is another mechanism intended to model coding requirements via type-checkable specifications.

4.2.1 Dependent Types and Typestate

Typestates are finer-grained classifications than types. 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., *type-states* 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 getting 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 value "holders" which share the same symbol-name (and maybe the same type) but differ in state (some holding values unrelated to actual program state). Both options upend normal programming expectations. This situation can be juxtaposed with the "metaconstructor problem", i.e., how Dependent Types force a rethink on basic value-constructor 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 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 indeed 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 (or, as I will discuss later, using a "passkey" to vouch that a callback's file argument *can* be thus cast).

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 [118, 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 [28], and [43, 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], [88], [136]). 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.

4.2.2 Simulating Dependent Types with Preconstructors

Because Dependent Types and typestate recognize fine-grained requirements on which values may be passed to which functions, it might seem as if they are a logical continuation of the telos toward granular data modeling. If our goal is to provide the most expressive data models possible within the bounds of computational tractability (I will return to this in the conclusion), we should certainly allow for Dependent Types and any other constructions which logically imply them (essentially, any formulation wherein types or their constructors are ad-hoc temporary values).

However, Dependent Types have the technical consequence of leaving pre-runtime values (or whatever construct we recognize as "holding" or "carrying" values) either *without* types, or with *different* types than they have at runtime.[15] In this case it becomes difficult to query code *outside* runtime, which arguably *subtracts* expressiveness from the framework. In short, we are free to explore some foundation for emulating Dependent Types *without* giving up on static reflection; the resulting system would not necessarily be expressively lesser than a $\mathbb{T}$ with full-fledged Dependent Type support.

The solution I propose to effectuate this compromise involves using preconstructors as witnesses that a given value construction *could* be performed — so that a given value (or values) is/are *logically consistent* with being construed as instance of some (perhaps ad-hoc) type, but are not *literally* assigned to that type. The preconstructor can then be passed in to functions as an extra parameter, which when valid (e.g., when not a null function pointer) vouches for the co-construction it references being permissible. That is, the preconstructor become a "passkey" parameter returned from a gatekeeper procedure and then passed on as evidence of the gatekeeping validation.

As a concrete example, suppose a procedure requires two numbers where the second is greater than the first (the inverse of the systolic-over-diastolic mandate): **f(x, y)** where **x** $<$ **y**. How can we express the **x** $<$ **y** condition within **f**'s signature, assuming the signature can only express semantics pertinent to **f**'s type attribution? On the face of it, we know that the desired "increasing" condition is equivalent to **y** having a type like **range_gt<x>** — a range bounded (only) from below — where this "**x**" is the **x** preceding **y** in **f**'s signature. But using such directly as **y**'s type-attribution means that from the perspective of **f**'s own type-attribution, **y** does not have a single, fixed type; its type varies according to the value of **x**. Here again we encounter a "metaconstructor" problem: in order for the **x** $<$ **y** condition to be modeled *directly* by **y**'s type-attribution we would need the constructor for **y**'s value-constructor to be some operation that produces a temporary function-value — not simply the compilation of a code-graph to an addressable, non-temporary implementation.

These issues go away if, instead of working with a function taking *two* integers, we instead consider a function taking *one* value which is a monotone-increasing pair (something like **int f(mi_pair pr)**). A type like **mi_pair**, based on ordered pairs x, y of **int**s, solves the metaconstructor problem for **y** because **x** and **y** are no longer distinct **f** parameters with distinct value-constructors; they are subsumed into one pair, whose own value-constructor can check the **x** $<$ **y** condition. The *requirements* for the original (two-valued) **f** may then be *described* as **x** and **y** being convertible into a pair **pr** which is an instance of **mi_pair** (so that **x** $<$ **y**). This *description* is not a *type*, but elevating the description to type level can be at least approximated with a signature like **f(int x, int y, mi_pair pr = mi_pair(x, y))**, which when called as **f(x, y)** will silently call the **mi_pair** constructor. This is only approximate because it allows anomalies like **f(x, y, mi_pair(0, 1))**, defeating the purpose — but at least we can approximate a Dependent Type signature with a passkey protocol that is not difficult to enforce via calling conventions (client code should never call the three-argument form directly).

