Perspector Algebra and the λ4

Nathaniel Christen
2015

ABSTRACT

Proposed in 2006 by Michel Biezunski, Perspector Algebra is a variation on Semantic (or Semantic Web) models which is especially well-suited for integrating heterogeneous data sources. In this paper, I argue for the elegance of Biezunski's "Data Projection Model" as applied to software development tasks, and especially as applied to code analysis and annotation, in conjunction with data integration. I will focus on potential applications of the Perspector concept to User Interface Design and to Cybersecurity analysis. I will also briefly explore techniques for merging Perspector Algebra with other formal systems, such as Kleene Algebra, Lambda Calculus, and Category Theory, showing how Lambda Calculus and Semantic Web can be bridged by a special "λ4" Ontology.

Because markup languages like XML, HTML, and LaTeX are intended to be mostly neutral carriers of structured information — modeling the semantics of the data they encode — it is easy to overlook that these (like any) languages also have their own semantics. There is, for example, a semantic model of an XML document — with central concepts such as tag names, attributes, text nods, and the Document Object Model. There is a semantics for any language or structure through which data is visualized — whether via markup, like HTML; application Graphical User Interfaces, built in languages like C++ and Java; or graphical displays of data sets, via charts, diagrams, simulations, three-dimensional scenes, and Virtual Reality. There is a Semantics of relational and "NoSQL" databases; of Object-Relation Mapping for marshalling data between visual and database points; in effect, there is a Semantics of all components which collectively define a software application, a computational ecosystem, or a framework for building these. The "Semantics" of structured data therefore resides within a mesh of "ambient" Semantics latent in all the tools through which that data is processed.

Taking *ambient* Semantics seriously presents both challenges and opportunity for software theory and practice. The *challenge* is that *application* semantics does not rest on some prior foundation, like Natural Language vis-à-vis human cognition, and/or a general semiotics which can rest semantics upon the production, recognition, and exchange of signs in general, as a capability of machines and living things at all levels of complexity. There is of course a semantics of linguistic and semiotic research — a terminology of grammatic categories, terms for signifying units (*lexeme*, *morpheme*) or aspects of signs (*signifier*, *signified*, *referent*, *icon*, *iterpretant*). But this is a reflective, second-order observation of how signs and language work, so to build a scholarly discipline; there is not a need to *build* languages; to stabilize a terminology *of* semantics so that language-like systems can be engineered. By contrast, computer languages are artifacts; they do not evolve organically from nature and so do not inherit the *a priori* latency of sign-participation. It needs to be formally defined what a language like XML or C++ *is*; what makes a system an *implementation* of such languages; which demands some enveloping semantics to describe the rules and criteria of their well-formedness. What can be overlooked is that this enveloping semantics is still Semantic and, in turn, has its own enveloping Semantics, and, potentially, so on. The *potential* of this realization, once we accept the challenge of thinking through this refocus toward Semantics as ambient and not language-specific, is that we can better reason through the reality of language interpenetration. In the modern computing world, languages rarely exist in isolation. From the C++ application which uses XML configuration files, to the Java component generating SQL queries, to the JavaScript code manipulating HTML and CSS, an essential feature of modern formal languages is that intended effects often can only be achieved by processing or emitting code in other languages.

The reality of "ambient" Semantics also calls into question the intuitive contrast of semantics and syntax. These notions are distinct insofar as the syntactic correctness of an expression in some language does not, in itself, point towards its semantic meaning. But it is impossible to define criteria of correctness without appealing to concepts in an enveloping Semantics. To describe XML, for example, we need at hand notions of *documents*, *nodes*, *tags*, *attributes*, *characters*, *text*, *entities*, and *references*. These concepts provide the orbit within which a grammar can be defined: a *tag* begins with a less-than *character* and has a *name*, possibly with a *namespace*, with characters drawn from a restricted character *class*. One consequence of this formalization is that whenever a lexical aggregate — which we can call a *leximorph*, by analogue to how a non-aggregate lexical unit is a *lexeme* — can be validated according to the grammar of a language, there is already a provisional semantic model available, drawing on the "semantics of the grammar". If, say, a sequence of characters is parsed as an XML tag, then there is a minimal semantic assertion warranted, that *these* characters are parsed as *this* language element. Against this background, formal semantics can be seen as the process of enriching these "provisional" semantic assertions toward a "higher level" semantics of a language, and a data space.

