Hypergraph Type Theory for Specifications-Conformant Code and Generalized Lambda Calculus

Nathaniel Christen

May 31, 2019

Abstract

This chapter will develop a theoretical framework for the practical problem of introducing complex type-theoretic constructs such as Dependent Types and Typestate into mainstream programming languages. All programming languages have some form of type system, wherein types are associated with symbols in computer code and their corresponding runtime values. Type systems differ according to their level of detail or "expressiveness". In general, the more expressive a type system, the more information is implied by the fact that a symbol or value is associated with a type. Types help to both document and enforce the assumptions which are made as code is written. Accordingly, more expressive type systems describe more detailed coding assumptions, increasing the likelihood that programming errors are discovered before code is deployed, and improving the technology for evaluating computer code's conformance to safety, security, and privacy standards. However, despite these benefits of types' expressiveness, there are several constructions for very expressive type systems that emerge from mathematical type theory and have been incorporated into academic and special-purpose programming languages, but not mainstream, general-purpose languages. This chapter will consider some explanations for this situation and present an essentially new foundation for type theory from which emerges strategies for practically realizing "expressive" constructs — such as Dependent Types and Typestate — while avoiding the pitfalls that have blocked practical adoption of these constructs in the past.

A fundamental aspect of programming is asserting specifications on when bodies of code should be executed — criteria which can be coarse-grained (like requiring that numbers passed to a function must be integers), or finer (like stipulating that a number must lie in a fixed range, or that two lists of numbers must have the same size). Logically separating code which *makes* assumptions — in the course of performing operations (which may have concrete, even physical effects, as in Cyber-Physical Systems) — from code which *checks* assumptions (to ensure that effectual code is not improperly called), simplifies the design and maintenance of both kinds of code — which I'll call "fragile" and "gatekeeper". Fragile code is code which can fail (for instance, throw an exception or cause a software crash) when used improperly. The canonical example of fragile code is a function which can fail when called with the wrong sort of arguments — arguments that have the correct *type* but are not in the proper range, or use incorrect scales of measurements, or do not obey expected inter-argument relationships. Gatekeeper code is then code which examines the arguments intended for fragile functions, preventing the fragile code from executing when it might be unsafe or improper to do so.

While stretches of code may informally be identified as "fragile" or "gatekeeper", it is a good idea to make this distinction explicit and deliberate, as a design pattern. Distinguishing gatekeeper and fragile code is a good example of "separation of concerns": the maxim that code serving different purposes should be logically separated. Gatekeeper and fragile code have distinct roles, and tend to involve different programming techniques. Both kinds of code therefore tend to be clearer and easier to maintain insofar as they are logically separated.

To demonstrate the real-world importance of properly planning and co-ordinating fragile and gatekeeper code, 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, the preeminent "Ubiquitous Sensing for Healthcare" (USH) principles include "transparency" (openness about how USH systems are designed and operationalized) and "trustworthiness" (demonstrably secure engineering, particularly in the context of personal privacy and protecting access to personal medical data).[1] Data in the USH context is generated by physical biomedical devices, and the shape and properties of this data — including details such as its numeric dimensions, scales of measurement, and possible range for component values — are closely tied to the scientific purpose and manufacturing of particular devices.

Therefore, applications which process USH data need to rigorously organize their functionality around specific devices' data profiles. The functions that directly interact with devices — receiving data from and perhaps sending instructions to each one — will in many instances be "fragile" in the sense I invoke in this chapter. Each of these functions may make assumptions legislated by the relevant device's specifications, to the extent that using any function too broadly constitutes a system error. Furthermore, CyberPhysical devices that are not full-fledged computers may exhibit errors due to mechanical malfunction,

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

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 [14] and [39], for example). As a consequence, *error classification* is especially important — distinguishing kinds of software errors and even which problems are software errors to begin with.

To cite concrete examples, a heart-rate sensor generates continuously-sampled integer values whose understood Dimension of Measurement is in "beats per minute" and whose maximum sensible range (inclusive of both rest and exercise) corresponds roughly to the $[40-200]$ interval. Meanwhile, an accelerometer presents data as voltage changes in two or three directional axes, data which may only produce signals when a change occurs (and therefore is not continuously varying), and which is mathematically converted to yield information about physical objects' (including a person's) movement and incline. The pairwise combination of heart-rate and acceleration data (common in wearable devices) is then a mixture of these two measurement profiles — partly continuous and partly discrete sampling, with variegated axes and inter-axial relationships.

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

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

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

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

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

Fragile code is not necessarily a sign of poor design. Sometimes implementations can be optimized for special circumstances, and optimizations are valuable and should be used

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

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

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

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

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

Most type systems provide only relatively coarse classification of a universe of typed values — even though many functions require their arguments to fit more precise specifications than practical type systems allow. This is unfortunate given the premise of "separation of concerns" and the maxim that functions should have single and narrow roles: *validating* input is actually a different role than *doing* calculations. Maximwise, then, functions with fine requirements can be split into two: a gatekeeper that validates input before a fragile function is called, and separate from that the function's own implementation itself. A related idea is overloading fragile functions: for example, a function which takes one value can be overloaded in terms of whether the value fits in some prespecified range. These two can be combined: gatekeepers can test inputs and call one of several overloaded functions, based on which overload's specifications are satisfied by the input.

Despite their potential elegance, most practical programming languages do not supply much language-level support for expressing groups of fine-grained functions along these lines. Dependent Types, typestate, and effect-systems are each models or paradigms which can document function implementation requirements with more precision than can be achieved via conventional type systems alone. Integrating these paradigms into type systems — which is done, at least incompletely, at least in some versions of some programming languages — allows requirements to be confirmed by general type checking, without the need for static analyzers or other "third party" tools (that is, projects maintained orthogonally to the actual language engineering; i.e., to compiler and runtime implementations). So there are no intractable *formal* obstacles to augmenting the expressiveness of practical languages' type systems. Neverthe-

less, such enhancements appear to be either sufficiently difficult to "language engineer", and/or to adversely affect language performance, enough that the benefits of type-expressiveness do not on net improve the language. Or at least, it is reasonable to assume that those responsible for maintaining languages and language tools (specifications, compilers, standard libraries) believe as much, so that Dependent Types (for example) are only internally supported by a few (mostly academic) languages.

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

Programming languages and the code written in them are a kind of structural creole, partly engineered artifacts and machinery existing in virtual/digital spaces, and partly human texts and conventions. The structure and elemental form (grammar, data layout) of these artifacts similarly joins human and engineering concerns. Language design should therefore be informed by human as well as mathematical and computing sciences — after all, a programming language is a *language*, a vehicle of human expression and communication. Code "expresses" routines for computer processing rather than address to other humans, but as programmers we also write code to be understood by others, communicating indirectly via our shared understanding of how computer languages work. So programming languages are indeed communicative media of a certain sort. As human artifacts rigid enough for a digital environment, they are an exceptional forum for exploring human cognition and signification in a structured, formally tractable milieu. Software Language design should be based on human cognitive and communicative structures, as well as on structural mathematics, and philosophy centered on human thought and experience should be able to guide (and learn from) Software Language Engineering (William Rapaport [40] has an interesting discussion along these lines, which is intriguing to read alongside overviews of the OpenCog project I will cite later).

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

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

Of the many approaches to specifications and verification, we might recognize two distinct tendencies. On the one hand, some languages and projects prioritize specifications that are intrinsic to the language and integrate seamlessly and operationally into the language's foundational compile-and-run sequence. Improper code (relative to specifications) should not compile, or, as a last resort, should fail gracefully at run-time. Moreover, in terms of programmers' thought processes, the description of specifications should be intellectually continuous with other cognitive processes involved in composing code, such as designing types or implementing algorithms.

The attitude I just summarized — which perhaps can be called "internalist" — is evident in passages like this (describing the Ivory programming language): "Ivory's type system is shallowly embedded within Haskell's type system, taking advantage of the extensions provided by [the Glasgow Haskell Compiler]. Thus, well-typed Ivory programs are guaranteed to produce memory safe executables, *all without writing a standalone type-checker*" [13, p. 1] (my emphasis). In other words, the creators of Ivory are promoting the fact that their language buttresses via its type system code guarantees that for most languages require external analysis tools.

Contrary to this "internalist" philosophy, other approaches (perhaps I can call them "externalist") favor a neater separation of specification, declaration and testing from the "core" programming language and coding activity. In particular, most of the more important or complex safety-checking does not natively integrate with the underlying language, but instead requires some external source code analyzer, or runtime libraries. Moreover, it is unrealistic to expect all programming errors to be avoided with enough proactive planning, strong typing, and safety-focused paradigms: any complex code base requires some retroactive design, some combination of unit-testing and mechanisms (including those third-party to both the

language and the projects whose code is implemented in the language) for externally analyzing, observing, and higher-scale testing for the code, plus post-deployment monitoring.

As a counterpoint to the features cited as benefits to the Ivory language, consider Santanu Paul's Source Code Algebra (SCA) system described in [36] and [35], [46]:

> Source code Files are processed using tools such as parsers, static analyzers, etc. and the necessary information (according to the SCA data model) is stored in a repository. A user interacts with the system, in principle, through a variety of high-level languages, or by specifying SCA expressions directly. Queries are mapped to SCA expressions, the SCA optimizer tries to simplify the expressions, and finally, the SCA evaluator evaluates the expression and returns the results to the user.
>
> We expect that many source code queries will be expressed using high-level query languages or invoked through graphical user interfaces. High-level queries in the appropriate form (e.g., graphical, command-line, relational, or pattern-based) will be translated into equivalent SCA expressions. An SCA expression can then be evaluated using a standard SCA evaluator, which will serve as a common query processing engine. The analogy from relational database systems is the translation of SQL to expressions based on relational algebra. [36, p. 15]

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

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

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

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

Language engineers, then — particularly for general-purpose, multi-paradigm programming languages — have to work with two rather different constituencies. One community of programmers tends to prefer that specification and validation be integral to/integrated with the language's type system and compile-run cycle (and standard runtime environment);

whereas a different community prefers to treat code evaluation as a distinct part of the development process, something logically, operationally, and cognitively separate from hand-to-screen codewriting (and may chafe at languages restricting certain code constructs because they can theoretically produce coding errors, even when the anomalies involved are trivial enough to be tractable for even barely adequate code review). If this gloss (which I admit rests on some speculation and mind-reading) has any merit, it implies that one challenge for language engineers is to serve both communities. For example, we can aspire to implement type systems which are sufficiently expressive to model many specification, validation, and gatekeeping scenarios, while also anticipating that language code should be syntactically and semantic designed to be useful in the context of external tools (like static analyzers) and models (like Source Code Algebras and Source Code Ontologies).

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

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

After this mostly theoretical prelude, I conclude this chapter with a discussion of code annotation, particularly in the context of CyberPhysical Systems. Because CyberPhysical applications directly manage physical devices, it is especially important that they be vetted to ensure that they do not convey erroneous instructions to devices, do not fail in ways that leave devices uncontrolled, and do not incorrectly process the data obtained from devices. Moreover, CyberPhysical devices are intrinsically *networked*, enlarging the "surface area" for vulnerability, and often worn by people or used in a domestic setting, so they tend carry personal (e.g., location) information, making network security protocols especially important ([4], [26], [39], [43], [44]). The dangers of coding errors and software vulnerabilities, in CyberPhysical Systems like the Internet of Things (IoT), are even more pronounced than in other application domains. While it is unfortunate if a software crash causes someone to lose data, for example, it is even more serious if a CyberPhysical "dashboard" application were to malfunction and leave physical, networked devices in a dangerous state.

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

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

1 Directed Hypergraphs and Generalized Lambda Calculus

Historically, Lambda Calculus represented the first formal model of computation that was reasonably close to actual programming languages, allowing correspondence between Lambda Calculus as a mathematical theory and the design and implementation of early programming languages. Languages have evolved considerably since then, becoming more complex both semantically and syntactically. Semantically, this has been reflected in the proliferation of many 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, and so forth. Syntactically, code representations have evolved which supplement Abstract Syntactic Trees — essentially, unreduced (beyond so-called "β-normal" form) lambda expressions — with more detailed presentations of computer code, like Source Code Algebras and Abstract Semantic Graphs. This richer detail is important not only for program execution, but also other kinds of interaction with computer code, such as querying, testing, scripting, and debugging.

The classical Lambda Calculus is formulated in terms of "Applicative Structures", whose constituents are *symbols* and *terms* — symbols are terms, and more complex terms are created by combining a term which designates a "function" with one or more terms designating "arguments". These arguments can be other terms, so terms can be nested. Applicative Structures are roughly equivalent to mathematical formulae, or in the programming world to simple languages in the LISP family, like SCHEME. Certain symbols in Applicative Structures are subject to "lambda abstraction", which makes the Applicative Structure into something like a programming "function", where "calling" the function amounts to assigning values to the abstracted symbols (which happens at runtime as a program is executed). Upon un-abstracting all symbols an Applicative Structure can then be *evaluated* — the mathematical analogy would be replacing symbols with numbers in mathematical formulae. This model of computation describes functions which have one list of "input" parameters and "return" one value.

Many functions in modern computer code fit that simple profile, but many others do not: modern languages have multiple kinds of "channels" through which "functions" (that is, blocks of code that can be called from other code) can communicate with other code. This affects code syntax as well as semantics: to understand how one function call interacts with its surrounding code, analyzers need to consider more than just values explicitly passed to the function as arguments, or bound to its return value. For example, if the function throws an exception, then we have to look for surrounding **try** blocks to identify how the exception will be handled. We also have to consider side-effects that may not be apparent from the function-call syntax itself. Given **y=f(x,z)**, say, we can assume that the symbols **x**, **y**, and **z** are involved with the function call, but there may be other symbols in earlier or later lines of code that are *also* affected by (or have an effect on) **f**. These relations of influence are intrinsic to modern programming language grammars. They are also intrinsic to Program Semantics, because to estimate how programs will behave in different scenarios it is necessary to consider every form of influence between functions and values — not just the argument-and-return semantics of classical Lambda Calculus.

Retrospectively, Computer Scientists see λ-Calculus as a kind of precursor to the modern theory of programming languages. This attitude probably distorts history to some extent — partly because programming languages themselves did not exist when λ-Calculus was formulated, but partly also because the original vision for λ-Calculus as a mathematical theory was less about blueprints for calculating machines and more about *abstract* formulation of calculational processes. While there were no digital computers at the time, there *was* a growing interest in mechanical computers that would soon evolve into cryptographic machines important during the Second World War, and then into electronic systems with designs that that we would see as early "computers". So the mathematicians of the time *could* have explored formal prototypes for "computing machines", as John von Neumann did eventually.

It is, however, probably more accurate to describe the original purpose of λ-Calculus as a mathematical *simulation* of computations, which is not the same as a mathematical *prototype* for computations. 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., [16], [23], [47]).

This abstract perspective, however, stands at some re-

move from actual code-writing. Programs written with Object-Orientation and Exceptions can always be *mathematically* replicated with programs using only the simplest kind of language. As a result, the properties of more sophisticated programming languages — in particular, the topics formally studied within Software Language Engineering — lie outside the mathematical bounds of early λ-Calculus or related disciplines, such as Recursive Function Theory (which is not to say that they cannot be mathematically studied from other angles, such as Category Theory). From a sufficiently mathematized perspective — one appropriate for, say, the theory of Computable Numbers — distinctions such as that between OO and non-OO procedural code disappear; the mathematical framework is not formulated with theoretical posits that would bring these kinds of practical SLE-oriented distinctions into focus.