Now, notice that we do not actually need the third argument; we just want to know that the **mi_pair** constructor *would* accept the x, y pair. So we can replace the *actual* **mi_pair** constructor with a *preconstructor* that *could* be used as a factory for **mi_pair** instances if needed, but can *also* serve to certify that a certain set of arguments (here a pair of numbers) meets the logical preconditions which an actual constructor would check off.

For this to work, the **mi_pair** type would need to be implemented with a static procedure that returns a valid preconstructor for valid inputs — plus, assuming the preconstructor/co-constructor pattern I am advocating, a co-constructor whose address would be the basis of the preconstructor value. These are obviously not features of C++ (or any other language I know of) but could readily be an implementational norm for data types used in a safety-conscious project. In effect, consistent use of preconstructors for fine-grained types is one strategy for siting gatekeeping code broadly throughout a code base.

Further discussion of preconstructors is outside the

[15] Notice also that the sense of "difference" in the second alternative is not compatible with polymorphism via subtyping: the runtime type of a parameter is not a *subclass* of the type family over which the former quantifies, unless we modify Dependent Type theory itself to allow quantification only over families that could be collapsed into one sole base class.

scope of this paper, but concrete examples of range-checking via preconstructor passkeys can be found in the demo. Here I'll make the further point that — if we accept a Channel Algebra which expands beyond present programming languages — we can move preconstructor passkeys to a separate channel, thereby approximating Dependent Types more eloquently. Co-constructors may be identified via a dedicated co-constructing channel — **coconstruct** — which signals that a return value is not *any* procedure returning the associated type, but a constructing procedure which is part of the type's interface and helps to demarcate its space of values. A **coconstruct** channel paired with a special **preconstruct** channel, for preconstructor passkeys, provides a metamodel wherein Dependent Types, typestate, and many effect-systems can be reasonably encoded.

This last case also points to how a theory of channels adds semantic expressiveness to code models: we can achieve via descriptions of inter-procedure information flows — including distinguishing distinct roles such as passkeys vs. ordinary parameters, and constructing returns vs. ordinary procedures happening to return a given type — a semantic exactitude that is implementationally harder (from a language engineering perspective) to achieve directly within a type system. Channel Algebras are not limited to channels actually recognized by existing languages — they could be the basis for new languages, and/or new analytic tools isolating patterns in existing code. With this flexibly channels can be lifted into a construct recognized within data and code modeling paradigms — as well as an added structural layer within hypergraphs — in general. These possibilities may become clearer as I present a theory of channels in more detail next section.

5 Channels and Carriers

Suppose one procedure calls a second. From a high level perspective, this has several consequences that can be semantic-graph represented — among others, that the calling procedure depends on an implementation of the callee being available — but at the source code level the key consequence is that a node representing source tokens which designate functional values enters into different semantic relations (modeled by different kinds of edge-annotations) than nodes marking other types of values and literals. Suppose we have an edge-annotation that x is a value passed to $\mathfrak{f}$; this graph is only semantically well-formed if $\mathfrak{f}$'s representatum has functional type (by analogy to the well-formedness criteria of $\lambda x.fx$).

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

This implicitly assumes that symbols "hold" values; to make the notion explicit, I will say that symbols are *carriers* for values. Carriers do not necessarily hold a value at every point in the execution of a program; they may be "preinitialized", and also "retired" (the latter meaning they no longer hold a meaningful value; consider deleted pointers or references to out-of-scope symbols). A carrier may pass through a "career" from preinitialized to initialized, maybe then changing to hold "different" values, and maybe then retired.[16] I assume each carrier is associated with a single type throughout its career, and can only hold values appropriate for its type.[17]