The study of ambient Semantics demands that we consider the most provisional semantic layers as a basis for higher levels: to define the minimal units of any formal semantic system whatsoever. The rise of Semantic Web has suggested that the proper model of these minimal units is a three-part binary operator or *Subject-Predicate-Object* complex, which I will call a *trope*. A trope asserts that some relation exists, or some operation is modeled, between an ordered pair of elements (the inverse trope, with the order reversed, may or may not be semantically equivalent). Larger-scale aggregates can always be decomposed into collections of such tropes. In many approaches, for example, the minimal aggregate of data might be construed as a "record" or "object", a collection of named values: given some object, we can compile all facts known or relevant into an associative array. A contact in an address book for example might have a first name, last name, email address, phone number, and so forth. But this structure is itself a union of distinct triples: contact C has *this* first name, etc. Similarly, n-ary operations, or data collections (say, a list of different email addresses), can be decomposed into sequential tropes: an association between the original object and the *first* address in a list; which in turn is associated with the *next* item, and on through the list. Of course, these assertions are specific to contexts where they occur: the fact that two names are adjacent in an alphabetized address book is a relation between them which does not hold in general, but does hold in the specific context of that list. In other words, *tropes*, in contrast to general Semantic Web *triples*, exist in contexts that influence their meaning. Tropes, for instance, may be asserted in *lists* of tropes, semantically and/or syntactically depending on which tropes precede or follow them.

The Semantic Web can be described as first and foremost an intellectual program to marshal the observation that "tropes" or "triples" are minimal semantic units, into a "Universal" representation system. But this decomposition only captures one dimension: how units aggregate into complexes is as crucial to semantics as the meaning of tropes themselves. Water is determined by the nature of water molecules, but the properties of water are not properties of water molecules. The idea of *emergence*, which has a long history in the Philosophy of Science, is a useful picture for formal semantics: the propensities of water as *substance* are emergent properties of water as collection of water molecules. Analogously, the goal of formal semantics is to study, and normatively enforce, the emergent properties of large collections of semantic tropes. This demands reasoning about tropes themselves alongside reasoning about their aggretive patterns.

In this paper I will propose a framework based on Michel Biezunski's *Perspector Algebra* [3], which I believe is an elegant approach to representing this tropicality and emergence. Perspector Algebra can be seen as a generalization of Semantic Web technologies, such as Resource Description Framework (RDF) — in effect, any RDF representation is straightforwardly an instance of a Perspector Algebra. But there are some rather open-ended philosophical questions concerning RDF which we can address in systematic ways, I believe, through a Perspector framework. One obvious question is how we actually define a *resource*. According to the concept of RDF, information about resources can be provided through "tropes" or "triples", where the resource in question is a "subject" and various facts asserted about its relation to other resources. A canonical example is that a web page is a resource, whose author is represented by a different resource via an `author-of` relation. The open question here is that every sign in these assertions needs to fit the criteria of a "resource", perhaps extended by some universe of "literals", which would include things like character strings and numeric values. In the "official" semantics of RDF, it is assumed that everything is either a literal or a resource, meaning that there must be some canonical Universe of resources — which, insofar as RDF provides the foundation for the Semantic Web, effectively becomes the universe of Uniform Resource Identifiers (URIs). This ends up confusing an implementation *of* a Semantic Network with its general Semantics; but it also raises vexing questions: considering that URIs are universal and uniform only relative to the concrete (and imperfect) technology of the World Wide Web, what is the Semantic significance of broken links, of resources duplicated across multiple servers, and related implementational artifacts? What is the relation between units of meaning as humans understand them — providing the Semantics for computer tools insofar as they are part of everyday life — with the digital processes of curating and sustaining internet archives?

To put it differently, reifying web addressability as the signature criterion of "resourceness" is a dubious theoretical and practical assumption. It is possible to reason about RDF in a more abstract way, bracketing the technological, political, and economic complications of Semantic Web; but we can also consider RDF and Semantic Web resources as one implementation of a more general concept of Semantic Networks, for which Perspector Algebra can provide a conceptual framework. This is the approach I will take here.