However, the subsequent evolution of programming languages *did* inspire the emergence of an ever-widening circle of λ-Calculus variations, whose role was no longer to abstract away from concrete computational implementations for the sake of logico-mathematical investigations but rather to provide formal specifications for models of computation which actual programming languages could then put into practice. So a reasonable history can say that λ-Calculus mutated from being an abstract model for studying Computability as a mathematical concept, to being a paradigm for prototype-specifications of concretely realized computing environments.

Superficially, the earlier λ-Calculus just codifies the basic practices of mathematics. For example, in a formula like $\frac{4\pi R^3}{3}$ the volume of a sphere is expressed in terms of its radius R; but the symbol R is just a mnemonic which could be replaced with a different symbol without the formula being different. It can also be replaced by a more complex expression to yield a new formula: how much bigger is a sphere than a cube? Substitute the formula for a cube's half-diagonal — $\sqrt{3}\sqrt[3]{V}$ where V is its volume — for R in the first formula, to get $\frac{4}{3}\sqrt{27}\pi V$, from which V can be subtracted or divided ([2] has similar interesting examples in the context of code optimization).

The formal review of basic mathematical notation practice was interesting to 20th century mathematicians at a time when general questions were first being asked about the logical completeness and consistency of mathematical systems — which led to mechanical models of computing processes and eventually to computers as we know them. The actual name of a symbol during computations is not significant, only its structural role as an element of aggregate expressions; so the symbol itself is "abstracted" — in the above formulae, R has no formal meaning (the fact that it uses the first letter of "radius" is incidental) and serves only as a place-holder (the correct cube-sphere comparison depends on substituting $\sqrt{3}\sqrt[3]{V}$ *in the right place*). Multiple computations are linked by how names of values in one context are linked to names in another context; "calling" a function essentially entails plugging names in the calling context into names used in the called function's body. So symbols are "abstracted" in that their meaning lies only in the chain of name-to-name links enacted during a computation.

Practical programming languages, however, have many different ways of handing-off values between function bodies. 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. Functions can also "receive" data indirectly from pipes, streams, sockets, network connections, database connections, or files. All of these are potential "input channels" whereby a function implementation may access a value that it needs. In addition, functions can "return" values not just by providing a final result but by throwing exceptions, writing to files or pipes, and so forth. To represent these myriad "channels of communication" computer scientists have invented a menagerie of extensions to λ-Calculus — "Sigma" calculus to model Object-Oriented Programming; λ-Calculus with exceptions, captures, lazy evaluation, or named parameters; etc.

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

Computer scientists have explored these more complex programming paradigms in part by inventing new variations on λ-calculi, as well as new code representations which can take the place of Applicative Structures. 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.

1.1 *Directed Hypergraphs and "Channel Abstractions"*

A *hypergraph* is a graph whose edges (a.k.a. "hyperedges") can span more than two nodes ([21, e.g. p. 24], [31], [34]; [38], [42]). A *directed* hypergraph ("DH") is a hypergraph where each edge has a *head set* and *tail set* (both possibly empty). Both of these are sets of nodes which (when non-empty) are called *hypernodes*. A hypernode can also be thought of as a hyperedge whose tail-set (or head-set) is empty. Note that a typical hyperedge connects two hypernodes (its head- and tail-sets), so if we consider just hypernodes, a hypergraph potentially reduces to a directed ordinary graph. While "edge" and "hyperedge" are formally equivalent, I will use the former term when attending more to the edge's representational role as linking two hypernodes, and use the latter term when focusing more on its tuple of spanned nodes irrespective of their partition into *head* and *tail*.

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

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. I will call DHs meeting this condition *linear*, in the sense that any sequence of nodes appearing within one hypernode will always appear in the exact same order whenever they are included in other hypernodes. Trivially, all reducible DHs are linear.

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

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

1. They are linear.

2. 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).

3. 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 hypernodes, "vertex" will be a general term for both hypo- and hypervertices. Nodes which do have a vertex are called *initialized*. The hypovertices "of" a hypernode are those of its hyponodes.

4. 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.

5. 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.

6. 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.

7. 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.

8. 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).

9. 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.

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

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

12. 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.

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

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

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

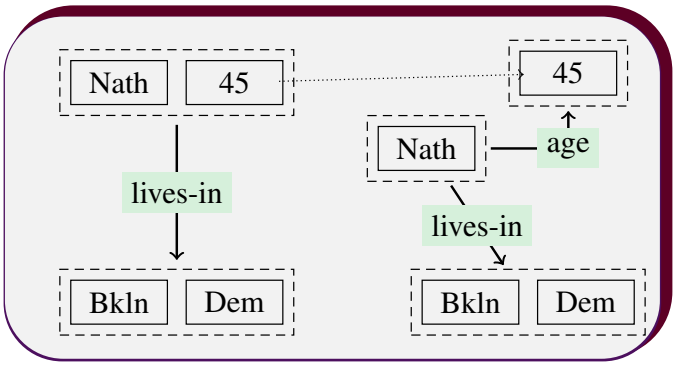

Diagram 1: "Unplugging" a Node

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

These alternatives are not redundant. Generally, representing information via hyperedges connecting two hypernodes implies that this information is somehow conceptually apart from the hypernodes themselves, whereas representing information via hyponodes *inside* hypernodes implies that this information is central and recurring (enforced by types), and that the data thereby aggregated forms a recurring logical unit. In a political survey, people's names may *always* be joined to their age, and likewise their district of residence *always* joined to their political affiliation. The left-hand side representation of the info (seen as an undirected hyperedge) $\langle\lceil\textbf{Nathaniel}\rceil, \lceil\textbf{45}\rceil, \lceil\textbf{Brooklyn}\rceil, \lceil\textbf{Democrat}\rceil\rangle$ in Diagram 1 captures this semantics better because it describes the name/age and place/party

pairings as *types* which require analogous node-tuples when expressed by other hypernodes. For example, any two hypernodes with the same type as ⟨⌈Nathaniel⌉, ⌈45⌉⟩ will necessarily have an "age" hypovertex and so can predictably be compared along this one axis. By contrast, the right-hand ("unplugged") version in Diagram 1 implies no guarantees that the "age" data point is present as part of a recurring pattern.

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

1.2 *Serializing Data via Hypergraphs*

The formal Channelized Hypergraph specification I sketched earlier described both "hypervertices" and "hypovertices" as values associated with hyper- and hyponodes, respectively. There is no *mandated* relation between hyper- and hypovertex values, but usually there should be a sensible transformation between them. In a typical case, hypovertices are used primarily for serialization, while hypervertices are used for analysis and processing. That is, a CH graph as a *runtime* data structure will hold most or all values in hypernodes. When the graph is sent to a different location, the hypervertex values are split into minimal units (like individual strings, numbers, or node-proxies), from which hyponodes are initialized.

Sample 1: Initializing Hypernodes

```
caon_ptr<RE_Node> RE_Graph_Build::make_new_node(
  caon_ptr<RE_Token> token)
{
 caon_ptr<RE_Node> result = new RE_Node(token);
 RELAE_SET_NODE_LABEL(result, token->string_summary());
 return result;
}

caon_ptr<RE_Node> RE_Graph_Build::make_new_node(
  caon_ptr<RE_Call_Entry> rce)
{
 caon_ptr<RE_Node> result = new RE_Node(rce);
 RELAE_SET_NODE_LABEL(result, QString("<call %1>")
   .arg(rce->call_id()));
 return result;
}

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;
}

caon_ptr<RE_Node> RE_Graph_Build::make_new_node(
  caon_ptr<RE_Block_Entry> rbe)
{
 caon_ptr<RE_Node> result = new RE_Node(rbe);
 RELAE_SET_NODE_LABEL (result, QString("<block %1>")
   .arg(rbe->call_id()));
 return result;
}

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
   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);

 return result;
}
...

if(caon_ptr<RE_Call_Entry> rce =
  current_node_->re_call_entry())
{
 if(last_pre_entry_node_)
 {
  caon_ptr<RE_Node> fdef_node = graph_build_->
    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_->
    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_);
  result = current_node_;
  fdef_node << fr_/rq_.Run_Call_Entry >> current_node_;

  current_node_ << fr_/rq_.Run_Data_Entry >>
    tuple_info_node;
  tuple_info_node << fr_/rq_.Run_Data_Entry >>
    entry_node;

  current_node_->delete_relation(
    rq_.Run_Call_Entry, entry_node);
  caon_ptr<RE_Node> tuple_leave_node = graph_build_->
    create_tuple_node(
      RE_Tuple_Info::Tuple_Formations::N_A,
      RE_Tuple_Info::Tuple_Indicators::Leave_Array,
      RE_Tuple_Info::Tuple_Formations::N_A, false);
  tuple_info_node << fr_/rq_.Run_Data_Leave >>
    tuple_leave_node;
 }
}
```

When the graph is later *deserialized*, the hypovertices are then re-aggregated to re-construct the hypervertices, which in turn initialize their corresponding hypernodes.

Figure 1 shows an example of code from the demo where the CH graph builder is adding content to a code-graph, within functions that are used from callbacks triggered by rule-matches in the parser. The methods labeled at the top of the sample (like ❶) show the graph builder creating nodes when provided with values (technically, hypervertices) allocated elsewhere (they are wrapped in a special kind of pointer used for values internal to the graph system). By contrast, ❷ shows similar logic but where the vertex and node are created together, by the same function. As this illustrates, making nodes is a two-step process, where *first* the vertex is allocated, *then* the node is likewise, perhaps in a different code location (these comments apply both to hypo- and hypervertices and nodes).

The freshly created nodes are not yet inserted into a graph; this final step can occur via code like at ❸, where new nodes are joined to prior nodes via a specified hyperedge annotation/classifier, called a "connector" in demo code. The demo library uses C++ operators to modify graph structures, essentially creating a "query language" for graph traversal and manipulation as an embedded DSL (Domain-Specific Language), both adding nodes and (as at ❹) for moving between nodes based on a desired connector (modifying the graph structure can also be done with method-calls, as at ❺). The main point for the present context is that nodes serve as wrappers for ordinary runtime data structures: the CH graph library does not examine or process vertex values directly. Instead, it allows applications to traverse graphs until they find nodes which are of interest for whatever their present purpose, and allows the application to use the node's initialized value however it sees fit (unpacking the vertex as an ordinary C++ pointer).

This illustrates one contrast between CH and RDF: CH graphs can be used in a way that fully decouples graph *structure* from any notion of value-manipulation or semantics. The demo libraries, for instance, recognize only structural relations between values; they do not have any means to "look inside" or use values. As sampled in 2, each graph has a "dominion" which encompasses a set of types that may be used as vertex values. The demo Hypergraph library assumes that all node-values belong to types identified at compile-time, and on that basis enforces some measure of compile-time type checking.

The node classes are equipped with constructors that accept vertex pointers from "dominion" types, and have methods to extract pointers of those types (the actual implementation of these functions is centered at ❶ in 3, which is macro code that gets included while the node classes are being compiled — in general, the list of types asserted in **types.h** generates several different kinds of code depending on which preprocessor macros are defined at the point where the **types.h** file is included). Overall, the node classes hold values (vertices) but do not operate on then; each vertex is opaque to the CH graph engine, stored within a node for subsequent use elsewhere in the application, but only actually used at the application level.

Sample 2: Asserting Vertex Types

```
struct RE_Dominion
{
 ...
 enum class Type_Codes { N_A,
  #define DOMINION_TYPE DOMINION_TYPE_ENUM
  #include "dominion/types.h"
  #undef DOMINION_TYPE
 };
 ...
}

// // in types.h
#ifndef DOMINION_HIDE_NO_NAMESPACE
// // No namespace
#include "relae-graph/dominion-macros.h"
DOMINION_TYPE(qstring, QString, QStr)
...
#endif

#define DOMINION_OUTER_NAMESPACE RZ

#define DOMINION_INNER_NAMESPACE RECore
#include "relae-graph/dominion-macros.h"
DOMINION_TYPE(re_root, RE_Root, RE_Root)
DOMINION_TYPE(re_token, RE_Token, RE_Token)
DOMINION_TYPE(proxy, RE_Node_Proxy, Proxy)
DOMINION_TYPE(re_tuple_info, RE_Tuple_Info,
  RE_Tuple_Info)
DOMINION_TYPE(re_call_entry, RE_Call_Entry,
  RE_Call_Entry)
DOMINION_TYPE(re_block_entry, RE_Block_Entry,
  RE_Block_Entry)
DOMINION_TYPE(re_function_def_entry,
  RE_Function_Def_Entry, RE_Function_Def_Entry)
...
#undef DOMINION_INNER_NAMESPACE


#define DOMINION_INNER_NAMESPACE GBuild
#include "relae-graph/dominion-macros.h"
DOMINION_TYPE(core_function,
  RZ_Lisp_Graph_Core_Function, Graph_CoreFun)
DOMINION_TYPE(empty_tuple, RZ_Lisp_Empty_Tuple,
  EmptyTuple)
...
#undef DOMINION_INNER_NAMESPACE

#define DOMINION_INNER_NAMESPACE GVal
#include "relae-graph/dominion-macros.h"
DOMINION_TYPE(block_info, RZ_Lisp_Graph_Block_Info,
  Block_Info)
DOMINION_TYPE(rz_function_def_info,
  RZ_Function_Def_Info, RZ_Function_Def_Info)
DOMINION_TYPE(rz_code_statement, RZ_Code_Statement,
  RZ_Code_Statement)
DOMINION_TYPE(rz_expression_review,
  RZ_Expression_Review, RZ_Expression_Review)
...
#undef DOMINION_INNER_NAMESPACE

#undef DOMINION_OUTER_NAMESPACE
#undef DOMINION_NODE_TYPE
```

Sample 3: Mapping Vertex Types to Node Constructors

```
// // in relae-node-ptr.h
template<typename DOMINION_Type, typename GALAXY_Type =
    typename DOMINION_Type::Galaxy_type,
  typename VERTEX_Type =
    typename GALAXY_Type::Root_Vertex_type>
class node_ptr
{
 ...
 VERTEX_Type vertex_;
```

```
 template<typename T>
 node_ptr(caon_ptr<T> v) :
   type_code_(
     dominion_get_type_code<DOMINION_Type, T>()),
   vertex_(
     v.template caon_cast<VERTEX_Type>())
 {
 }

public:

 template<typename T>
 caon_ptr<T> as()
 {
  return vertex_.template as<T>();
 }
 ...
};

// //  in rz-re-node.h
class RE_Node : public node_ptr<RE_Dominion>
{
 // //  These fields are used for structuring graphs as
  //    Directed Hypergraph -- this node class is used
  //    for both hyper- and hyponodes.  Client code can
  //    enforce the hypo/hyper distinction by ensuring
  //    at least one of these fields is empty/null --
  //    or else create multi-scale graphs by allowing
  //    nodes to be both hyper and hypo (with both a
  //    parent, and a non-empty hyponode vector).
 QVector<caon_ptr<RE_Node>> hyponodes_;
 caon_ptr<RE_Node> parent_;

public:

 #define DOMINION_TYPE DOMINION_NODE_CONSTRUCTOR
 #include "kernel/dominion/types.h"
 #undef DOMINION_TYPE
 ...
};

// //  in dominion-macros.h
#define DOMINION_TYPE_METHOD(node_method_name, \
  cpp_type_name, type_code_name) \
 caon_ptr<DOMINION_NAMESPACED(cpp_type_name)> \
 node_method_name() const \
 { \
  if(type_code_ == Type_Codes_type::type_code_name) \
    return vertex_.ptr_cast< \
      DOMINION_NAMESPACED(cpp_type_name)>(); \
  return caon_ptr< \
    DOMINION_NAMESPACED(cpp_type_name)>(); \
 }
#endif

#define DOMINION_NODE_ONE_WAY_CONSTRUCTOR( \
   node_method_name, cpp_type_name, type_code_name) \
 DOMINION_NODE_ONE_WAY_CONSTRUCTOR_ \
  (DOMINION_NODE_TYPE, node_method_name, \
    cpp_type_name, type_code_name)
#endif

#define DOMINION_NODE_ONE_WAY_CONSTRUCTOR_( \
   node_type, node_method_name, \
   cpp_type_name, type_code_name) \
 node_type(caon_ptr< \
   DOMINION_NAMESPACED(cpp_type_name)> v) \
   : node_type::node_ptr_base_type(v) {}
#endif

#ifndef DOMINION_NODE_CONSTRUCTOR
#define DOMINION_NODE_CONSTRUCTOR(node_method_name, \
   cpp_type_name, type_code_name) \
  DOMINION_NODE_ONE_WAY_CONSTRUCTOR(node_method_name, \
    cpp_type_name, type_code_name); \
  DOMINION_TYPE_METHOD(node_method_name, \
    cpp_type_name, type_code_name);
#endif
```

This manner of separating content from structure is conceptually and operationally different from RDF: RDF structures capture relations on multiple scales at once — which also means that RDF graphs are suitable for both serialization and analysis. By contrast, CH graphs — or at least those constructed via code sampled here — are not directly serializable, because each node holds a value that is, in its underlying form, a C++ pointer. Serializing the overall graph depends on serializing each vertex, which in turn depends on algorithms varying based on vertex types. Note that "serializing vertices" makes no sense in the RDF context, because RDF nodes by definition are either atomic literals or URLs pointing at web resources. Nodes in CH, in contrast to RDF, have "values" which can be multi-part aggregates of any type (whereas RDF literals are constrained to a handful of basic types like numbers and character strings).