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

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

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

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

5.1 Carrier Transfers

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

Taking this further, we can define a *channel* as a list of carriers which, by inter-carrier transfers, signify (or orchestrate) the passage of data into and out of function bodies.[18] I'll use the notation $\looparrowright$ to represent inter-carrier transfer: let $\mathfrak{C}_1$ and $\mathfrak{C}_2$ be carriers, then $\mathfrak{C}_1 \looparrowright \mathfrak{C}_2$ is a transfer "operator" (note that $\looparrowright$ is non-commutative; the "transfer" happens in a fixed direction), marking the logical moment when a value is moved from code-point to code-point. The $\looparrowright$ is intended to model several scenarios, including "forced coercions" where the associated value is modified. Meanwhile, without further details a "transfer" can be generalized to *channels* in multiple ways. If $\mathfrak{C}_1$ and $\mathfrak{C}_2$ are carriers which belong to two channels (χ_1, χ_2), then $\mathfrak{C}_1 \looparrowright \mathfrak{C}_2$ elevates to a transfer between the channels — but this needs two indices to be concrete: the notation has to specify which carrier in χ_1 transfers to which carrier in χ_2. For example, consider the basic function-composition $f \circ g$: **(f.g)(x) = f(g(x))**. The analogous "transfer" notation would be, say, **g:return**$_1 \looparrowright$**f:lambda**$_1$: here the first carrier in the **return** channel of **g** transfers to the first carrier in the **lambda** channel of **f** (the subscripts indicate the respective positions).

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

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

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

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

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

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

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

Let $\mathfrak{C}_1 \rightarrowtail \mathfrak{C}_2$ be the special case of $\looparrowright$ corresponding

[18] Note that this usage varies somewhat from process calculi, where a channel would correspond roughly to what is here called a single carrier; I assume channels in the general case are composed of multiple carriers.

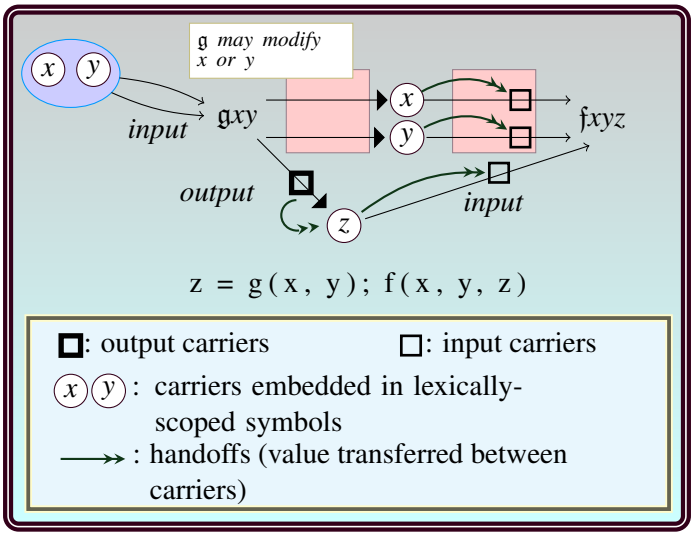

Diagram 3: Visualizing Carrier Transfers ("Handoffs")

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

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

Analysis of value-transfers is particularly significant in the context of Source Code Ontologies and RDF or Directed Hypergraph representations of computer code. This is because code-graphs give us a rigorous foundation for modeling computer programs as sets of function-implementations which call one another. Instead of abstractly talking about "procedures" as conceptual primitives, we can see procedures as embodied in code-graphs (and function-values as constructed from them). Figure 3, for example, shows how graph constructions can track the flow of carriers and values between procedures: this graph-modeling use-case (with additional illustrations) is discussed in greater detail within the demo's documentation. "Passing values between" procedures is then explicitly a family of relationships between nodes (or hypernodes) in disparate code-graphs, and the various semantic nuances associated with some such transfers (type casts, for example) can be directly modeled by edge-annotations. Given these possibilities, I will now explore further how the framework of *carriers* and *channels* fits into a code-graph context.