1. PERSPECTORS AND TROPE-SPACES

In Biezunski's original proposal, "tropes", or representations analogous to Semantic Web "triples", are defined as binary operations whose components are x and y *operands* and an *operator* (`o`). Biezunski uses a notation inspired by Paul Dirac's "bra-ket" notation for Quantum Mechanics, so a single assertion can take the form **`<x |o| y>`**, where the `o` operator may actually be a relation asserted between x and y (e.g., `parent-of`, between an adult and child), or may be the assertion of an operation to be performed (but not the result of the operation). So for example `o` may represent mathematical operations like addition or division, but without the presumption that the resulting triple is mathematically valid. The fact that a division by zero is (in most number systems) impossible represents a semantic property of a triple like **`<`**1 **`|divide|`** 0**`>`**, but this property can be described via the triple, which has a semantic *meaning* despite its numerical impossibility. Biezunski calls indidivual triples *Perspectors*, to convey the idea that Perspector Algebra promotes Data Integration by "projecting" heterogeneous data sources onto a common representation. I will use the term *trope* to mean a Perspector which has a given *context of assertion* (or context of *query*, if the goal of the trope is not to assert a fact but to describe a pattern for finding matching facts). This context may be implicit or may be explicitly described via a new trope in which the original trope is reified as an x-operator, or by some mechanism or listing all tropes asserted, assertable, or known in some context.

Because complex data structures can be decomposed into sets of Perspectors, a single Perspector may exist in a context in which its relation to other Perspectors reflects the higher-scale structure of the sources from which it is extracted. Biezunski invites us to envision these different kinds of higher-scale structures as "perspectives" on the underlying data-space, which motivates the term "Perspector". This creates an intuitive picture of data integration as well, because we can envision data from multiple sources — with different forms of high-scale structure — being dissembled into individual tropes, with new semantic associations becoming available and the potential of new perspectives being formed. Suppose, for example, that information is extracted from HTML documents by first extracting tropes, which will include both the visual content of the document and underlying markup details. Both content and markup can then be associated with other semantic units whose properties are asserted in other formats. Particular segments of text — perhaps isolated as the textual data associated with a given HTML tag, like an **`<a>`** hyperlink — can be matched with data in databases, or other formats (like XML), or application objects. Alongside the HTML-specific relations (modeled as `o` operators in Perspectors), relations such as that between a tag and its text data, there are then new associations asserted via other relations (those pertaining to a database, Object

Oriented data, other kinds of markup), so that the space of Perspectors becomes richer, more densely interconnected, with greater breadth in the kinds of relations recognized. The resulting space can then be visualized, so that this richer information becomes accessible, by converting back to HTML, or creating other views or perspectives entirely. For example, by taking HTML-specific relations and replicating them with LaTeX-specific tropes, an HTML document can be "cross-compiled" to LaTeX and then viewed as a PDF file. These provide an example of different "Perspectives" on a single (albeit integrated) data space.

This example represents both a motivation and a potential application for the idea of triples as "Perspectors"; but it also points to differences, even if conceptual more than formal, from RDF and Semantic Web. Implicitly, for example, a "Perspector Space" is considered to be all relations which can be extracted from "input" sources, even those which would seem to be "background" relations which are not directly part if high-level semantics. If, for example, we assert relations about a contact in an address book (`first-name`, `email`, etc.) via XML, then this very representation introduces new relations (like tag-to-text) which seem *incidental* relative to the actual Semantics (the real-world "contact in an address book" semantics) at issue. But, as I argued above, such "ambient" Semantics is still semantics and can be relevant in some areas (like an application which seeks to automatically populate a GUI via XML data). A crucial assumption or capability of Perspector Algebra is therefore that Perspector "spaces" are not only *annotated* facts, or "meta" data, which add semantic detail to an underlying representation (via XML, HTML, etc.); the Perspector Space is rather a neutral projection of all relations, both ambient and high-level, which are embodied in some input sources. The "ambient" relations then play a role in reconstructing or cross-compiling these "inputs" so as to recover original perspectives, or create new ones. The semantics asserted by a document, a collection of "leximorphs", which conforms to a formal language (like XML), is embedded along with a semantics latent in the document merely by being a well-formed expression in the language. In this sense, constructing Perspector Spaces out of input documents is an alternative way of parsing them so as to record their underlying "grammatical" semantics as well as their explicit "intended" semantics. From this perspective, Perspector Algebra represents an approach to formal *grammar* as well as formal *semantics*.