On the other hand, CH graphs can be *transformed* into serializable structures by mapping hypervertices onto sets of hypovertices, thereby appointing hypernodes with an array of hyponodes. Enforcing constraints related to linearity, types, and product-type transforms can result in CHs whose hypernodes exhibit the same structural properties as a program's runtime memory. It is helpful to think of hypernodes as analogous to either C **struct**s or C arrays. Varying the precise requirements on hypernodes allows CHs to mimic the memory conventions of different programming languages (for example, prohibiting non-identical but overlapping hypernodes mimics a language without pointer arithmetic, which as such has no mechanism for creating a reference to a proper subset of the memory area allocated for a typed value). These specifications help ensure that serialization algorithms are straightforward to design: for example, code that serializes hypervertex types to binary buffers (like **QDataStream**) can with little change be adopted to map hypernodes to hyponodes. As such, CH graphs can be used via processing *hypervertices* as runtime data structures, and used via processing *hypovertices* for data sharing and network transfers: the hyponodes become a kind of "transport" layer whose main role lies in moving graphs from place to place.

The fact that CH allows a stricter separation of *structure* and *content* also has ramifications for semantics and analytic capabilities, compared to RDF, which I will discuss next.

1.3 *Channelized Hypergraphs and RDF*

The Resource Description Framework (RDF) models information via directed graphs ([3], [9], [10], and [50] 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, PREDI-

CATE, OBJECT⟩", where the "Predicate" is the label). In practice, however, higher-level RDF notation such as TTL (TURTLE or "Terse RDF Triple Language") and Notation3 (N3) deal with aggregate groups of data, such as RDF containers and collections.

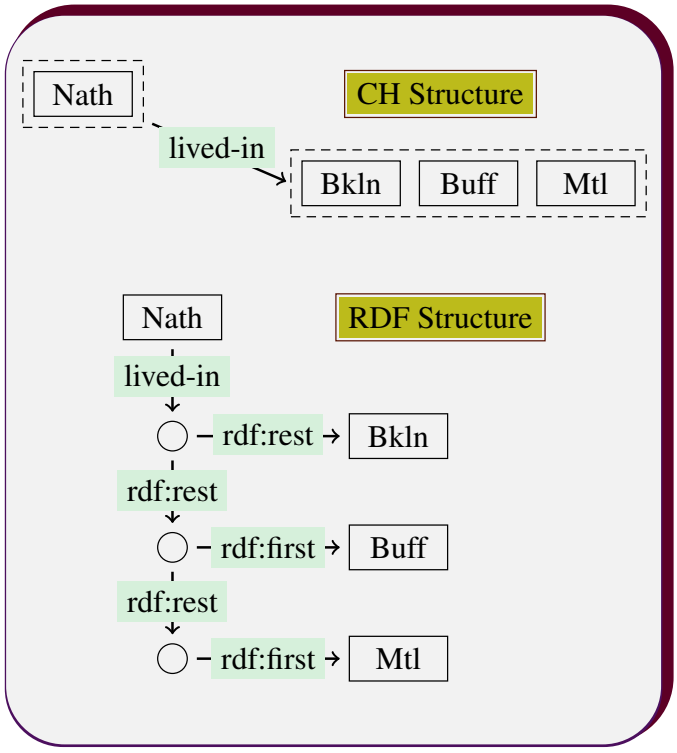

Diagram 2: CH vs. RDF Collections.

For example, imagine a representation of the fact "(A/The person named) Nathaniel, 45, has lived in Brooklyn, Buffalo, and Montreal" (shown in Diagram 2 as both a CH and in RDF). An N3 graph of the sentence might look like this:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix ex: <http://example.org/stuff/1.0/> .

# Turtle (explicit)
ex:Nath ex:lived-in [
  rdf:first ex:Bkln ;
  rdf:rest [
    rdf:first ex:Buff ;
    rdf:rest [
      rdf:first ex:Mtl ;
      rdf:rest ]]] .