5.1.1 Channel Groups and Code Graphs

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

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

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

More precisely, the **f(x)** example represents a carrier transfer whose target is part of f_2's **lambda** channel, which we can notate $\mathfrak{c}_1 \overset{\text{lambda}}{\rightarrowtail} \mathfrak{c}_2$. Furthermore, the full statement **y = f(x)** shows a transfer in the opposite direction: the value in f_2's ***return*** channel is transferred to the carrier **y** in the *package*. This relation, involving a return channel, can be

expressed with notation like $\mathfrak{C}_2 \circ\!\!\!\!\rightarrow^{\textbf{return}} \mathfrak{C}_1$. The syntax of a programming language governs how code at a call site supplies values for carrier transfers to and from a function body: in the current example, binding a call-result to a symbol always involves a transfer from a ***return*** channel, whereas handling an exception via code like **catch(Exception e)** transfers a value from a called function's ***exception*** channel. The syntactic difference between code which takes values from **return** and **exception** channels, respectively, helps reinforce the *semantic* difference between exceptions and "ordinary" returns. Similarly, method-call syntax like **obj.f(x)** visually separates the values that get transferred to a "**sigma**" channel (**obj** in this case) from the "ordinary" (**lambda**) inputs, reinforcing Object semantics.

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

A Channel Expression Language (**CXL**) can describe channels both in signatures and at call-sites. The aggregation of channels generically described by **CXL** expressions I am calling a *Channel Group*. A Channel Group representing a function *signature* I am calling a *channel complex*, whereas groups representing a function *call* I am calling a *channel package*. Input channels are then those whose carrier transfers occur in the package-to-complex direction, whereas output channels are the converse.

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

The channel/group/complex/package/carrier vocabulary, I believe, codifies a descriptive framework integrating the *semantic* and *syntactic* dimensions of source code and program execution. Specifically, on the *semantic* side, computer programs can be understood in terms of λ-Calculi combined with models of computation (call-by-value or by-reference, eager and lazy evaluation, and so forth). These semantic analyses focus on how values change and are passed between functions during the course of a running program. From this perspective, source code is analyzed in terms of the semantics of the program it describes: where and when are values moved around?

Conversely, source code can also be approached *syntactically*, as well-formed expressions of a computer language. From this perspective, correct source code is matched against language grammars, and individual code elements (like tokens, code blocks, expressions, and statements) — plus their inter-relationships — are established against this background.

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

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

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

semantics.

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

By intent, Channel Algebras provide a machinery for modeling function-call semantics more complex than "pure" functions which have only one sort of input parameter (as in lambda abstraction) — note that this is unrelated to parameters' *types* — and one sort of (single-value) return. Examples of a more complex paradigm come from Object-Oriented code, where there are two varieties of input parameters ("lambda" and "sigma"); the "sigma" (**this**) carrier is privileged, because its type establishes the class to which function belongs — influencing when the function may be called and how polymorphism is resolved.[19]

Another case-study is offered by exceptions. A function throws an exception instead of returning a value. As a result, **return** and **exception** channels typically evince a semantic requirement (which earlier — see page 30 — I sketched as an algebra stipulation): when functions have both kinds of channels, only one may have an initialized carrier after the function returns. Usually, thrown-exception values can only be bound to carriers in **catch(...)** formations — once held in a carrier they can be used normally, but carriers in **exception** channels themselves can only transfer values to other carriers in narrow circumstances (this in turn depends on delineating code blocks, which will be reviewed below). So **exception** channels are not a sugared form of ordinary returns, any more than objects are sugar for functions' first parameter; there are axiomatic criteria defining possible formations of **exception** and **return** channels and carriers, criteria which are more transparently rendered by recognizing **exception** and **return** as distinct channels of communication available within function bodies.

In general, extensions to λ-Calculus are meaningful because they model semantics other than ordinary lambda abstraction. For example, method-calls (usually) have different syntax than non-method-calls, but ς-calculi aren't trivial extensions or syntactic sugar for **lambda**s; the more significant difference is that sigma-abstracted symbols and types have different consequences for overload resolution and function composition than **lambda**-abstractions. Similarly, exceptions interact with calling code differently than return values. Instead of scattered λ-extensions, Channel Algebra unifies multiple expansions by endowing functions (their signatures, in the abstract, and function-calls, in the concrete) with multiple channels, each of which can be independently modeled by some λ-extension (objects, exceptions, captures, and so forth).