At the same time, the high-level "intended" semantics of documents raises the same questions as raised by formal semantics of any kind: how exactly are constructs of a formal language to be linked with things in the actual world? This question is side-stepped for RDF because of the simplifying assumption that "resources" are just whatever is pointed to by URIs. We can observe that URIs are themselves just a special sort of *value*, alongside literals for, say, strings and numbers; so the more general unit of meaning is a "value", taking this very generally. That is, the x and y operands of a perspector are "values", but the question then needs to be addressed of what values *are*, what exactly structures the universe of values. I will consider these questions below; for now let me just make the simplifying assumption that values are *typed*, so that the minimal units of meaning are *typed* values, and that each type has its own criterion of identity and extent: that is, the universe of values is constrained by the range of what constitutes a value for a given type. This transfers the question of what are values to the question of what are *types*, but at least we can introduce type-theoretic notions, such as union and product types. Moreover, we can draw on an existing body of literature which models types via Category Theory. So we an assert a category $\mathcal{T}$ of types, and on that basis assert a category $\mathcal{P}$ of Perspectors, based on the types of their operands. Following the conventions of Conceptual Graphs, I will consider relations (or $\mathfrak{o}$ operators) as elements of partially ordered sets, rather than as typed values; however, there is a straightforward Categorical treatment of posets as well. So, at least for a preliminary "formal" semantics, we can construct a category $\mathcal{P}$ and three functors for x and y, taking operands to typed values, and for $\mathfrak{o}$, taking operators to the category of lattices. In this paper I will focus on Perspectors in specific contexts, such as ordered lists; or, in my terms, on *tropes* and *trope spaces*, which are specific kinds of Perspector Space with precise notions of context, rather than just sets of Perspectors. For example, an *ordered* trope space is an ordered list of tropes; this is the structure I will emphasize in a moderately formal Categorical treatment below.

Insofar as spaces of Perspectors are "imported" from documents, all values are encoded in some lexical form, so we can associate values with sequences of characters in some alphabet, which at least provides a minimal semantics (where the "meaning" of a value is its string representation). Obviously this is an inadequate semantics, but it expresses the very first semantic identification during parsing: some character sequence is identified as meeting the criteria for some type (or set of types); for example, a character string starting with a nonzero numeric digit may be parsed as a positive base-ten number. Such grammatic structures need to be joined with a type system to extend this primordial semantics. But this gives an outline of the unfolding process of building "semantic" Perspector spaces out of input sources, at least insofar as these are processed as raw lexical input. Examining this process in greater detail will link Perspector Algebra to both formal grammars and type theory.

1.1 Perspector Algebra and Formal Grammar

Here I assume that Perspectors are derived from parsing lexical sources. As such, each Perspector exists in a lexical context, which serves as its "context of assertion", and each Perspector is a *trope* in that context. A *parser* in this sense then converts lexical sources into a Trope Space, in which tropes assert syntactic relations between lexical units.