# Turtle (simplified)
ex:Nath ex:lived-in [
  a rdf:Seq ;
  rdf:_1 ex:Bkln ;
  rdf:_2 ex:Buff ;
  rdf:_2 ex:Mtl ; ] .

#N3
ex:Nath ex:lived-in ( ex:Bkln, ex:Buff, ex:Mtl ) .
```

Sample 4: Turtle Formats

The final (N3 proper) expression, in particular, actually seems structurally closer to the CH model than to the RDF. If we consider TURTLE or N3 as *languages* and not just *notations*, it would appear as if their semantics is built around hyperedges rather than triples. It would seem that these languages encode many-to-many or one-to-many assertions, graphed as edges having more than one subject and/or predicate. Indeed, Tim Berners-Lee himself suggests that "Implementations may treat list as a data type rather than just a ladder of rdf:first and rdf:rest properties" [5, p. 6]. That is, the specification for RDF list-type data structures invites us to consider that they *may* be regarded integral units rather than just aggregates that get pulled apart in semantic interpretation.

Technically, perhaps, this is an illusion. Despite their higher-level expressiveness, RDF expression languages are, perhaps, supposed to be deemed "syntactic sugar" for a more primitive listing of triples: the *semantics* of TURTLE and N3 are conceived to be defined by translating expressions down to the triple-sets that they logically imply (see also [50]). This intention accepts the paradigm that providing semantics for a formal language is closely related to defining which propositions are logically entailed by its statements.

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

Why, that is, are Brooklyn and Democratic grouped together? What concept does this fusion model? There is a presumptive rationale for the name/age blank (i.e., the fusing name/age by joining them to a blank node rather than allowing them to take edges independently): conceivably there are multiple 45-year-olds named Nathaniel, so *that* blank node plays a key semantic role (analogous to the quantifier in "*There is* a Nathaniel, age 45..."); it provides an unambiguous nexus so that further predicates can be attached to *one specific* 45-year-old Nathaniel rather than any old ⟨⌈**Nathaniel**⌉, ⌈**45**⌉⟩. But there is no similarly suggested semantic role for the "place/party" grouping. The name cannot logically be teased apart from the name/age blank (because there are multiple Nathaniels);

but there seems to be no *logical* significance to the place/party grouping. Yet pairing these values *can* be motivated by a modeling convention — reflecting that geographic and party affiliation data are grouped together in a data set or data model. The logical semantics of RDF make it harder to express these kinds of modeling assumptions that are driven by convention more than logic — an abstracting from data's modeling environment that can be desirable in some contexts but not in others.

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

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

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

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

Also, RDF types are less intrinsic to RDF semantics than in CH (see [37]). The foundational elements of CH are value-tuples (via nodes expressing values, whose tuples in turn are hypernodes). Tuples are indexed by position, not by labels: the tuple $\langle \lceil \textbf{Nathaniel} \rceil, \lceil \textbf{45} \rceil \rangle$ does not in itself draw in the labels "name" or "age", which instead are defined at the type-level (insofar as type-definitions may stipulate that the label "age" is an alias for the node in its second position, etc.). So there is no way to ascertain the semantic/conceptual intent of hypernodes without considering both hyponode and hypernode types. Conversely, RDF does not have actual tuples (though these can be represented as collections, if desired); and nodes are always joined to other nodes via labeled connectors — there is no direct equivalent to the basis-level CH modeling unit of a hyponode being included in a hypernode by position.

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

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

Finally, as a last point in the comparison between RDF and CH semantics, it is worth considering the distinction (introduced, notably, in the "OpenCog" system) between "declarative knowledge" and "procedural knowledge" [21, especially pages 182-197]. According to this distinction, canonical RDF data exemplifies *declarative* knowledge because it asserts apparent facts without explicitly trying to interpret or process them. Declarative knowledge circulates among software in canonical, reusable data formats, allowing individual components to use or make inferences from data according to their own purposes.

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

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

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

While Ontologies are valuable for coordinating and integrating disparate semantic models, the Semantic Web has perhaps influenced engineers to conceive of semantically informed data sharing as mostly a matter of presenting static data conformant to published Ontologies (i.e., alignment of "declarative knowledge"). In reality, robust data sharing also needs an "alignment of *procedural* knowledge": in an ideal Semantic Network, procedural capabilities are circled among components, promoting an emergent "collective procedural knowledge" driven by transparency about code and libraries as well as about data and formats. The CH model arguably supports this possibility because it makes type assertions fundamental to semantics. Rigorous typing both lays a foundation for procedural alignment and mandates that procedural capabilities be factored in to assessments of network components, because a type attribution has no meaning without adequate libraries and code to construct and interpret type-specific values.

Still, having just identified several notable differences between RDF and the Semantic Web, on the one hand, and Hypergraph-based frameworks, on the other — perhaps OpenCog and its "Atom Space" system, and certainly the CH model highlighted in this chapter — I hope not to overstate these differences; both belong to the overall space of graph database and graph-oriented semantic models. RDF graphs are both a plausible serialization of CH graphs and a reasonable interpretation of CH at least in some contexts. In particular, there are several Ontologies that formally model computer source code. This implies that code can been be modeled by suitably typed DHs as well. A rigorous review of DH code models may be premature before I discuss "channels", but assume provisionally that such a model does exist. So, for any given program, assume that there is a corresponding DH representation which I will call the *implementation graph*; the code thereby represented I'll call an *implementation body* or just "body".

Assume moreover that these programs do not always run the same way: that their behavior and results depend on *inputs* which are fixed before the program begins, and produce "outputs" once it ends (by "program" I mean an integral body of computation in general, such as the implementation of one function; not just "computer programs" in the sense of particular binary executables or "applications"). This implies that some hypernodes represent and/or express values that are *inputs* to a body, and 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. Intrinsically, the *order of evaluation* is indeterminate — it is neither (necessarily) eager nor lazy, and neither concurrent nor sequential, but perhaps could be described as "detached": each evaluation is a detached computation unless some additional structure convolutes them. Therefore, one requirement to CH semantics is to describe how the "detached" evaluation model translates to either lazy or eager paradigms.

Values can input into and output out of bodies in different ways; so this model of computation is very incomplete. Most of the rest of this chapter will focus on expanding it with greater detail.

1.4 *Type Systems' Architecture*

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

One way to systematize the diversity of type systems is to assume that, for any particular type system, there is a category $\mathbb{T}$ of types conformant to that system. This requires modeling important type-related concepts as "morphisms" or maps between types. Another useful concept is an "endofunctor": an "operator" which maps elements in a category to other (or sometimes the same) elements. In a $\mathbb{T}$ an endofunctor selects (or constructs) a type $\mathcal{T}_2$ from a type $\mathcal{T}_1$ — note how this is different from a morphism which maps *values of* $\mathcal{T}_1$ to $\mathcal{T}_2$. Some basic morphisms and endofunctors include the following:

1. "Selectors" for the first and second elements in a type-product $\mathcal{T}_1 \times \mathcal{T}_2$: note how this is a way of saying that product-types exist, since they are the types for which the "select second" endofunctor exists.

2. Given type $\mathcal{T}_1$, an endofunctor yielding $\mathcal{T}_1 \times \mathcal{T}_1$ for which both the "select first" and "select second" endofunctors yield back $\mathcal{T}_1$. We can then define arbitrary-sized tuples of $\mathcal{T}_1$ wherein a "select first" endofunctor yields $\mathcal{T}_1$ and a "select rest" endofunctor yields another $\mathcal{T}_1$-tuple.

3. Given types $\mathcal{T}_1$ and $\mathcal{T}_2$, there is a type comprising functions whose domain is $\mathcal{T}_1$ and codomain is $\mathcal{T}_2$ ($\mathbb{T}$ is "Cartesian Closed"), so an endofunctor $\mathcal{T}_1 \times \mathcal{T}_2 \Longrightarrow \mathcal{T}_1 \to \mathcal{T}_2$. This kind of type can be generically called a *function-type*, though I more often prefer looser terms such as "function-like" types because the word "function" has multiple (interrelated but importantly different) meanings.

4. As a special case, let **Unit** be a type holding one sole **unit** value. Then morphisms **Unit** $\to \mathcal{T}$ are equivalent to selecting a single value that can instantiate $\mathcal{T}$, motivating the concept of a "set of values" spanned by $\mathcal{T}$.

5. Given types $\mathcal{T}_1$ and $\mathcal{T}_2$, where $\mathcal{T}_2$ represents a (finite) set of nonnegative integers, then a morphism $\mathcal{T}_1 \to \mathcal{T}_2$ "enumerates" the elements of $\mathcal{T}_1$; so we can construct arbitrary types with a finite set of symbols (having no particular meaning, at least internal to the type system), as domains of an Enumeration.

6. Given any type $\mathcal{T}$, we can associate an enumeration E (so an endofunctor $\mathcal{T} \to E$) which represents "states" that meaningfully partition the set of $\mathcal{T}$-values into groups with similar behaviors or properties; then morphism $\mathcal{T} \to E$ maps "elements" of $\mathcal{T}$ to their associated states.

7. Given types $\mathcal{T}_1$ and $\mathcal{T}_2$, we can consider variations on "sum types" (as endofunctors mapping from $\mathcal{T}_1 \times \mathcal{T}_2$), including

the type whose *values* can be either in $\mathcal{T}_1$ or $\mathcal{T}_2$, and a type whose "typestate enumeration" unifies E of $\mathcal{T}_1$ and $\mathcal{T}_2$. In Haskell, for example, the **Maybe** endofunctor uses a special one-valued "Nothing" type and augments any other type by adding this value, to represent missing or "null" data; similar constructions yield zero-able pointers in other languages.

Overall type systems are built up from a smaller set of "core" types via operations like products, sums, enumerations, and "function-like" types. We may think of the "core" types for practical programming as number-based (booleans, bytes, and larger integer types), with everything else built up by aggregation or encodings (like ASCII and UNICODE, allowing types to include text and alphabets). 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). Ultimately, a type system $\mathbb{T}$ is characterized (1) by which are its core types and (2) by how aggregate types can be built from simpler ones (which essentially involves endofunctors and/or products).

In Category Theory, a Category $\mathbb{C}$ is called "Cartesian Closed" if for every pair of elements e_1 and e_2 in $\mathbb{C}$ there is an element $e_1 \to e_2$ representing (for some relevant notion of "function") all functions from e_1 to e_2 [7]. The stipulation that a type system $\mathbb{T}$ include function-like types is roughly equivalent, then, to the requirement that $\mathbb{T}$, seen as a Category, is Cartesian-Closed. The historical basis for this concept (suggested by the terminology) is that the construction to form function-types is an "operator", something that creates new types out of old. A type system $\mathbb{T}$ first needs to be "closed" under products: if $\mathcal{T}_1$ and $\mathcal{T}_2$ are in $\mathbb{T}$ then $\mathcal{T}_1 \times \mathcal{T}_2$ must be as well. If $\mathbb{T}$ is *also* closed under "functionalization" then the $\mathcal{T}_1 \times \mathcal{T}_2$ product can be mapped onto a function-like type $\mathcal{T}_1 \to \mathcal{T}_2$.

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

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

1.5 *Kinds of Abstraction*

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

So, different hypernodes in a CH are subject to different kinds of abstraction. Speaking rather informally, hypernodes can be grouped into *channels* based on the semantics of their kind of abstraction. More precisely, channels are defined initially on *symbols*, which are associated with hypernodes: in any "body" (i.e., an "implementation graph") hypernodes can be grouped together by sharing the same symbol, and correlatively sharing the same value during a "runtime manifestation" of the CH. Therefore, the "channels of abstraction" at work in a body 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 $/\!/_{\text{LAM}.xy}$.

Separate and apart from notating channels vis-à-vis one graph, we also need a way to represent the semantics of different kinds of channels; labels like "lambda" and "return" do not carry much significance outside the primitive distinction of "input" and "output". Here I will mention only simple examples of "formulae" related to channels. Let $/\!/_{\text{RET}}$ be the name of a channel kind, such as the output of functions returning normally. Let $\#/\!/_{\text{CH}}$ denote the size of $/\!/_{\text{CH}}$; the formula $\#/\!/_{\text{ret}} \leq \mathbf{1}$ thereby asserts that functions can return at most one value. Let $/\!/_{\text{EXC}}$ be the name of a channel kind intending to represent thrown exceptions. Since a function cannot *both* return a value and throw an exception, these two kinds of channels are interrelated. In particular, when modeling the semantics of a specific programming language, we may observe that in any runtime manifestation the return and exception channels cannot both be non-empty. A programming environment may also stipulate that every function must either return a value or throw an exception, meaning that after a function returns the $/\!/_{\text{RET}}$ and $/\!/_{\text{EXC}}$ runtime channels cannot both be empty, either. So exactly one of the channels must be non-empty, which can be notated like $/\!/_{\text{RET}} \asymp /\!/_{\text{EXC}}$.

Formulae can also express how channels combine to describe possible function signatures: if every function *must* have **lambda** and **return** channels and *may* have an exception channel, this could be notated like ⫽LAM!, RET!, EXC?. Note that such notation relates to function *signatures*, not runtimes: a function signature may declare that it *can* throw exceptions, but usually *not* do so, so the exception channel is usually empty. Having a runtime-empty kind of channel is different from not having that kind of channel at all. Function signature options may be combined with formulae about channel sizes, yielding notation like ⫽LAM!*,RET!%,EXC?%, asserting that an allowable function signature *must* include **lambda** and **return**, *may* include exceptions, where lambda channels can be of zero or greater size and the other channels sized at most one.

Finally, adding an axiom about the restricted interrelationship between **return**s and **exception**s yields a formula like ⫽LAM, RET, EXC : LAM!*,RET!%,EXC?% :: RET≍EXC; the initial "⫽LAM, RET, EXC" asserting in effect "there exist" channels satisfying these categories. Collectively I will say that formulae like these describing channel kinds, their restrictions, and their interrelationships describe a *Channel Algebra*. The purpose of a Channel Algebra is, among other things, to describe how formal languages (like programming languages) formulate functions and the rules they put in place for inputs and outputs. If χ is a Channel Algebra, a language adequately described by its formulae (with respect to functions and function-types) can be called a χ-language. The basic Lambda Calculus can be described as a χ-language for the algebra ⫽LAM, RET : LAM!*,RET!%.

This wording can also be extended to types. A Channel Algebra describes the channels that may (or must) be declared for a complete type signature. Let χ be a Channel Algebra governed by the characterization ⫽LAM, SIG, RET, EXC : LAM!*,SIG?%,RET!%,EXC?% — representing an Object-Oriented language where ⫽SIG is a "Sigma" channel as in "Sigma Calculus" (written as ς-calculus: see e.g. [1], [53], [18]). These channels are associated with function "signatures", and a function's signature determines its type. So we can say that the Type System $\mathbb{T}$ manifest in the underlying Object-Oriented language is a "χ-type-system" and is "closed" with respect to χ in that valid signatures described using channel kinds in χ correspond to types found in $\mathbb{T}$. Types may be less granular than signatures: as a case in point, functions differing in signature only by whether they throw exceptions may or may not be deemed the same type. But a channel construction on types in $\mathbb{T}$ must also yield a type in $\mathbb{T}$.

I say that a type system is *channelized* if it is closed with respect to some Channel Algebra. Channelized Hypergraphs are then DHs whose type system is Channelized. We can think of channel constructions as operators which combine groups of types into new types (this operative dimensions helps motivate describing channel logics and their formulae as "algebras"). Once we assert that a CH is Channelized, we know that there is a mechanism for describing some Hypergraphs as "function implementations" some of whose hypernodes are subject to kinds of abstraction present in the relevant Channel Algebra. The terse notation for Channel formulae and signatures describes logical norms which can also be expressed with more conventional Ontologies. So Channel Algebra can be seen as a generalization of (RDF-environment) Source Code Ontology (of the kinds studied for example by [28], [29], [30], [48], [51], [52]). Given the relations between RDF and Directed Hypergraphs (despite differences I have discussed here), Channel Algebras can also be seen as adding to Ontologies governing Directed Hypergraphs. Such is the perspective I will take for the remainder of this chapter.

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

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

Some TXL expressions designate function-types, but not all, since there are many types (like integers, etc.) which do not have channels at all. While a TXL lies "above" a CXL by adding provisos that yield type-definition semantics from CXL expressions, the TXL simultaneously in a sense lies "beneath" the CXL in that it provides expressions for the non-functional types which in the general case are the basis for CXL expressions

of functional types, since most function parameters — the input/output values that populate channels — have non-functional types. Section §3 will discuss the elements that "populate" channels (which I will call "carriers") in more detail.

Identifying CH code-graphs with function-typed values — values that are instances of function-like types — provides an elegant theoretical foundation for exploring functional types and, potentially, Functional Programming in general (among other things providing a useful definition of "function types" in the first place, insofar as these are types which need to be described via a CXL, not just the underlying TXL). One of the central tenets — arguably *the* central tenet — of Functional Programming is that "functions are values"; their "life cycle" as values constructed, used by and potentially passed to other functions, and potentially then destructed, should be seen as effectively the same as any other value. It is interesting that this understanding of function-typed values sometimes gets overlooked in the context of discussions about "pure" functions. Contra the attention afforded to pure functions, there is no reason to single out effect-free functions as particularly valuable or paradigmatic function-typed values, at least none which seems compelling on type-theoretic grounds.

Function-typed values are not "mathematical" functions in the sense of one-to-one or many-to-one mappings: functions f_1 and f_2 may be different (as function-typed values) even if $f_1(x) = f_2(x)$ for all x. The definitive criteria for functions are their implementations, which can be modeled as CH-graphs; particularly since many "impure" functions depend on external data outside their explicit inputs, so **f(x)** may yield different values at different times for the same **x**. As "normal" values, functions implicitly have "constructors" like any other type; and most of these constructors work by starting from a formal representation of implementation source-code. This picture, which I will analyze further in the next section — of function-typed values as values constructed from code graphs, so constructing function-typed values reveals a mapping (or even functor) from (suitably abstracted) implementation code graphs to suitably channelized types — provides a formal theory of function types that avoids direct reference to functions' input values and the specific relations they construct between inputs and outputs.

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 functions as *values constructed* from code-graphs.

2 Channel Algebra and Value Constructors

Many Type- and Category-theoretic approaches to computer science discuss topics like *type systems* at some remove from concrete programming practice. This abstraction is valuable — it discovers best practices and conceptual rigor that may be obscured by practical considerations. But it also forces researchers and practitioners to entertain a degree of code-switching, mentally circuiting between concepts whose full profiles emerge only in theory and in practice, respectively. Here, my goal is to develop type theory within a foundation informed more by programming practice. In particular, the *values* which are inhabited by types are understood not as mathematical abstractions, but as the results of mappings ranging over code-graphs — either as values constructed from single nodes (whose vertices are, or include, source code literals) or from complexes of hypernodes, representing collections-type instances or function bodies. Attending to concrete implementational details in type systems still entails a level of abstraction — identifying practical differences in computing environments logically implies a certain abstract vantage point — but the abstractions are different in kind and site.

Formalism in spirit but not in delivery may seem little more than a philosophical exercise, and someone could fairly object that a new framework for investigating (in particular) applied type theory cannot be effectively contrasted with competing paradigms unless it is formalized in turn, enough to establish a ground of comparison. I will respond, however, that *implementations* also establish a ground of comparison. Many sophisticated type-theoretic ideas are understood in theory but realized only in experimental, "academic" programming languages (or not at all). As a result, we have good reason to consider the Software Language Engineering and library development that expands type-system expressiveness at a practical level — that introduces formal methods to coding practice (either as core language features or as libraries) — as the most important formalism; formalism in the aegis of implemented enhancements to compiler and/or runtime capabilities, that tangibly meliorate programming languages' effective use. The framework developed in this chapter, accordingly, is one designed to be a motivation for code generation and analysis tools, that can fit organically into compiler pipelines or application runtimes. Published alongside this chapter are code libraries and examples which hopefully make a case that these practical applications are reasonable. In effect, readers inclined to explore formal models should think of the associated code set as the *de facto* formalization of what I am more informally suggesting in the text, using an actual programming language (specifically LISP and C++) in lieu of a logicomathematical metalanguage reliant on formal definitions and theorems.

One recurring topic here will be similarities and dissim-

ilarities between competing type systems. In practice, whatever our preferred approach to "type theory" in the abstract, different programming environments reveal nontrivial differences in how types are formed and used, and even in what "types" are to begin with. I prefer to define "types" in a conceptual circle with *values* and *functions*. In this co-definition, where the concepts are equally primordial, the purpose of types is to classify values; and this classification is necessary because it allows functional polymorphism, which in turn allows code re-use.

Digital technology as we know it would be essentially impossible without polymorphism (see e.g. [11], [41], [15], [20], [27], [32], which all cover polymorphism and generic programming from a theoretical perspective). Since there are many types defined in any useful computing environment, it would be prohibitively time-consuming (even, depending on the type system, perhaps mathematically impossible) to re-implement functions for every possible type. Instead, functions are conceptualized as inherently polymorphic: their implementations can be run with a plurality of possible types exhibited by their parameters. For example, an algorithm to sort a list of values can work with many types of values; it can be implemented in a single function reused in many contexts.

At the same time, there are limits on polymorphism's propriety — sometimes differences in type require differences in implementation. For example, sorting character strings (to alphabetize a list of names, say) demands different algorithms than sorting numbers (in C, say, passing character arrays to a simple **sort** function is "correct" from the compiler's point of view but yields incorrect results). The logical complement to polymorphism is *overloading*, where multiple functions are declared that are designated via equivalent notations but implemented differently — typically, where one function *symbol* refers to several possible function *bodies*. While serving competing goals — unifying vs. disunifying implementations relative to type variation — polymorphism and overloading are mutually reinforcing. Polymorphic implementations can include overloaded function calls, so that the outer function implementation can accept a wider range of argument types than the ("inner") functions it calls in turn. Combining polymorphism and overloading allows code to be reused wherever possible, while differences that would inhibit code reuse get factored into separate functions. A function — called within an outer function's implementation — may need different implementations for different types, but this does not force the *outer* function to be rewritten for the same range of cases where the *inner* function needs specialized implementations. Polymorphism and overloading, in consort, allows "outer" and "inner" functions to have greater or fewer distinct implementations, as the needs arise, independent of one another.

As a concrete example, a **sort** function can work with both numbers and character strings, even though (for reasons I alluded to earlier) comparison functions will be implemented differently for different types. By overloading these comparison functions, one single **sort** code body can end up calling many different comparison functions, branching to whichever overloaded comparison is appropriate based on the input arguments' types (see 5). Or, consider a function that evaluates a list of values for evidence of a trend to increase: that is, return a count of values which are temporary maxima (greater than all preceding values). The basic elements of this algorithm — keeping track of a result count and latest maximum and incrementing the count whenever the temporary maximum changes — are independent of the values' type (see 6). The actual value comparison at (1) (for measuring whether the current value is greater than the prior maximum), however, *will* vary across types, so the outer function becomes most reusable insofar as the "inner" functions (specifically the comparison procedures) are overloaded — the outer function relying on overloaded functions to actually perform the relevant comparisons.