Specific examples of unorthodox λs (objects, exceptions, captures) suggest a general case: relations or operators between procedures can be modeled as relations between their respective channels, subject to channel-specific semantic restrictions. A *method* can be described as a function with several different channels: "**lambda**" with ordinary arguments (as in λ-calculus); "**sigma**" channel with a distinguished **this** carrier (formally studied via "ς-calculus"); and a **return** channel representing the return value. Because the contrast between these channels is first and foremost *semantic* — they have different meanings in the semantics of the programs where they appear — channels may therefore have restrictions governed by programs' semantics. For example, as I mentioned in the context of "method chaining", it may be stipulated that both **sigma** and **return** channels can have at most one carrier; as a result, a special channel-to-channel operator can be defined which is specific to passing values between the carriers of **return** and **sigma** channels. This operator is available because of the intended semantics of the channel system. Each channel kind has its own semantic interpretation, with commensurate axioms and restrictions. Subject to these semantics, carrier-to-carrier operators translate to channel-to-channel operators. A Channel Algebra in this sense is not a single fixed system, but an outline for modeling function-call semantics in the context of different programming languages and environments.

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

5.2.1 Statements, Blocks, and Control Flow

The previous paragraphs discussed expanded channel structures — with, for example, objects and exceptions — that model call semantics more complex than the basic **lambda**+**return** (of classical λ-Calculus). A variation on this theme, in the opposite direction, is to *simplify* call structures: procedures which lack a **return** channel have to com-

[19] Also, as I discussed earlier (page 31), "chaining" method calls means that the result of one method becomes an object that may then receive another method (the following one in the chain). Such chaining allows for an unambiguous function-composition operator.

municate solely through side-effects, whose rigorous analysis demands a "type-and-effect" system. Even further, consider functions with neither **lambda** nor **return** (nor **sigma** nor, maybe, **exception**). As an alternate channel of communication, suppose function bodies are nested in overarching bodies, and can "capture" carriers scoped to the enclosing function. "Capture semantics" specifications in C++ are a useful example, because C++ (unlike most languages that support anonymous or "intra-expression" function-definitions) mandates that symbols are explicitly captured (in a "capture clause"), rather than allowing functions to access surrounding lexically-scoped with no further notation: this helps visualize the idea that captured symbols are a kind of "input channel" analogous to **lambda** and **sigma**.

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

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

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

This vocabulary can be useful for interpreting program flow. Assignment statements with no other side effects can — in principle — be delayed until their grounding carrier is "convoluted" with some other carrier.[20] When modeling eager-evaluation languages, particular edge-types can be designated as forcing a temporal order or else edges can be annotated with additional temporalizing details. Without this extra documentation, however, execution order among graph elements can be evaluated based on other criteria.

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

This order needs to broken, of course, for statements with side-effects. A case in point is the expansion of "**do**-notation" in Haskell: without an *a priori* temporality, Haskell

[20] Of course, the default choice of "eager" or "lazy" evaluation is programming-language-specific, but for abstract discussion of source code graphs, we have no *a priori* idea of temporality; of a program executing in time. This is not a matter of concurrency — we have no *a priori* idea of procedures running at the *same* time any more than of them running sequentially. Any temporal direction through a graph is an interpretation *of* the graph, and as such it is useful to assume that graphs in and of themselves assert no temporal ordering among their nodes or edges.

source code relies on the asymmetric order of values passed into lambda abstractions to enforce requirements that effectual expressions evaluate before other expressions (Haskell does not have "statements" per se). Haskell's **do** "blocks" can be modeled (in the techniques used here) as a series of assignment statements where the anchoring carrier of each statement becomes (i.e., transfers its value to) the sole occupant of a **lambda** channel marking a new function body, which includes all the following statements (and so on recursively). There are two concepts in play here: interpreting any sequence of statements (plus one terminating expression, which becomes a statement initializing a **return** carrier) as a function body (not just those covering the extent of a "block"); and interpreting assignment statements as passing values into "hidden" lambda channels. What *looks* like *one* block in Haskell source code internally maps to a string of blocks interspersed with hidden **lambda** transfers. Operationally, Haskell backs this syntactic convention with monad semantics — **lambda** values passed are not the actual value of the monad-typed carrier but its "contained" value (see [57] or [109] for a review of monads[21]). For sake of discussion, let's call this a *monad-subblock* formation.