Let us assume a parsing environment which can be modeled via Kleene Algebra (or, more or less equivalently, concretely implemented in terms of Regular Expressions). That is, a grammar is a collection of *rules* each of which is defined using Kleene operators (such as matching one out of a set of characters, and repeating or not repeating the match at the previous position). As implemented, a parser associates rules with *productions*, or actions performed upon a match being found within lexical sources. In the direct joining of Kleene and Perspector Algebras, we can represent such productions as tropes: that is, a parser is provided with a set of rules and a set of associated actions, where each action creates or modifies some trope. For a concrete example, suppose a primitive XML parser has a rule like the following (using the notation for a Perl-compatible Regex engine with the "extended" syntax which ignores whitespace in the Regex code itself; and ignoring real-world issues like name-spaces and different character sets): `< (?<tag-name> \w+ )` . A match for this rule would be a string like `<img`; here the "tag-name" match field would hold the string "img". The associated action for this match should define a new HTML node for the `<img>` tag, as well as enter a state corresponding to being within the head section of a tag (the tag may be immediately closed, but may also include attributes; for example, a useful `<img>` tag would need a "src" attribute). The new tag will be a child element of some parent node, so the newly "asserted" trope produced by this match should be of a form like `<`n_1 `|firstChild|` n_2`>`, where n_i are the two nodes. In addition to producing this trope, the match should trigger the parser's internal state to change, which will alter the "context of assertion" for subsequent tropes as well as which rules are active — for example, a rule which matches attribute declarations of the form `key='val'` (or the equivalent with double-quotes), that would be parsed as ordinary outside of a tag. Here I implicitly assume the use of context-sensitive (rather than context-free) grammars; at the same time, Perspector Algebra can help ensure that context is used rigorously. The parse context influences what tropes will be produced by a match: an attribute match, for example, produces an attribute node whose relation to a parent tag node is different from text or a nested tag. That is, a parse context can be defined by the set of rules which are active when the parser is in that state, along with the **o** operators which are used by the resulting trope productions. Perspector Algebra therefore provides a tool for systematically defining context-sensitive grammars. Moreover, this applications marshals the "algebraic" aspects of perspectors.

Specifically, I have thus far been considering tropes with concrete, specific x, y, and **o** components. However, Biezunski's Algebra includes a lattice of operators in which one or more of these components is "abstracted", or unspecified. In a context of assertion, we can call these *trope patterns*; a specific trope may *match* or *conform to* a pattern. Insofar as we have a set of "givens" which a trope-pattern "queries", Perspector Algebra provides in effect a "query language" technically similar to SPARQL. In a more Kleene-algebraic setting, trope patterns provide representations of parse contexts. Consider the state of the parser upon matching the `<img` opening tag. Assuming the production system I outlined above, the parser has asserted a trope introducing the `<img>` tag into the Document Object Model and, moreover, subsequent tropes will be asserted with this node as x-operand. Moreover, although the string "img" is not sufficient to identify the node overall — there may be multiple image tags in the document — the tag name is sufficient identification for the immediate context (this is why not all tags in an XML document have identifiers). Each tag-node in an XML document provides an implicit scope such that nested or sibling tags with that parent can be identified just through their tag name. This applies to HTML as well: it suffices to write `<img` to introduce a new `<img>`, which is uniquely referenced by its position in the DOM. Using bra-ket notation, this parse context can be identified as `<img|`, meaning that an "img" tag is the "active" x-operand. Neither the **o** or y is known at this particular parse stage, although assuming the parser is working on valid HTML only a handful of **o** relations are possible at this point (connecting the `<img>` to child, sibling, or attribute nodes). However, allowing for other relations to be asserted at this point also allows for HTML extensions or various sorts of preprocessors. For example, an `<img>` may be linked to information about an algorithm which generates an image file to serve as its source. More generally, modeling grammars in terms of Perspectors allows for a straightforward extension of grammars to represent situations like preprocessors and code generators: consider modeling the grammar of HTML variants which include nested computer code, such as ERB ("embedded Ruby") or JSP (Java Server Pages).

1.2 Perspectors and Static Code Analaysis

These HTML examples represent Perspector-based grammars applied to a markup language, but similar principles govern grammars for general-purpose programming languages. Using Perspectors in this setting allows for variations in compiler strategies, but also allows for general code analysis. Trope Spaces offer a global intermediate representation for computer code, which can potentially be used as part of a compiler pipeline, assuming a collection of relations (**o** operators) is designed to capture the semantics of a given programming language. For example, a simple application of a unary function to a single argument ($f{\cdot}arg$) represents one kind of trope, a relation between the function (f) and its argument (arg). More generally, an n-ary function relates the function itself to a list of arguments, the list in turn able to be represented as a first argument followed by a sequence of tropes asserting order. Here the encoding of lists and function-calls recalls the Lisp programming language, where complex data structures are built from minimal units, or *cons* sells. Perspector tropes can be associated with Cons cells, which in turn are just associated pairs of values, corresponding to x and y operands; the **o** connecting them, for Cons cells, being fixed by context. Within a *list* structure, each Cons pairs a value with a reference to a different Cons cell, so the list itself is formed from the sequence of first, or "car" values in the cells. A *proper* list, which is terminating and non-circulating, has a final "null" reference, indicating the end of the list. Lisp code can be readily projected into Trope spaces; a useful encoding is to choose **o** operators so to distinguish between initial and subsequent elements in a list. Consider the modeling of a binary operation like $(+xy)$ (an encoding which can be contrasted with Biezunski's analogous model, [3, p. 6]). Here there is a relation which we can term "Call Sequence" (or cs for short) and two tropes: a call sequence `<+ |cs| x>` and `<x |cs| y>`. The same operation can also be modeled as a single trope whose operator is + (as in `<x |+| y>`), but the double-trope representation is more flexible, because it allows for arbitrary functions and arbitrary numbers of arguments.