```
template<typename LIST_Type, typename VAL_Type,
  typename COMPARISON_FN_Type>
void _sort(LIST_Typ<VAL_Type>& list, COMPARISON_FN_Type fn)
{
 // ... generic implementation
}

template<typename LIST_Type, typename VAL_Type>
void sort(LIST_Typ<VAL_Type>& list)
{
 _sort(list, [](const VAL_Type& lhs, const VAL_Type& rhs)
   -> bool
 {
  return lhs < rhs;
 });
}

template<>
void sort(QVector<QString>& list)
{
 _sort(list, [](const QString& lhs, const QString& rhs)
   -> bool
 {
  return lhs.toLower() < rhs.toLower();
 });
}
```

Sample 5: Templates and Explicit Specialization

```
template<typename LIST_Type, typename VAL_Type>
int monotone_increasing(const LIST_Typ<VAL_Type>& list)
{
 if(list.length() == 0)
 {
  return 0;
 }
 VAL_Type current_maximum = list[0];
 int result = 1;
 for(int i = 1; i < list.length(); ++i)
 {
  if (list[i] > current_maximum)
  {
   current_maximum = list[i];
   ++result;
  }
 }
}
```

Sample 6: Polymorphic Operators and Overloading

All of this is common-sense to programmers, but I emphasize these elementary concepts to call attention to the intersecting role of types and functions as foundational constructs: a function body has a set of types for which it applies, distributed amongst its various parameters (or, more generally, "channels of communication"); no two identical function bodies can have different associated type-sets in this sense. Conversely, types are different insofar as they offer an opportunity for overloading: if **x** and **y** have different types, then the notations **f(x)** and **f(y)** use symbol "**f**" to designate a function that may be overloaded, and may require different implementations for **x** and **y**, a requirement which could not arise if not for **x**'s and **y**'s type difference. Consequently, the interplay of polymorphism and overloading is constitutive of the space of types to begin with — types are different if there is a possible overload-resolution (in the context of a polymorphic function accepting both types) which diverges specifically because of this difference. There are other ways to ground the possibility of a rigorous notion of "type", but this implementation-focused perspective is the one I start from here.

In this chapter I am defining concepts like "function body" and "implementation" in terms of Directed Hypergraphs. In practice, any sufficiently complete and self-contained stretch of computer code (the precise terms of these criteria will depend on semantics internal to each programming language) can be compiled to an Intermediate Representation — for example, an Abstract Semantic Graph — which captures all significant properties of the original code. Intermediate Representations are in a useful sense formal representations, qualified by unambiguous criteria of which representational structures are allowed and of their possible transformations into other structures. So for any type system we can assume there is an equally rigorous "code representation" system through which we can define a code body, for example, as the code which must get executed when a particular function implementation is deemed the proper resolution, in a calling context, on type grounds.

In this chapter, I am also making the basically axiomatic assumption that any acceptable Intermediate Representation can be itself represented by Channelized Hypergraphs — an assumption motivated by arguing that any computing engine can be modeled by a suitably expansive combination of λ-calculi, and any such combination can (or so I propose) be modeled as the "Channelization" of some Directed Hypergraph Ontology. In combination, these two assumptions produce a paradigm according to which any type system can be associated with a Source Code Ontology such that any *function-typed* value can be identified with a code-graph that *implements* that function, or at least serves as an Intermediate Representation amid the processes that compile the function's implementation to executable byte- and/or machine code.

When modeling code via DH, *all* values can be initialized from subgraphs (including individual nodes, as in numeric literals) — not only function-typed values. However, function-typed values have extra complications which deserve to be examined one at a time.

2.1 *Initializing Function-Typed Values*

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

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

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

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

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

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

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

2.2 *Addressability and Implementation*

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

To see why, consider first that type systems are intrinsically pluralistic: there are numerous details whereby the type system underlying one computing environment can differ from those employed by other environments. These include differences in how types are composed from other types. There is therefore no abstract vocabulary (including the language of mathematical type theory) that provides a neutral and complete way of describing types across systems. Instead, each system has its own structure of multi-type aggregation, and so requires its own style of type description (mathematically, there is no one "Category" of types, but rather multiple candidate Categories with their own logic). So there is no single, universal "Type Expression Language": each type system has its own TXL with subtly different features than others.

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

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

constructors. Figure 7 shows a comparison (in this discussion I mostly skip over the technical detail that in C++, at least, one cannot actually take the address of a constructor function; but this is related to C++ "constructors" having dual roles of allocating memory and initialization: we *can* take the address of an initialization function that would be analogous to a "pure" value constructor, like **construct** in the following code).

```
template<typename VAL_Type>
struct My_List {
// hide the constructor so client code must use an initializer
 static My_List construct();
private:
 My_List() ...
};

template<>
struct My_List<int> {
 static My_List construct();
private:
 My_List() ...
};

int main(int argc, char *argv[])
{
 auto test1 = &My_List<int>::construct(); // ok
 auto test2 = &My_List::construct(); // compile error
}
```

Sample 7: Addressable vs. Multi-Addressable Initializers

Addressability refers at one level, as the word suggests, to "taking the address" of functions (and accordingly to function-pointers); but here I also refer to a broader question of how functions can be designated from vantage points outside their immediate implementation — code searches, scripting engines, IDE-based code exploration, and other reflection-oriented use cases. Language engines should try to minimize their reliance on "temporary function values" which are opaque to these reflection-oriented features. And yet this can complicate the implementation of type-system features. To reiterate: the goal of "expressive" type systems is to define types, on many occasions, fairly narrowly. Doing so allows specifications on the preconditions for calling a function implementation to be encoded directly in the type system, making it possible to restrict the diversity of values which a function may receive and thereby allow the implementation to make assumptions which would be unwarranted otherwise. For example, safety certification can proceed by documenting the assumptions that each safety-critical function implementation makes and then that those assumptions are reasonable given how each function are called. Such an "assume-and-justify" two-step is easier when assumptions are modeled via structures intrinsic to the type system.

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

For a language designer, however, it raises questions. What should happen if someone tries to construct an **int**⟮**0,100**⟯ value with the number, say, **101**? How should the range-test code be exposed for reflection (is it a separate function; is it a regular function-typed value or some alternative data structure, and how does that affect its external designating)? What if the number comes from a location that opaque to the language engine, like a web API: should the compiler assume that the API is curated to the same specifications as the present code or should it report that there is no way to verify that the declared **int**⟮**0,100**⟯ type is being used correctly? Moreover, would such choices lead to behind-the-scenes, perhaps auto-generated code which is hard to wrangle for reflection and development tools? Are the benefits of automated gatekeeping worth the risk of codewriters' mental disconnect with language internals? Given these questions, it is reasonable for a language designer to *allow* certain sorts of types to be described, but programmers must explicitly implement them for the types to be actualized and available for use in programs. Therefore there is a difference between a *described* type and an *actual* type, and the key criterion of actual types is an addressable value constructor. So the crucial step for type-theoretic design promoting desired software qualities, like safety and reliability, is to successfully pair the *description* of types which have desirable levels of specification and granularity, with the *implementation* of types that realize the design patterns promised by their description. In some cases the description must become more complex and nuanced to make implementation feasible.

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

2.3 *Described Types and Actual Types*

I consider it a maxim of the kind of type system I propose here that each system is grounded in a universe of typed values;

that for each system there is a Category $\mathbb{T}$ of types which in principle satisfies Category Theoretic axioms; that "to be a value with a type" means to be the image of a morphism from a type with one sole inhabitant (modeling the phenomenon of "initializing" a symbol, or variable, or temporary rvalue, or holder, or as I'll generically call it a "carrier" holding a value); that each inhabitable type has at least one "data"- or (the term I'll prefer) "value"-constructor, which is a function returning a value of that type, and that at least one of these value constructors does not in turn require a value of the type (more generally that the system of value-constructors is "acyclic" in a sense I'll define later); and that functions (in the sense of procedures that are implemented in code and can be called from other functions) are themselves typed values. These last two assumptions contain the seed of a debate with "ordinary" type theories: most Category-flavored work on types seems to take as a general principle that morphisms (which must have a nontrivial structure if $\mathbb{T}$ is a nontrivial Category) are analogous to functions (i.e., *implemented* functions, as opposed to mathematical constructs at one remove from an actual programming language), to the degree that such a distinction is recognized.

The notion of "type systems" I adopt here sees them not logical abstractions but as engineered artifacts (as are languages themselves). Types are not *sets* (or some Constructive or intuitionistic Category that can stand in for sets, adding mathematical structure or logical discipline to sets' notorious philosophical haziness [25]). In the general case there is no *a priori* association between a type and the sets of values it can house (on any particular computing environment). As a matter of logic, we can say, for example, that allocating memory and appending the resulting pointer to a pointer-list yields a new pointer-list — but there is no guarantee that this operation will succeed (there may, among other issues, be no memory available). An expression, as in this case, which appears to logically describe a plausible value of a type (an "inhabitant") may not actually be constructable or representable at some computing environment and some time (or even any time). Similar issues are sometimes addressed by a *modal* type theory (cf., e.g., [19]) 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").

Along similar lines, a logical description of a plausible type — say, the type of all functions that take equal-sized lists of integers — may not correspond to a type that can be concretely implemented in a given programming languages and environment. There is no ambient mathematics which makes a type available for practical use as soon as it is unambiguously described — alongside inferences driven by the Curry-Howard isomorphism that there are some *necessarily uninhabited* types, we can say in the present setting that there may be *contingently* uninhabited types, types which cannot be inhabited *in some particular computing environment* because there are no constructors implemented; or because the environment has no compile-time or run-time mechanism for enforcing the requirements stipulated by the type — insofar as they are used either to describe values or as an element in typing judgments.

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

A consequence of this framing is that defining what exactly constitutes a type — via an expression notating a type description and a corresponding type implementation — depends intrinsically on defining what constitutes a *function*, and particularly an *implementation* or *function body*. Moreover, since functions are implemented in terms of other functions, another primordial concept is a function *call*. Reasoning abstractly about functions needs to be differentiated from reasoning about available, implemented functions. For example, a common functional-programming idiom is to treat the composition of two unary functions as itself a function-typed value to pass to contexts expecting other function-typed values. In my perspective here, $f \circ g$ may be a *plausible* value, but it is not an *actual* value without being implemented, whether via a code graph (spelling out the equivalent of $\lambda x.fgx$) or some indirect/behavioral description (analogous to **inc** represented as ⟨**&add, 1**⟩). Consider function pointers: what is the address of $f \circ g$ if that expression is interpreted in and of itself as evaluating to a functional value? This suggests that a composition operator does not work in function-like types quite like arithmetic operators in numeric types (which is not unexpected insofar as functional values, internally, are more like pointers than numbers-with-arithmetic). 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). To put it differently, an **address-of** operator *may* be available for $f \circ g$ if it is available for **f** and **g**, but this depends on language design; it is not an abstract property of type systems.

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

Aside from showing a concrete case of type-theoretic assertions which may be refined via modal operators (giving us means to assert possible but not necessary "Curry equivalence"), this example also points to how formal criteria of what, in fact, constitutes a function, influences our notion of what in fact constitutes a type. Exploring this relation, I contend, is one way to consider the relations and variation between formulations like λ and π (process) calculi.

In a mathematical type-theoretic context — such as Typed λ-Calculus — there is a prior notion of "function application" or "term" — so terms (which may be arbitrarily nested) denote function calls, while lambda-abstracting symbols within terms forms the basis of notating function bodies. For serious programming languages, however, there are (as I have stressed) many λ-calculi — adding forms of abstraction for objects, exceptions, permissions, mutable state, and any other mechanism through which functions may pass values to and from other contexts. This variation intersects with type theory insofar as, for example, type operators (abstracting to bind symbols to types rather than values) combine with other variations on the lambda theme. There are many "lambda cubes" because the Barendregt construction can be duplicated for many variations of the underlying lambda calculus — of the calculus for value-abstraction irrespective of what type-abstractions are allowed; e.g., abstracting objects apart from function arguments.

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

To be precise, consider how "carriers" of values — the holders, bound to symbols in source code, which designate internal runtime representations of values and so make them accessible for computations — become initialized. Consider first symbols being initialized from source-code literals, like **int x = 90**. The value-constructor in this case must take a value representing a "literal", some sort of character string obtained directly from source code. In other cases, initialization can unfold over several steps. Consider **x = 90; y = 100; g = grade(x, y)**: the "**g**" is initialized from a function return, so the actual value constructor whose result initializes **g** is inside **grade**, and a chain of intermediary carriers brings the result to its target. But we can assume that by following the chain of carriers and value constructors we can eventually reach a value constructor which works on a raw source code literal. Call a type "acyclic" (in the context of constructors) if it has either a default constructor (a value constructor which takes no arguments), or a value constructor which takes one argument of a single global type that includes (representations of) raw source code literals, or a value constructor which only takes other acyclic types, such that a call diagram whose nodes are types and whose edges are value constructors meeting these criteria forms a Directed Acyclic Graph. Call a type system acyclic if every inhabitable type in the system is acyclic in this sense. Intuitively, this means that every type can be built up from source code literals by a chain of value constructors.

Notice, however, that *function* values seem, at least in the canonical cases — see item 1 on page 22 — to be defined from "bodies" which are not single literals, but rather aggregate code-spans with semantically and syntactically regulated internal structure. I assume that code is written in a specific language and that, in the context of that language, any sufficiently complete code-span can be compiled to an Abstract Semantic Graph (or similar graph-like Intermediate Representation). The logic of this representation may vary among languages and/or type systems — optimal graph representation of source code is an active area of research, not only for compiler technologies but also for code analyzers and "queries" and for code deployment on the Semantic Web. In this chapter, I assume that an adequate graph representation will be isomorphic to a Directed Hypergraph. So, assume that there is a "code-DH" type such that every code-span suitable for compilation into a function-implementation is a value of that type. Assume also that every source code literal is a **code**-DH graph with one hypernode and no edges. In that case, initializing carriers from literals is one example of initializing carriers from code-graph instances more generally, including initializing "function" values.

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

Notice that one single literal may initialize carriers of multiple types; the number "0" could become a signed or un-

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

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

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

Consider a function **f** which takes values that must be in a specified range ⟨**r**⟩ (say, an integer between ⟨**0,100**⟩). By extension, suppose we want to overload **f** based on whether its argument (say, **x**) falls inside or outside ⟨**r**⟩. This is not hard to achieve if **f**'s *type* internally references a *fixed* ⟨**r**⟩. Let **T** be a symbol that quantifies over the typeclass of types with magnitude/comparison operators, and **ranged**<**T**> be a type formed from **T** and two **T**-values, implementing the semantics of closed intervals over **T**: so **ranged**<**T, t1, t2**> is a "type-expression" mapped to a type constructor yielding a single type (not a type family, typeclass, or higher-order type). According to my "maxims", a type is concretely inhabitable if it has an acyclic value constructor, which in turn is a value of a function type. So, each acyclic type is associated with one or more *values* that are in a vague Curry-Howard sense "proofs" of its inhabitability. Note that "inhabitable" does not in this context mean the abstract complement of "uninhabitable" types whose "corresponding" propositions would be paradoxical if the type had a value. Rather, "inhabitable" means that value constructors are available to be called and type-instances can be used computationally, e.g., passed to other functions.

For any type-with-intervals **T** and **T**-range ⟨**r**⟩, a compiler (for instance by specializing a template) can produce a value constructor that takes **T**-values and tests (or coerces) them such that the constructor only returns a value in ⟨**r**⟩. This can then be the input type for a function which requires ⟨**r**⟩-bounded input (see Figure 8). What to do when a values *fails* the range-test is another question which I set aside for now. We can similarly define a "non-range" type which only accepts values *not* in ⟨**r**⟩; and, since these two types (the in-range and the out-of-range) are different, we can overload **f** on them. So, assuming we accept ⟨**r**⟩ being fixed, we can achieve something semantically — "overload-wise" — like dependent typing. Of course, Dependent Types as a programming construct are more powerful than this: an example of "real" dependent typing would be something like an **f** which takes *two* arguments: the first a range, and the second a value within the range. We want the type system to be engineered allowing the condition on the second argument to be verified as part of *typechecking*.