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

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

5.2.2 Code Blocks as Typed Values

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

[21] I cite these articles among many because, written by linguists, they bring an extra multi-disciplinary interest; more linguistics-with-Computer-Science perspectives are conspicuous in [76], by the author of [109] in collaboration with a significant contributor to Idris.

Sample 2: Implementing If/Then/Else Blocks

```
void test_if_then_else(quint64 args_ptr)
{
 QVector<quint64>& args = *(QVector<quint64>*)
   (args_ptr);

 int i = 0;
 bool test = false;
 for(quint64 qui: args)
 {
  if(i % 2)
  {
   if(test)
   {
    PHR_Callable_Value** pcv =
      (PHR_Callable_Value**) qui;
    (*pcv)->run();
    return;
   }
  }
  else                                    1
  {
   PHR_Expression_Object** pxo =
     (PHR_Expression_Object**) qui;

   PHR_Channel_Group_Evaluator* ev = (*pxo)->run();
   qint32 i1 = ev->get_result_value_as<qint32>();
   test = (bool) i1;
  }
  ++i;
 }
}
...
void init_basic_functions(PhaonIR& phr,
  PHR_Code_Model& pcm,
  PHR_Channel_Group_Table& table,
  PHR_Symbol_Scope& pss)
{
 init_test_functions(phr, pcm, table, pss);

 PHR_Type_System* type_system = pcm.type_system();
 PHR_Channel_System& pcs = *phr.channel_system();
 PHR_Channel_Semantic_Protocol* lambda =
   pcs["lambda"];
...
 PHR_Channel_Group g1;
...
 {                                        2
  PHR_Type* ty = type_system->get_type_by_name(
    "argvec");
  PHR_Carrier* phc = new PHR_Carrier;
  phc->set_phr_type(ty);                   3
  g1.init_channel(lambda, 1);
  (*g1[lambda])[0] = phc;

  table.init_phaon_function(g1, pss, "if-t-e", 700,
    &if_t_e);
  table.init_phaon_function(g1, pss,
    "test-if-then-else", 700, &test_if_then_else);

  g1.clear_all();
 }
...
}
```

Given a runtime engine based on Channel Algebra, deferred evaluation is relatively straightforward: any delayed expression can be saved according to its channel-package data structure, which can be passed to functions as an encapsulation of the (not-yet-evaluated) expression itself. Code Sample 2 shows an implementation from the demo, on the runtime side. At ❶, a pointer to the relevant unevaluated expression is extracted, and a **run** method is called which completes the hitherto-delayed evaluation. The demo **test_if_then_else** procedure takes an "**argvec**" parameter (❷), which allows a variant number of blocks and expressions to be passed as inputs (analogous to C++ **var_arg** lists). The code around ❸ shows a channel complex being constructed which is then used to register the signature for the **if...then...else** kernel function in a lookup table.

Meanwhile, the hook into this C++ runtime code is demonstrated in Sample 3, in the "channelized" Intermediate Representation used for the demo. One pertinent point in this code is the instruction at ❺, where the name **temp_anchor_channel_group_by_need** suggests how the compiled channel package is being stored on a "by need" basis (internally, it gets a flag suppressing evaluation before being used in runtime procedures). At ❻, a block itself is assigned to a "callable value" type (cf. the "cv" initials). The arguments identified for the kernel procedure (❹), then, alternate between encapsulations of by-need expressions and compiled blocks deemed internally as opaque "callable values".