Next consider a nested operation (say, $(+x(+yz))$), which can be modeled via nested Perspectors; however, it can also be modeled by varying the trope-operator. In place of a Call Sequence, we can introduce a "call entry" (say, ce) operator so that the tropes in $(+x(+yz))$ become an ordered list of tropes `<+ |cs| x>`, `<x |ce| +>`, `<+ |cs| y>`, and `<y |cs| z>`. To represent arbitrarily complex nested function calls, it is actually necessary to define four different call operators, because a nested call entry, if it is not the final argument to the enclosing function, may then be followed by an atomic value or by another nested call. This can also be represented (as in Lisp proper) by making a type distinction between values and lists (the latter including function calls); however, modeling call structures with operators alone allows us to theoretically dissociate the preliminary parsing of computer code from the semantic type-identification. In short, a Trope Space encoding, using a minimum of four operators, can represent arbitrary complex aggregates of nested function calls, which in lambda calculus are called *applicative structures*. That is, as I will demonstrate in the next section, any model of Lambda Calculus can be projected onto Trope Spaces provided relations are identified to provide the four fundamental operators of applicative grammar.

Models of modern programming languages are much more complex than these elementary lambda structures; function calls themselves, for example, may involve *objects* which are distinct from ordinary arguments; exceptions; default values; function overloads, and so forth. Variations on Lambda Calculus have been developed to formally explore these richer structures; in Trope modeling, it is only necessary to introduce different relation-types, essentially creating an Ontology of source code relations. For example, the relation between an object and a method called on it, is distinct from the relation between a function and its explicit arguments. Moreover, any typed language represents not only the function-call relation between values but also their types. So the representation of $(+xy)$ needs extra tropes to identify the types of the operands, because the + operation has different semantic representations depending on these types. The call-structure operators also can distinguish whether the operation demands a typecast and whether the cast is "faithful", or if a value may be altered for the cast (for example, passing a floating-point value to an integer operation will cause the value used for the operation to be an integral approximation of the original value). With these more complex structures identified, representing source code in terms of Trope Spaces allows for static code analysis in which individual tropes are scanned for particular patterns — to search for instances of particular functions, or any function which can throw exceptions, or functions which require non-faithful type casts, to mention analyses, as examples, which are relevant to finding potential security flaws.

The practical development of Semantic Web has shown that triple-based representations can replace specifications like Database Schemas, XML Document Type Declarations, or service-oriented architectures, being more flexible and suitable for integrating more disparate data sources. However, these applications still tend to involve querying explicit data structures, rather than more fine-grained analysis of, say, markup and programming language code. Compiling code to "trope" spaces allows Semantic query mechanisms to be used for problem-spaces related to static code analysis, cross-compilation, translation between markup formats, and generating visualizations or GUIs. In the next section I will consider how this low-level analysis integrates with higher-level data semantics.