```
template<typename VAL_Type, VAL_Type min, VAL_Type max>
class ranged
{
 ...
 VAL_Type _v;
 ...
public:
 ranged(VAL_Type v)
 {
  if( (v < min) || (v > max)
  {
   // throw exception?
  }
  _v = v;
 }
};
```

Sample 8: Templated Range Constructor

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

We could certainly write a function that takes a range and a value and ensures that the value fits the range — perhaps by throwing an exception if not, or mapping the value to the

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

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

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

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

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

A reasonable language, conversely, may have "non-addressable" function values: for example, suppose a lambda passed to a function is defined just via an operator, like an $f \circ g$. Say, sorting two lists of strings on a comparison which calls a "to lowercase" function before invoking a less-than operator. This could be notated with a lambda block, but some languages

may allow a more "algebraic" expression, something meaning "lower-case then less-than", with the idea that function values can be composed by rough analogy to numbers being added (see item 3 on page 22). In this case, the *value constructor for the function type* does not take a code graph, at least not one visible near the $f \circ g$ expression, just as the value constructor for **x** in **x** = **y** + **z** is hidden somewhere in the "**y** + **z**" implementation. A language can reasonably forbid taking the address of (or forming pointers to) "temporary" function values derived algebraically from other functions. Indeed, the concepts of "constructed from a code graph" and "addressable" may coincide: a compiler may allocate long-term memory for just those function-implementations it has compiled from code-graphs.

But value-constructors are not just any function-value: they have a privileged status vis-à-vis types, and may be invoked whenever an appropriately-typed value is used. Allowing large type families (like one type for each **int** — similar to "inductive typing" as discussed by Edwin Brady in the context of the Idris language [6, p. 14]) effectively forces a language to accept non-addressable value-constructors. Conversely, forcing value constructors to be addressable prohibits "large" type families — like types indexed over other (non-enumerative) types — at least as *actual* types. A language engine may declare that value constructors, in short, cannot be "temporary" values. This apparently precludes full-fledged Dependent Types, since dependent-typed values invariably require in general some extra contextual data — not just a function-pointer — to designate the desired value constructor at the point where a value, attributed to the relevant dependent type, is needed. It may be infeasible to add the requisite contextual information at every point where a dependent-typed value has to be constructed — unless, perhaps, a description of the context can be packaged and carried around with the value, sharing the value's lifetime.

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

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

2.5 *Dependent Types and Typestate*

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

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

of states for one value — which must then have different types at different times, since this is the whole point of modeling successive states via alternations in type-attribution.

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

A good real-world example of the overlap between Dependent Types and typestate (also grounded on file input/output) comes from the "Dependent Effects" tutorial from the Idris (programming language) documentation [24]:

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

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

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

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

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

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

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

> Traditionally, this is simple: types are compile-time only, values are run-time, and erasure consists simply of erasing types. In a dependently typed language, however, erasing types alone is not enough [45, 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 [8], and [12, page 210]). This means that a lot of contextual detail is *not* erased; Idris provides machinery to join executable code and user specifications onto *types* so that they take effect whenever affected types' values are constructed or passed to functions. Despite a divergent technical background, the net result is arguably not vastly different from an Aspect-Oriented approach wherein constructors and function calls are "pointcuts" setting anchors upon source locations or logical run-points, where extra code can be added to program flow (see e.g. [33], [22], [54]). Recall my contrast of "internalist" and "externalist" paradigms, sketched at the top of this chapter: Aspect-Oriented Programming involves extra code added by external tools (that "modify" code by "weaving" extra code providing extra features or gatekeeping). Implementations like Idris pursue what often are in effect similar ends from a more "internalist" angle, using the type system to host added code and specifications without resorting to some external tool that introduces this code in a manner orthogonal to the language proper. But Idris relies on Haskell to provide its operational environment; it is not clear how Idris's strategies (or those of other Haskell and ML-style Dependent Type languages) for attaching runtime expressions to type constructs would work in an imperative or Object-Oriented environment, like C++ as a host language.

This discussion emanated from Idris examples based on file-management typestate, but it generalizes to other cases, such as range-delimited types or states. Practically, working with scenarios like range-bounded values — which in principle exemplifies programming tasks where Dependent Types can be a useful idiom — in practice arguably ends up more like typestate management as exemplified by gatekeeping vis-à-vis, say, files (only call *this* function if *that* file is open). A range-interval is typestate-like in that the practical intent is to affix certain gatekeeper functions to an accompanying value so that it will never be incorrectly used — for instance, that it will never be modified to lie outside its assigned range. Conforming to a range restriction is more like a typestate than a type: indeed, a range-plus-value pair has an obvious covering via two typestates (the value is either in or out of its closed interval).[2] So the semantics of Dependent Types can practically be captured via a typestate framework — and perhaps vice-versa, since typestates can be seen as type families indexed over (possibly enumerative) types; file typestates as a family indexed over ⟨**closed**, **open**, **nonexistent**⟩, for example.

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

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

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

3 Channels and Carriers

The term "channels of communication" is usually associated with process calculi, wherein channels pass values between procedures, which in turn are combined sequentially or in parallel. The possibility of "bidirectional" channels between concurrent processes gives channels more structure, and points to the wider scope of process calculi, compared to lambda calculi. There is a well-known "embedding" of (classical) λ-Calculus into process calculi, although there does not appear to be comparable quantities of research into process-algebraic interpretations of "expanded" λ-calculi with objects, exceptions, and so forth. Since process algebras and calculi prioritize the analysis of computations that run in parallel — and the algebraic combinations of procedures, such as concurrent vs. sequential — the narrower themes of λ-abstraction in one single procedure may seem tangential to serious process analysis.

If real, such an assumption is unfortunate, because the underlying semantics of communicating values between disparate computing procedures — even apart from issues of concurrency, or from issues concerning whether two superficially

different λ-expressions are structurally equivalent (i.e., apart from the main analytic themes of process- and λ-calculi, respectively) — is important in its own right. Any value passed between procedures *may* be reinterpreted as belonging to a different type (the number **1** can be an integer in one context that gets interpreted as the decimal **1.0** somewhere else), which *may* even cause a modification (**1.1** gets truncated to just **1**, say, if the decimal is passed to a function expecting an integer). Moreover, inter-procedure communications *may* be subject to gatekeeping and/or overloading which blocks the target procedure from running if the passed values are incorrect (relative to some specification) or which selects one from a set of possible procedures to run based on the values passed for each occasion.

These processual issues give rise to different operators than conventional process-calculi, but they can still be introduced as algebraic structures on groups of procedures. Let π_1, π_2, and π_3 represent three procedures; we can use notation like $\pi_1 \multimap \pi_2$ to represent a "guarded" inter-procedure combination, wherein a gatekeeper is in effect that may *block* π_2. We may also use notation like $\pi_1 \multimap \lceil \pi_2, \pi_3 \rfloor$ to represent a "polymorphic" inter-procedure combination, wherein a gatekeeper selects one of π_2 or π_3 as the proper procedure to target based on passed values. Note that the former case is a special example of the latter if we assign π_3 in the first notation as a "null" procedure, which does nothing, having no actions or effects.

These inter-procedure relations have nontrivial semantics even if we do not include concurrency or bidirectional communication, so that "channels" are closer in spirit to lambda abstraction than to stateful "message-lines" as in the Calculus of Communicating Systems. As the expansion of lambda calculi toward Object-Orientation (in the "Sigma" or ς-calculus) and other variations shows, even a simpler notion of channels, generalized from lambda abstractions (from which bidirectional mutable channels can then be modeled) is not uninteresting. In the literature, criteria like "guarded choice", supplementing underlying process calculi, appears to be understood principally at the level of *procedures* — for one example, the case of "choosing" one of several procedures is similar to an operator that unifies multiple procedures into *one* procedure with an "either or" logic: "π_1 *or* π_2" is a procedure which may on some occasions execute as π_1 and elsewhere as π_2 (see for instance [49]). With this either-or logic defined on procedures, inter-procedure combinations (where one procedure sends values to a second, causing the second to begin) which are guarded and polymorphic can be modeled as combinations where the second procedure is an either-or sum of multiple more-specific procedures (possibly including one "do nothing" no-op).

This model, however, neglects the details which guide how either-or choices are understood to be made, according to the intended theoretical models. An important class of guarded-choice cases is where choices are made on the basis specifically of the *values* sent between procedures — say, dispatching to different function-bodies according to whether or not a number is in a range `(r)`. These cases can perhaps be described as *localized guarded choice* because all information pertinent to the guard is derived from the passed values (and not from any global or ambient state). We can further identify *constructor-localized* guarded choice as cases where the "guards" are simply the value-constructors for values passed in to the target procedure (maybe cast). In a language like C++ one or more "hidden" functions can execute before a called function actually begins, insofar as "copy constructors" may be in effect, controlling how values are copied from one function's context to another. Without deliberately modeling process calculi at a technical level, then, C++ does provide a "hook" where various process-related ideas can be implemented.

Insofar as Localized Guarded Choice bases guarded-choice semantics solely on inter-procedure passed values, I would argue that it lies in a theoretically intermediate position between λ-style and process calculi. From one perspective, Localized Guarded Choice is a variation on inter-procedure combination, wherein one already-executing procedure initiates a second by passing one or more values between them. This perspective is perhaps the more intuitive when seen from the point of view of the prior procedure.

However, Localized Guarded Choice is also a variation on λ-abstraction, because it presumes that the abstracting of some symbol in the target procedure's implementation is not abstracted indiscriminately, but rather can only be furnished with values conformant to some specification. We can envision a "Local Guard Calculus", maybe dubbed γ-calculus, which works on the assumption that for each λ-abstraction there is a corresponding value constructor which executes as a procedure before a target procedure is called (and may even prevent the target procedure from starting, e.g. by throwing an exception). From the perspective of the prior, calling procedure, this hidden execution appears to be an added procedural layer, an intermediary function that lies between itself and the target. From the perspective of the target procedure, however, the intermediary value-constructors appear more as logical guarantees keyed to its implementational specifications — i.e., as abstraction-refinements. In short, depending on perspective, this hypothetical Local Guard Calculus can be seen as either a special case of Lambda or Process calculi, respectively.

Since the relevant issues of gatekeeping and overloading are "intermediate" in this sense, they are not central foci of either genre of calculi, and call for a distinct terminology and conceptualization. I will accordingly think of function implementations as for practical purposes "procedures" which communicate via "channels", but my sketch of "Channel Algebra" is not a process calculus or process algebra *per se*, and will skirt around foundational process-oriented topics like concurrency, or the possibility of stateful, bidirectional channels. Moreover, my ambition is to model source code and behavior via semantic

graphs which can fit into the RDF (and DH/CH) ecosystems, a reconstruction that can start at the simpler non-concurrent, single-threaded level. So for the duration of this section I will attend exclusively to the semantics of functions passing values to other functions so as to initiate the procedures which the target functions implement; setting aside considerations such as whether the prior function "blocks" until the second function completes (which represents sequential-process semantics) or, instead, continues running, concurrently.

Suppose one function calls a second. From a high level perspective, this has several consequences that can be semantic-graph represented — among others, that the calling function 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 f; this graph is only semantically well-formed if f's representatum has functional type (by analogy to the well-formedness criteria, at least in typed settings, 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".

For every function called, there is a corresponding function implementation which has its own graph representation. Assume this implementation includes a *signature* and a *body*. With sufficient compiler work, the body can be expressed as an Abstract Syntax Tree and, with further analysis, an Abstract Semantic Graph — which in turn will identify semantic connections such as the scoped relation between a symbol appearing in the signature and the same symbol in the body — these symbols will all share the same value (unless a symbol in a nested lexical scope "hides" the symbol seen in the parent scope).

This implicitly assumes that symbols "hold" values; to make the notion explicit, I will say that symbols are *carriers* for values. It may be appropriate to consider literals as carriers (as demonstrated by the C++ **operator""**, the conversion of character-string literals to typed values is not always trivial, nor necessarily fixed by the language engine). For now, though, assume that carriers correspond to lexically-scoped symbols (I also ignore globals). 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. I assume a carrier is associated with a single type throughout its career, and can only hold values appropriate for its type. 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.

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

3.1 *Carriers and Careers*

In material fact, computers work with "numbers" that are electrical signals; but in practice, we reason about software via computer code which abstracts and is materially removed from actual running programs. In these terms, we can note that "computers" (in this somewhat abstract sense) do not deal *directly* with numbers (or indeed other kinds of values), but rather deal with "symbols" that express, or serve as placeholders for, or "carry" values. Even "literal" values are composed of symbols: the number (say) **123**, which has a particular physical incarnation in electrons, is represented in source code by a Unicode or ASCII character string (which materially bears little resemblance to a "digital" **123**).

This primordial fact is orthogonal to type systems. The underlying notion of type theory is of *typed values* — the idea that the whole universe of values that may be encountered during a computation can be classified into types, so a particular value always has one single declared type (though it may be a direct candidate for classification as other types as well). There is a core distinction between *types* and *sets*: two distinct types can be instantiated by the same set of possible values, and one type can cover a different spectrum of values on different computers. For example, any type which has an expandable data collection — say, a list of numbers — can have values which would take ever larger amounts of memory; e.g., as lists grow larger (I made essentially the same comment about pointer-lists on page 25). Since different computers have different available memory at different times, there is no fixed relation between the set of "inhabitants" of these types which can be represented on different computers, or on one single computer at different times. Type theory therefore belongs to a logicomathematical (or philosophical) foundation distinct from set theory. Instead of defining types from sets of values, types are instead built up from more primitive types via several operators, most notably "products", or "tuples" wherein a fixed number of values of (in the general case) distinct types are grouped together (as I outlined from one perspective within subsection §1.4).

But type theory still consider these "values" as abstract, primitive constructs. Both types and values are in essence undefinable concepts, for which there are no deeper concepts in terms of which they can be defined (although this de-facto non-definition can be given a rather elegant gloss with Category Theory). Recognizing that the type-value pair does not itself include the medium through which a value is *represented* in computer code, we can say that an equally primordial concept can be (say) a *carrier*, a symbolic expression which "holds" a value, or the anticipation of one. So we introduce "carriers" as a third fundamental concept. All carriers, in this theory, have one single (declared) type; and all carriers can hold one (or in some contexts many) values; so there is an interrelationship between the notions *type*, *value*, and *carrier*.

Carriers come in two broad groups: *literals* (like "**123**") which embody a single value everywhere they occur in source code, and *symbols* (like **x**) which can have different values when they occur in different places in source code. There may be many "instances" of a single source code symbol in one code base, each of which, if they occur in different "scopes", may have their own value unrelated to the others. In general one symbol might carry different values at different time, though some programming environments may restrict this variability.

Immutable symbols are tightly bound to their values; they have one single value for the full time that they have any value at all, and therefore act as "transparent" proxies for their value — especially if languages discourage constructs like non-initialized values and **address-of** operators (which can cause symbols to hold no-longer-meaningful values).

Because "unitialized" carriers and "dangling pointers" are coding errors, within "correct" code, carriers and values are bound tightly enough that the whole carrier/value distinction might be considered an artifact of programming practice, out of place in a rigorous discussion of programming languages (as logicomathematical systems, in some sense). But even if the "career" of symbols is unremarkable, we cannot avoid in some contexts — within a debugger and/or an IDE (Integrated Development Environment, software for writing programs), for example — needing to formally distinguish the carrier from the value which it holds, or recognize that carriers can potentially be in a "state" where, at some point in which they are relevant for code analysis or evaluation, they do not yet (or do not any longer) hold meaningful values. 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. I'd make the analogy to how lambda calculus models "symbol rewriting" as a formal part of its theory, rather than a notational intuition which reflects mathematical conventions but is seen as part of mathematical "metalanguage" rather than mathematical "language". The history of mathematical foundations reveals how convention, notation, and metalanguage tends to become codified into theory, models, and (mathematical) language proper. Likewise, we can develop a formal theory of carriers which models that carriers have different states — including ones where they do not hold meaningful values — separate and apart from whether the underlying type system allows for mutable value-holders, or pointers, references to lexically scoped symbols, and other code formations that can lead to "undefined behavior".

Consider a carrier which has been declared, so it is assigned a type and belongs to some given scope in its source code, but has not yet been initialized. The carrier does not therefore, in the theory, hold any value at all. This is different from holding a default value, or a null value, or any other value that a type system might introduce to represent "missing" information. In practice, of course, insofar as a "carrier" refers to a segment of memory, the carrier will be seen by the computer as having "some" (effectively random) value; some bit pattern left behind by some other calculation, which obviously has no "meaning". Programming languages usually talk about the (mis)use of uninitialized variables (i.e., in this context, carriers) as leading to "undefined behavior" and therefore beyond any structured reasoning about code, and leave it at that. We can be more rigorous however and define being uninitialized as a *state* of a carrier, as is holding a meaningful value (once it *is* initialized) and then, some time later, no longer holding a meaningful value, because its associated memory has been deemed recyclable (typically, once a symbol falls out of scope). Any carrier therefore has at least three kinds of state, which we can call *pre-initialized*, *initialized*, and *retired*.[3]

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

Taking this further, we can define a *channel* as a list of carriers which, by inter-carrier transfers, signify (or orchestrate) the passage of data into and out of function bodies (note that this usage varies somewhat from process calculi, where a channel would correspond roughly to what is here called a single carrier; here channels in the general case are composed of multiple car-

[3] Again, though, we should not think of (say) "Preinitialized" as a *value* with a *type*. A carrier declared as type **int32**, say (32-bit integer) can only hold values of this type, when it holds any value at all, which precludes interpreting "Preinitialized" as some kind of "null" value which is somehow arranged to be an instance of **int32**.

riers). I'll use the notation $\not\mapsto$ to represent inter-carrier transfer: let $\mathfrak{C}_1$ and $\mathfrak{C}_2$ be carriers, then $\mathfrak{C}_1 \not\mapsto \mathfrak{C}_2$ is a transfer "operator" (note that $\not\mapsto$ 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 $\not\mapsto$ 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 $\mathfrak{C}_1 \not\mapsto \mathfrak{C}_2$ 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**$_1 \not\mapsto$ **f:lambda**$_1$: here the first carrier in the **return** channel of **g** transfers to the first carrier in the **lambda** channel of **f** (the subscripts indicate the respective positions).

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

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

Let $\mathfrak{C}_1$ and $\mathfrak{C}_2$ be two carriers. The $\not\mapsto$ operator (representing a value passed from $\mathfrak{C}_1$ to $\mathfrak{C}_2$) encompasses several specific cases. These include:

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

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

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

4. A value transferred between a **return** channel and either a **lambda** or **sigma** channel, as part of a nested expression or a "chain of method calls". So in **h(f(x))**, the value held by the carrier in **f**'s **return** channel is transferred to the first carrier in **h**'s **lambda**. An analogous **return**$\not\mapsto$**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 $\not\mapsto$ 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 $\not\mapsto$ corresponding to item (3): a transfer effectuated by a function call, where $\mathfrak{C}_1$ is at the call site and $\mathfrak{C}_2$ is part of a function's signature. If f_1 calls f_2 then $\mathfrak{C}_1$ is in f_1's context, $\mathfrak{C}_2$ is in f_2's context, and $\mathfrak{C}_2$ is initialized with a copy of $\mathfrak{C}_1$'s value prior to f_2 executing. A *channel* then becomes a collection of carriers which are found in the scope of one function and can be on the right hand side of an $\mathfrak{C}_1 \rightarrowtail \mathfrak{C}_2$ operator.[4]

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

So, to review my discussion in this section so far, I started with the process-algebraic notion of inter-procedure combinations; but whereas process calculi enlarge this notion by distinguishing different syntheses of procedures (such as, concurrent versus sequential), I have focused instead on the communication of values between procedures. The semantics of inter-procedure value-transfers is unexpectedly detailed, because it has to recognize the possibility of nontrivial copy semantics, casts, overloading, and perhaps "gatekeeping". Furthermore, in addition to these semantic details, analysis of value-transfers is particularly significant in the context of

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

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

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

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

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

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

More precisely, the **f(x)** example represents a carrier transfer whose target is part of f_2's **lambda** channel, which we can notate $\mathfrak{C}_1 \looparrowright^{\textbf{lambda}} \mathfrak{C}_2$. Furthermore, the full statement **y = f(x)** shows a transfer in the opposite direction: the value in f_2's ***return*** channel is transferred to the carrier **y** in the *payload*. This relation, involving a return channel, can be expressed with notation like $\mathfrak{C}_2 \looparrowright^{\textbf{return}} \mathfrak{C}_1$. 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. The distinction between functions' "inputs" and "outputs" can be more rigorously stated, with this further background, as the distinction between channels in function signatures which receive values *from* carriers at a call site (inputs), and those *from which* values are obtained as a procedure has completed (outputs).

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