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

Further discussion of code models via Channel Groups and Channel Algebras is outside the scope of this chapter, but is demonstrated in greater detail in the accompanying codeset. Hopefully, the best way to present Channel Semantics outside the basic **lambda/sigma/return/exception** quartet is via demonstrations in live code. In that spirit, the demo

Sample 3: Channelized Intermediate Representation, with deferred evaluation

```
 .; generate_from_fn_node ;.
push_carrier_stack $ fground ;.
hold_type_by_name $ fbase ;.
push_carrier_symbol $ &test-if-then-else ;.
 .; args ;.
push_carrier_stack $ lambda ;.

push_unwind_scope $ 1 result ;.

 .; unwind_scope: 1 ;.

 .; generate_from_fn_node ;.
push_carrier_stack $ fground ;.
hold_type_by_name $ fbase ;.
push_carrier_raw_value $ #=? ;.
 .; args ;.
push_carrier_stack $ lambda ;.
hold_type_by_name $ u4 ;.
push_carrier_raw_value $ 4 ;.
hold_type_by_name $ u4 ;.
push_carrier_raw_value $ 5 ;.
push_carrier_stack $ result ;.
index_channel_group ;.
coalesce_channel_group ;.
 .; pop ;.
pop_unwind_scope ;.
temp_anchor_channel_group_by_need ;.
 .; end fground entry ;.
hold_type_by_name $ u4 ;.
push_carrier_expression ;.


 .; block ... ;.
@fnp ;.


 .; generate_from_fn_node ;.
push_carrier_stack $ fground ;.
hold_type_by_name $ fbase ;.
push_carrier_symbol $ &prn ;.
 .; args ;.
push_carrier_stack $ lambda ;.
hold_type_by_name $ u4 ;.
push_carrier_raw_value $ 78 ;.
coalesce_channel_group ;.
evaluate_channel_group ;.
delete_temps ;.
delete_retired ;.
clear_temps ;.
reset_program_stack ;.
 .; end of statement ;.
@fne ;.

 .; end block ... ;.

hold_type_by_name $ pcv ;.
push_carrier_anon_fn @ last_source_fn_name ;.
```

code focuses on practical engineering and problem-solving where channel models can be useful, and I'll briefly review its structure and its organizing rationales in the Conclusion.

6 Conclusion

To regard data modeling as just a practical, behind-the-scenes endeavor is to underestimate the scientific richness and importance of data modeling paradigms as theoretical constructs. In science, data models delineate the structure of information generated during scientific investigations (e.g., experiments and field work), and so their structure concretizes scientific theories and/or experimental protocols. Meanwhile, data modeling has to balance the complexity of human concepts with a predictability conducive to software and computational treatments; data modeling therefore helps expose the boundary, in human cognition, between what is mechanical and what is not — between the "mind as computer" metaphor and competing paradigms which treat cognition as situational and embodied.

Because of these larger themes, data modeling can be seen not just as an operational prerequisite for scientific or technology research — and then the conversion of new discoveries into new technologies with practical benefits — but also as an interdisciplinary nexus informed by and relevant to Computer Science, mathematics, Philosophy of Science, Sociology of Knowledge, formal semantics, and so forth.

One key interdisciplinary question is: how can data models be expressive enough to represent cultural and scientific ideas and artifacts — without any sense of conceptual mismatch or simplification — but at the same time work in a software ecosystem? To be employed, that is, in a context where data structures have sufficient stability and classifiability that they are amenable to algorithms and mutations to accommodate different software roles, such as database and GUI presentations? Data models should be *systematic* so that they engender safe, reliable code. On the other hand, digital resources should be expressive enough to represent complex concepts without "dehumanizing" their structure — failing to recognize connections or distinctions which are part of human conceptualization, even if they are technically challenging to model computationally.

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 ([122], [123] 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], [39], [54], [67], [103], [115]). 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 code archive supplementing this chapter operationalizes the notion of "Procedural Hypergraph Ontologies", combining features of Procedural Ontologies and of Directed Hypergraphs that I have presented in this chapter. Procedural Hypergraph Ontologies extend (or diverge 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.