```
,fnd ::: Fn_Doc*;
fnd \== default;

,kenv ::: KCM_Env*;
kenv \= (envv "KCM_Env*");

fnd -> init kenv;

,test-fn ::: .(int) $-> extern;

fnd -> read "test-fn";
```

Sample 9: Sample IDL Code

The demo code for this chapter uses an Intermediate Representation which builds complexes representing both function signatures and dynamically evaluated function-calls. Figure 9 shows code in the special Interface Description Language

which includes a simple signature (❶). It also has several dynamic calls, including one to create a new object of a C++ class called **Fn_Doc** (❷) and one to retrieve a preconstructed object of a class called **KCM_Env** (❸) (in this language the == operator is used when the right-hand-side expression is a value constructor, and there are assignment operators, notated with a preceding back-slash as in \= or \==, specifically for assigning values to previously-uninitialized symbols).

```
; fn_sig_ ...
(PROGN
 (KA::KC :|kcg_clear_all| KCG)  1
 (KA::KC :|kcg_add_lambda_carrier_via_type_name| KCG "int")
 (SETQ KTO (KA::KC :|kcm_type_object_from_channel_group| KCG)))
(setq kcs (ka::kc
 :|kcm_promote_overloadable_function_type_binding_to_statement|
  "test-fn" kto))
(setq cmd-pkg (ka::kc :|kcm_statement_to_command_package| kcs))
(ka::kc :|kcm_direct_eval| cmd-pkg)
; _fn_sig (no def) ...

(ka::kc :|kcg_clear_all| kcg)
(setq kto (ka::kc  2
  :|kcm_type_object_by_qregistered_type_name| "Fn_Doc*") )
(ka::kc :|kcm_add_type_binding| "fnd" kto)
(setq kcx (ka::kc
  :|kcm_promote_type_binding_to_statement_via_type_
default_literal__let|
  kcg "fnd"))
(setq kcs (ka::kc :|kcm_promote_expression_to_statement| kcx))
(setq cmd-pkg (ka::kc :|kcm_statement_to_command_package| kcs))
(ka::kc :|kcm_direct_eval| cmd-pkg)
(ka::kc :|kcg_clear_all| kcg)
(setq kto (ka::kc
  :|kcm_type_object_by_qregistered_type_name| "KCM_Env*") )
(ka::kc :|kcm_add_type_binding| "kenv" kto)
(ka::kc :|kcm_kcg_add_fuxe_carrier| kcg kto "envv")  3
(setq kto (ka::kc :|kcm_type_object__str|))
(ka::kc :|kcg_add_lambda_carrier_via_typed_literal|
  kcg kto "KCM_Env*")
(setq kcx (ka::kc :|kcm_dissolve_to_nested_expression| kcg))
(setq kcs (ka::kc
  :|kcm_promote_type_binding_to_statement_with_expression|
  "kenv" kcx))
(setq cmd-pkg (ka::kc :|kcm_statement_to_command_package| kcs))
(ka::kc :|kcm_direct_eval| cmd-pkg)
(ka::kc :|kcg_clear_all| kcg)
(ka::kc :|kcm_kcg_add_fuxe_carrier| kcg kto "init")  4
(ka::kc :|kcg_add_sigma_carrier_via_symbol| kcg "fnd")
(ka::kc :|kcg_add_lambda_carrier_via_scoped_symbol| kcg "kenv")
(setq kcx (ka::kc :|kcm_dissolve_to_nested_expression| kcg))
(setq kcs (ka::kc :|kcm_promote_expression_to_statement| kcx))
(setq cmd-pkg (ka::kc :|kcm_statement_to_command_package| kcs))
(ka::kc :|kcm_direct_eval| cmd-pkg)
(ka::kc :|kcg_clear_all| kcg)
(ka::kc :|kcm_kcg_add_fuxe_carrier| kcg kto "read")
(ka::kc :|kcg_add_sigma_carrier_via_symbol| kcg "fnd")
(setq kto (ka::kc :|kcm_type_object__str|))
(ka::kc :|kcg_add_lambda_carrier_via_typed_literal|
  kcg kto "test-fn")
(setq kcx (ka::kc :|kcm_dissolve_to_nested_expression| kcg))
(setq kcs (ka::kc :|kcm_promote_expression_to_statement| kcx))
(setq cmd-pkg (ka::kc :|kcm_statement_to_command_package| kcs))
(ka::kc :|kcm_direct_eval| cmd-pkg)
(ka::kc :|kcg_clear_all| kcg)  5

;;- _file ...
```

Sample 10: Sample IR Code

Corresponding to this sample, the Intermediate Representation (actually Common Lisp) in Figure 10, to which the above code compiles, shows both channel constructions for signatures (❶) and dynamic calls (❷, ❸, and ❹). These calls vary in the channel formations used — for instance, ❹ uses a sigma channel (and maps to a C++ method via the Qt meta-object system); whereas ❸ uses a predefined C++ function exposed to the scripting engine (in this case one which retrieves a predefined **KCM_Env** object that is globally part of the scripting environment; **envv** stands for "environment value"), and ❷ default-constructs an object of the requested type (again, the Qt code called from this Lisp form will employ Qt meta-object code to actually allocate the object, providing a type identifier based on the type name). These structural differences are reflected in how the C++ callback methods are named (here visible indirectly by the naming convention for some Lisp symbols, which get translated behind the scenes to method names of a certain Qt-based class) — **kcm_promote_type_binding_to_statement_via_type_default_literal__let**, for example, maps to the particular C++ function which the engine uses to default-construct a value given a corresponding type name. This code shows that channel complexes can be constructed for several purposes, including but not limited to dynamic evaluation (when evaluation is desired, it can be triggered by **kcm_direct_eval**, as at ❺).

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

This formal channel/complex/package/payload/carrier vocabulary codifies what are to some degree common-sense frameworks through which programmers reason about computer code. This descriptive framework (I would argue) more effectively integrates the *semantic* and *syntactic* dimensions of source code and program execution (Figures 11 and 12, listed as an addendum the end of this section, sample the syntactic — or grammar/parsing — and semantic/evaluative components of the demo code compiler pipeline, respectively).

Computer programs can be understood *semantically* in terms of λ-Calculi combined with models of computation (call-by-value or by-reference, eager and lazy evaluation, and so forth). These semantic analyses focus on how values change and are passed between functions during the course of a running program. From this perspective, source code is analyzed

in terms of the semantics of the program it describes: what are the semantic patterns and specifications that can be predicted of running programs on the basis of source code in itself? At the same time, source code can also be approached *syntactically*, as well-formed expressions of a formal language. From this perspective, correct source code can be matched against language grammars and, against this background, individual code elements (like tokens, code blocks, expressions, and statements) — and their inter-relationships — may be identified.

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

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

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

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

This chapter has focused on modeling specifications captured entirely via a type system, so at this point I'll return to that topic in the specific context of Dependent Types. Consider first a function which must receive a signed integer no greater than **100**. This is a dependent type in the mild sense that the type expression depends on a value: for the range criterion to be TXL-expressible we must assume that the relevant TXL includes type expressions whose elements can be values which are not themselves types. For instance **ranged_lte** can be a type class such that **ranged_lte<V>** is a type-expression (with **V** a value, not a type), designating types whose inhabitants all compare $\leq$ to **V**. As I have alluded to, we do not have an *a priori* set-theoretic machinery at the implementation level; our ability to express criteria like "the set of unsigned **int**s $\leq$ **100**" depends on an explicit type implementation — and, in particular, a value constructor. Assuming we have generic programming, we can implement **ranged_lte<V>** via a constructor/initializer which takes an arbitrary (hitherto unchecked) **int** and verifies that it falls in the range $\leq$ **V**. This leaves unspecified what to do if the check fails — throw an exception? Silently covert the input to **V**, maybe logging a warning? But that choice can be left to the discretion of the type implementation; the fixed criterion is that any carrier purporting to hold **ranged_lte<V>** values guarantees that any initialized value has been range-checked.

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

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

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

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

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

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

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

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

Another case-study is offered by exceptions. The option to "throw" an exception can be considered an alternative kind of output channel. A function throws an exception instead of returning a value. As a result, **return** and **exception** chan-

nels typically evince a semantic requirement (which earlier — see the notations on page 19 — I sketched as an algebra stipulation): when functions have both kinds of channels, only one may have an initialized carrier after the function returns. Usually, thrown-exception values can only be bound to carriers in **catch(...)** formations — once held in a carrier they can be used normally, but carriers in **exception** channels themselves can only transfer values to other carriers in narrow circumstances (this in turn depends on things like code blocks, which in turn will be reviewed below). So **exception** channels are not a sugared form of ordinary returns, any more than objects are sugar for functions' first parameter; there are axiomatic criteria defining possible formations of **exception** and **return** channels and carriers, criteria which are more transparently rendered by recognizing **exception** and **return** as distinct channels of communication available within function bodies.

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

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

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

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

3.3.2 Statements, Blocks, and Control Flow

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

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

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

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

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

This vocabulary can be useful for interpreting program flow. Assignment statements with no other side effects can be delayed until their grounding carrier is convoluted with some other carrier. Of course, the default choice of "eager" or "lazy" evaluation is programming-language-specific, but for abstract discussion of source code graphs, we have no *a priori* idea of temporality; of a program executing in time. This is not a matter of concurrency — we have no *a priori* idea of procedures running at the *same* time any more than of them running sequentially (cf. "detached" evaluation as on page 17). Any temporal direction through a graph is an interpretation *of* the graph, and as such it is useful to assume that graphs in and of themselves assert no temporal ordering among their nodes or edges. When modeling eager-evaluation languages, particular edge-types can be designated as forcing a temporal order or else edges can be annotated with additional temporalizing details. Without this extra documentation, however, execution order among graph elements can be evaluated based on other criteria.

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

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

The temporalizing elements in this formation are the "hidden lambdas". In a multi-channel paradigm, we can therefore consider "monad-subblocks" with respect to other channels. Consider how individual statements can be typed: like blocks, statements can select from symbols in scope and can potentially result in thrown expressions, so their channel structure is something like **capture** plus **exception**. Even without hidden lambdas, observe that the runtime order of statements can be fixed in situations where an earlier statement can af-

fect the value (via non-constant capture) of a carrier whose value is then used by a later statement. So for languages with a more liberal treatment of side-effects than Haskell, we can interpret chains of statements *in fixed order* as successively capturing (and maybe modifying) symbols which occur in multiple statements. Having discussed convoluted *carriers*, extend this to channels: in particular, say two **capture** channels are convoluted if there is a modifiable carrier in the first which is convoluted with a carrier in the second (this is an ordered relation). One statement must run before a second if their **capture** channels are convoluted, in that order.

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

3.3.3 Code Blocks as Typed Values

Insofar as blocks can be typed as functions (in a sense), they may readily be passed around: so loops, **if…then…else**, and other control flow structures can plausibly be modeled as ordinary function calls. This requires some extra semantic devices: consider the case of **if…then…else** (I'll use this also to designate code sequences with potential "**else if**"s), which has to become an associative array of expressions and functions with "block" type (e.g., with only **capture** and **exception** channels). We need, however, a mechanism to suppress expression evaluation. Recall that expressions are concretized channel-structures which include a **lookup** channel providing the actual implementation to call. Assume that **lookup** can be assigned a flag which suppresses evaluation. Assume also that carriers can be declared which hold (or somehow point to) *expressions* that evaluate to typed values, in lieu of holding these values directly (note that this is by intent orthogonal to a type system: the point is not that carriers can hold values whose type is designed to encapsulate potential computations yielding another type, like **std::future** in C++). Consider again the nested-expression variant of $\mathfrak{C}_1 \not\mapsto \mathfrak{C}_2$: 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 $\mathfrak{C}_1 \not\mapsto \mathfrak{C}_2$ allowing $\mathfrak{C}_1$ to hold a delayed capability to evaluate the f_2 expression. Insofar as this is an option, functions can be given a signature — this would be included in the relevant TXL — where some carriers are of this "delayed" kind. Functions like **if…then…else** can then be declared in terms of these carriers.

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

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

Further discussion of code models via Channel Complexes and Channel Algebras is outside the scope of this chapter, but is demonstrated in greater detail in the accompanying 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 codeset focuses more on practical engineering and problem-solving where Channel models can be useful, and I'll briefly review its structure and its organizing rationales in the Conclusion.

3.4 *Addendum*

The following are more inclusive code samples demonstrating compilation and evaluation at both a syntactic and semantic level. Figure 11 shows a list of "prerules" () in a special parser-definition format for Context-Sensitive grammars, which

translate to Regular Expression code, and full-fledged "rules" which combine Regular Expressions, named captures (❷), contextual tests (❸, ❹, ❺), and C++ callbacks (❻). Figure 12 samples the code which leads up to dynamic method-invocations, and then the actual invocation-points (❶): the latter is wrapped in macros, for the code to work with various numbers of arguments. A key feature of this code is various sets of "conventions" — for input arguments, return values, binary encoding, etc. — which deal with differences in how the Qt Meta-Object System (and generic function pointers) can handle the encoding of disparate data types into a small set of fundamental binary structures, such as **QString**s; **QObject**- and **void***-pointers; and 64-bit unsigned integers.

```
void RE_Grammar::init(RE_Parser& p, RE_Graph& g,
     RE_Graph_Build& graph_build)
{
 pre_rule( "script-word",
   "(?:[^{}()\\[\\]\\s`;,:]|(?:\\w::?\\w))+" );
 pre_rule( "space-to-end-of-line",
   "[__\\t]* (?: ;- [^\\n]+ )? (?= \\n )" );
 pre_rule( "single-space", "[__\\t]" );
 ...

 Context run_context = add_context("run-context");
 Context data_context = add_context("data-context");
 Context lisp_context = add_context("lisp-context");

 track_context({&run_context, &data_context, &lisp_context});

 Context run_or_lisp_context = add_context("run-or-lisp",
   {run_context, lisp_context});

 activate(run_context);

 RE_Parse_Context& parse_context = graph_build.parse_context();

 add_rule(flags_all_(parse_context ,take_a_space),
   run_context, "t      a-space", " .single-space. "
   ,[&]
 {
   graph_build.finalize_take_a_space();
 } );

 add_rule( flags_all_(parse_context ,inside_string_literal),
      run_context,
   "string-literal-character",
   " (?: [^\\\"] | \\\\\" )+ "
 ,[&]
{
 QString str = p.match_text();
 if(str == "\\\"")
   str = "\"";
 graph_build.add_to_string_literal(str);
});

 add_rule( flags_all_(parse_context ,inside_string_literal),
      run_context,
   "string-literal-end",
   " \" "
 ,[&]
{
 graph_build.process_string_literal();
 parse_context.flags.inside_string_literal = false;
});

 add_rule( run_context, "string-literal-start",
   " \" "
 ,[&]
{
 graph_build.string_literal_start();
});
 ...
```

```
 add_rule( run_context, "run-tuple-indicator-with-name",
   " (?<name> \\w* ) "
   " (?<prefix> [;,:_+`'#$%^*~!@\\-\\.]* )"
   " (?<entry> (?: \\{{2,3} ) | "
   " (?: \\[{2,3} ) | [ [({ ] )"
   " (?<suffix> [*~]* (?=\\s) )?"
 ,[&]
{
 QString name = p.matched("name");
 QString prefix = p.matched("prefix");
 QString entry = p.matched("entry");
 QString suffix = p.matched("suffix");
 graph_build.enter_tuple(name, prefix, entry, suffix);
});
 ...

 add_rule( run_context, "run-token-with-eol",
  " (?<prefix> [:,;`']* )"
  " (?<word> .script-word.) "
  " (?<suffix> :? ) "
  " (?<eol> .space-to-end-of-line. \\n)?",
   [&]
 {...}
 ...
}
```

Sample 11: Excerpt from a Hypergraph-Oriented Parser

```
template<int Arg_Count>
void KCM_Command_Runtime_Router::Do_Invoke_Method<Arg_Count>
  ::run(KCM_Command_Runtime_Router* this_,
  QVector<KCM_Command_Runtime_Argument*>& args)
{
 ...
 switch(this_->reflection_convention_)
 {
 case Reflection_Conventions::Qt_Meta_Object:
  Do_Invoke_Method__Cast_Schedule__QOB__Cast_<Arg_Count>
    ::Type::template run<QObject*,
    typename Type_List__All_Cast_Needed<Arg_Count>::Type>
    (this_->fuxe_name(), this_->this_object(), 0,
    *this_, this_->argument_info(), args);
  break;
  ...
 }
}
...
class KCM_Command_Runtime_Router
{
 enum class QOB_Argument_Conventions { ...
 enum class Return_Conventions { ...
 enum class Reflection_Conventions { ...
 enum class Arg_Type_Codes { ...
 struct Argument_Info
 {
  void* void_argument; ...
  const QMetaType* qmt;
  const QMetaObject* qmo; ...
  QOB_Argument_Conventions qob_convention;
  ...
 };

 template<int Arg_Count>
 struct Do_Invoke_Method ...

 void do_invoke_method(
   QVector<KCM_Command_Runtime_Argument*>& args);
 void init_argument_info(QVector<
   KCM_Command_Runtime_Argument*>& args, QVector<quint64>&
        store);
 ...
};

// in "kcm-command-runtime-router-qob.h" ...
#define ARGS_TEMP_MACRO(INDEX) \
 typename Type_List_Type::Type##INDEX& arg##INDEX \
   = *reinterpret_cast<typename Type_List_Type::Type##INDEX*> \
    ( argument_info[INDEX].void_argument );

#define QARG_TEMP_MACRO(INDEX) QArgument<ARG##INDEX##_Type> \
```

```
  ( argument_info[INDEX].type_name_with_modifier( \
  args[INDEX - 1]->qob_reflection_modifier(), \
  args[INDEX - 1]->qob_reflection_type_name()).toLatin1(), arg
       ##INDEX) \

#define CASE_TEMP_MACRO_(INDEX, READY, INTERCHANGE_TYPE) \
 case INDEX: \
  Cast_##READY::run<OBJECT_Type, \
    typename Interchange<Type_List_Type, INDEX, READY>:: \
    template With_Type<INTERCHANGE_TYPE>::Result_Type> \
    (method_name, obj, next_cast_index, kcrr, argument_info,
         args); \
  break; \

#define ARGS_TEMP_MACRO(INDEX) \
 typename Type_List_Type::Type##INDEX& arg##INDEX \
   = *reinterpret_cast<typename Type_List_Type::Type##INDEX*> \
    ( argument_info[INDEX].void_argument );

#define CAST_READY_MACRO(ARG_COUNT) \
 struct Cast_##ARG_COUNT##_Ready { \
  ... \
  template<typename OBJECT_Type, typename RET_Type \
    TYPENAMES_typename> \
  static void run(QString method_name, OBJECT_Type obj, \
   QString return_type, RET_Type& ret, QVector< \
     KCM_Command_Runtime_Argument*>& args TYPENAMES_arg, \
     const QVector<KCM_Command_Runtime_Router:: \
                 Argument_Info>& argument_info) \
  { \
   QMetaObject::invokeMethod(obj, method_name.toLatin1(), \
     QReturnArgument<RET_Type>(return_type.toLatin1(), ret) \
     QARGUMENTS ); \
  } \
 }; \

#define CAST_STRUCT_START_MACRO(ARG_COUNT) \
  struct Cast_##ARG_COUNT {

#define CAST_STRUCT_RUN_MACRO(ARG_COUNT) \
  static constexpr int ready_at_cast_index = ARG_COUNT; \
  template<typename OBJECT_Type, typename Type_List_Type> \
  static void run(QString method_name, OBJECT_Type obj, \
    int cast_index, KCM_Command_Runtime_Router& kcrr, \
    QVector<KCM_Command_Runtime_Router:: \
           Argument_Info>& argument_info, \
    QVector<KCM_Command_Runtime_Argument*>& args) {

#define CAST_READY_SWITCH_MACRO(ARG_COUNT) \
   switch(kcrr.return_type_code()) { \
   case KCM_Command_Runtime_Router::Arg_Type_Codes::No_Return:\
    { \
     ARGS_TEMP_MACROS \
     Cast_##ARG_COUNT##_Ready::run(method_name, obj, args \
       ARGUMENTS, argument_info); \
    } break; \
   case KCM_Command_Runtime_Router::Arg_Type_Codes \
     ::Void_Pointer: \
    { \
     ARGS_TEMP_MACROS \
     Cast_##ARG_COUNT##_Ready::run(method_name, obj, \
       kcrr.return_type_name_strip_namespace(), \
       kcrr.raw_result_ref(), \
       args ARGUMENTS, argument_info); \
     } break; \

#define CAST_SWITCH_MACRO(ARG_COUNT)
   else { \
    int next_cast_index = cast_index + 1; \
    KCM_Command_Runtime_Router::QOB_Argument_Conventions ac = \
      argument_info[next_cast_index].qob_convention; \
    switch(ac) { \
    case KCM_Command_Runtime_Router::QOB_Argument_Conventions \
      ::QString_Direct: \
    case KCM_Command_Runtime_Router::QOB_Argument_Conventions \
      ::Value_From_QString: \
    case KCM_Command_Runtime_Router::QOB_Argument_Conventions \
      ::QObject_Direct: \
     { \
      CAST_INDEX_SWITCH(quint64) } break; \
...
```

```
#define TEMP_MACRO(ARG_COUNT) \
 CAST_READY_MACRO(ARG_COUNT) \
 CAST_STRUCT_START_MACRO(ARG_COUNT) \
 CAST_STRUCT_RUN_MACRO(ARG_COUNT) \
 if(cast_index == ready_at_cast_index) {
      CAST_READY_SWITCH_MACRO(ARG_COUNT)} \
 CAST_SWITCH_MACRO(ARG_COUNT) \
 CAST_STRUCT_END_MACRO

 // for arg count 1
 #define TYPENAMES_typename ,typename ARG1_Type
 #define TYPENAMES_arg ,ARG1_Type arg1
 #define QARGUMENTS ,QARG_TEMP_MACRO(1)
 #define ARGUMENTS ,arg1
 #define ARGS_TEMP_MACROS ARGS_TEMP_MACRO(1)

 #define CASE_TEMP_MACRO(INDEX, INTERCHANGE_TYPE) \
   CASE_TEMP_MACRO_(INDEX, 1, INTERCHANGE_TYPE)

 #define CAST_INDEX_SWITCH(TYPE) \
   switch(cast_index) { CASE_TEMP_MACRO(0, TYPE) }

 /*here, the actual dispatch: */ TEMP_MACRO(1)  1

 #undef TYPENAMES_typename ...
```

Sample 12: Sample Code Graph Evaluator

4 Conclusion

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

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

Manually creating residual values along these lines is admittedly cruder than pure type-level engineering, as exem-

plified by Idris's synthesis of Dependent Types, typestate, and effect-typing. The solutions chosen for the present context are less automated; they require deliberate choice by developers. Nevertheless, the Interface Definition technology makes these choices explicit and documented, which can help guide developers toward embracing these more fine-grained coding conventions and confirming that they have done so consistently.

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

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

Whether or not by technical necessity or just entrenched culture, GUI frameworks are still predominantly built in procedural or OO languages like C, Java, and C++. It seems likely that a truly integrated type system, covering User Interface as well as data management logic, will need a hybrid functional/procedural paradigm at some stratum — either the underlying GUI framework or at application-level code. So long as GUI frameworks remain committedly procedural, the most likely site for such hybrid paradigms to emerge is at the application level, and in the context of application-development SLE tools.

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

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

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

References

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

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

3 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

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

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

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

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

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

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

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

11 Gabriel dos Reis and Jaako Järvi, "What is Generic Programming?". https://pdfs.semanticscholar.org/e730/3991015a041e50c7bdabbe4cb4678531e35b.pdf

12 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

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

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

15 Martin Erwig, "Specifying Type Systems with Multi-Level Order-Sorted Algebra". Information and Computation 207 (2009), pp. 411-437. Third International Conference on Algebraic Methodology and Software, 1993, pp. 177-184. https://web.engr.oregonstate.edu/~erwig/papers/MultiLevelAlgebra_AMAST93.pdf

16 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

17 Sara Irina Fabrikant, "Visualizing Region and Scale in InformationSpaces". Proceedings, The 20th International Cartographic Conference, ICC 2001, Beijing, China, Aug. 6-10, 2001, pp. 2522-2529. https://pdfs.semanticscholar.org/526a/09e4767ff634c4cfbc51e6f7f4ebb700096a.pdf

18 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

19 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

20 Jeremy Gibbons, "Datatype-Generic Programming". https://www.cs.ox.ac.uk/jeremy.gibbons/publications/dgp.pdf

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

22 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

23 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

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

25 Michael Jubien, "Straight Talk about Sets". *Philosophical Topics* 17 (2), pp. 91-107 (1989)

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

27 Sunil Kathari and Martin Sulzmann, "C++ templates/traits versus Haskell type classes". http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.78.2151&rep=rep1&type=pdf

28 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

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

30 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

31 Haishan Liu, *et. al.*, "Mining Biomedical Data using RDF Hypergraphs". 2013 12th International Conference on Machine Learning and Applications http://ix.cs.uoregon.edu/~dou/research/papers/icmla13_hypergraph.pdf

32 Giuseppe Longo and Eugenio Moggi, "A Category-Theoretic Characterization of Functional Completeness". Theoretical Computer Science, 70 (2), 1990, pp.193-211. https://pdfs.semanticscholar.org/58a7/62acf822fd3520eeb1576b7758351d05c34d.pdf

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

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

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

36 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

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

38 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

39 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

40 William J. Rapaport, "Semiotic Systems, Computers, and the Mind: How cognition could be computing". International Journal of Signs and Semiotic Systems, 2(1), 32-71, January-June 2012. https://cse.buffalo.edu/~rapaport/Papers/Semiotic_Systems,_Computers,_and_the_Mind.pdf

41 Yuriy Solodkyy, *et. al.*, "Open and Efficient Type Switch for C++". http://www.stroustrup.com/OOPSLA-typeswitch-draft.pdf

42 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

43 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

44 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

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

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

47 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

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

49 Eric Walkingshaw, "The Choice Calculus: A Formal Language of Variation". Dissertation, Oregon State University, 2013 http://web.engr.oregonstate.edu/~walkiner/papers/thesis-choice-calculus.pdf

50 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

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

52 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

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

54 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