Architecturally, then, the demo includes several data sets repackaged in a hypergraph-serialization format, and,

simultaneously, application-level code which bridges the data sets to GUI components. The code base parses serialized hypergraphs to in-memory hypergraphs and then traverses them, using a kind of visitor pattern, to build in-memory C++ objects, which in turn are mapped to GUI objects. So this chain of processing steps models hypergraph-based data representations and the logistics of incoporating them at the application level. At the same time, C++ objects reconstituted from hypergraphs — as well as the GUI components which receive them — are documented with an Interface Description Language that employs channels for articulating procedural signatures, an IDL which in turn compiles to hypergraph structures leveraged for various code-analysis tasks (for example, generating a testing mechanism integrated with the application code).

The techniques thereby demonstrated can be practically adopted in several ways. On the one hand, concepts like channels and preconstructors can be applied to mainstream programming languages like C++, becoming new design patterns or new coding practices that, over a large code base, can help produce components which are statically analyzable and (by systematic documentation and validation of coding assumption) priotize safety at runtime. On the other hand (as profiled via special languages and Intermediate Representations in the demo), the techniques I have outlined can be used for new data and code models which guide, test, and/or retroactively analyze software components (e.g., using Channel Algebra in a fine-grained Interface Definition Language).

Aside from these practical applications, moreover, I contend that channels and Channelized Hypergraphs can be of interest in topics like linguistics and the Philosophy of Science as well — insofar as data models and representations of information flow ensapsulate the structure of scientific theories, and the conceptual networks that lie behind Natural Language as well as formal semantics.

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 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

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

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

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

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

22 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

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

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

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

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

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

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

29 Matúš Chochlík and Axel Naumann, "Static reflection: Rationale, design and evolution.". http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0385r0.pdf

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

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

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

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

34 Jeremiah Y. Dangler, "Categorization of Security Design Patterns". *Electronic Theses and Dissertations*. (East Tennessee State University), Paper 1119, 2013 https://dc.etsu.edu/cgi/viewcontent.cgi?article=2303&context=etd

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

36 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.

37 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

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

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

40 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

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

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

43 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

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

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

46 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

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

48 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

49 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

50 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

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

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

53 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

54 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

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

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

57 Gianluca Giorgolo and Ash Asudeh, "Monads as a Solution for Generalized Opacity". In *Proceedings of the EACL 2014 Workshop on Type Theory and Natural Language Semantics (TTNLS)*, pages 19–27, Gothenburg, Sweden, April 26-30 2014. http://www.aclweb.org/anthology/W14-1403

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 Dawid R. Ireno, "Dynamic Factory: New Possibilities for Factory Design Pattern". In *Proceedings 28th European Conference on Modelling and Simulation*. Flaminio Squazzoni *et. al.*, editors, 2014. http://www.scs-europe.net/dlib/2014/ecms14papers/dis_ECMS2014_0114.pdf

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

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

73 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

74 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

75 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

76 Oleg Kiselyov and Chung-chieh Shan, "Lambda: the ultimate syntax-semantics interface" https://pdfs.semanticscholar.org/fb41/793ae7098c40fcdb6706b905c48a270b0b48.pdf

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

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

79 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

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

81 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

82 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

83 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

84 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

85 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

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

87 William B. McNatt and James M. Bieman, "Coupling of Design Patterns: Common Practices and Their Benefits". In *Proc. Computer Software & Applications Conf.* (COMPSAC 2001), October 2001. http://www.cs.colostate.edu/pubserv/pubs/McNatt-bieman-Pubs-McnattBieman01.pdf

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

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

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

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

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

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

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

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

96 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

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

98 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

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

100 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

101 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

102 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

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

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

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

106 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

107 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

108 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

109 Chung-Chieh Shan, "Monads for natural language semantics". http://arxiv.org/pdf/cs/0205026.pdf (archived 17 May 2002)

110 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

111 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

112 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

113 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

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

115 Gregor Strle, "Semantics Within: The Representation of Meaning Through Conceptual Spaces". Univ. of Novi Gorici, dissertation, 2012.

116 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

117 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

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

119 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

120 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

121 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

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

123 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

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

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

126 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

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

128 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

129 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

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

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

132 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

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

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

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

136 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

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