2. TROPES, TYPES, AND FUNCTIONS

Having argued that Trope Spaces can encode general function calls — plus more complex coding constructs, like exceptions and object methods — in this section I will explore the relation between this encoding and general semantic typing. The majority of computer source code is concerned with defining and/or calling functions; most content which does not directly fit into this aspect involves defining types and associating them with values. That is, most or all of computer code's formal semantics is captured by the combination of modeling functions and types. There are several ways to formally study types, but here I will assume a Categorical treatment which recognizes a category $\mathcal{T}$ of types. This can be defined in different ways, so I will not assume *a* single $\mathcal{T}$ category, although below I will provide one possible definition. Following the work of David Spivak [9], we can generalize $\mathcal{T}$ to a space of typed values, for which types provide a fibration. To analyze functional definitions and expressions operating on these typed values, we can define a category $\mathcal{A}_{\|\mathcal{T}}$ of *applicative structures* over $\mathcal{T}$. An applicative structure meets *syntactic* criteria for being a function call, although it may not be a *valid* call, depending on whether an applicative structure α has a valid functional value to call, with the correct number and types of arguments. In lambda calculus, an α is *reducible* if it is semantically valid; however, the set of α within elements of $\mathcal{A}_{\|\mathcal{T}}$ will include non-reducible (that is, semantically invalid) notations. In practice, even invalid "attempts at" function calls will yield some result, perhaps by throwing an exception.

My earlier outline of encoding function calls in a Trope Space can be made precise by considering the category $\mathcal{F}$ of tropes whose operands are values with types in $\mathcal{T}$. Assume a category $\mathcal{L}$ of finite lattices, or taxonomies, to provide "Ontologies" from which $\mathfrak{o}$ operators in $\mathcal{F}$-tropes are selected. I claim there is a simple Ontology with four relations, which I'll call λ_4, such that applicative structures can be modeled as Trope Spaces with tropes of the form **<x |λ_4| y>**, meaning their $\mathfrak{o}$ are drawn from λ_4. The category $\mathcal{F}$ is defined via x and y functors into $\mathcal{T}$ and $\mathfrak{o}$ functors into $\mathcal{L}$; consider the preimage $\mathcal{F}_{/\lambda_4}$ of tropes restricted to λ_4 operators. Define $\mathring{\mathcal{F}}$ as the category of *ordered* trope-spaces over $\mathcal{F}$, and $\mathring{\mathcal{F}}_{/\lambda_4}$ its restriction to λ_4 operators. The main technical claim of this paper is then:

Theorem 1. *The category $\mathcal{A}_{\|\mathcal{T}}$ can be embedded in, or equivalently is isomorphic to a subset of, $\mathring{\mathcal{F}}_{/\lambda_4}$.*

Proof. By induction on the construction of an arbitrary applicative structure α. There are two cases to consider:

1. *α is an atomic value.* Assume there is a type in $\mathcal{T}$ with a single value $\mathfrak{r}$ which represents the "root" or initial state of a computation. Consider the "call sequence" **cs** operator in λ_4 (which I will formally define below). By convention, assume that tropes of the form **<$\mathfrak{r}$ |cs| $\mathfrak{v}$>** encode any atomic value $\mathfrak{v}$.

2. *α is notationally a function call.* Assume that any applicative structure contains, at its head, an atomic value which is "functional" (to be defined later) if it is semantically valid. This involves little loss of generality because an α which includes some nested function call to derive a function value can be re-expressed in terms of a function like *apply*, as in Lisp, which evaluates its first argument to yield a functional value then applied to its remaining arguments. So any non-atomic

α is an atomic value followed in sequence by other applicative structures, say, $\{\alpha_1 ... \alpha_n\}$. Assume each α_i is either atomic or encoded as a list of tropes, the first of which has an atomic value (a functional value if α is semantically valid) as x operand. To prove such encoding extends to α, we need to show that that the disparate lists of tropes can be united into a single list. Assume we are able to do so for $\{\alpha_1 ... \alpha_{n-1}\}$, and want to extend this list to include the encoding for α_n. Let h_{n-1} and h_n be the "head" values for α_{n-1} and α_n respectively. The encoding must distinguish between four cases, which can be done by distinguishing among the four operators in λ_4:

- Assume a "Call Sequence" relation `cs` which applies if α_{n-1} and α_n are both atomic.
- Assume a "Call Entry" relation `ce` which applies if α_{n-1} is atomic and α_n is non-atomic.
- Assume a "Cross Sequence" relation `cx` which applies if α_{n-1} and α_n are both non-atomic.
- Assume a "Call Continuation" relation `cc` which applies if α_{n-1} is non-atomic and α_n is atomic.

By selecting `o` from λ_4 based on these criteria, define a trope t = <h_{n-1} |o| h_n> and construct a list of tropes with the list from $\{\alpha_1 ... \alpha_{n-1}\}$, then t, then the list from α_n. To show that the resulting list uniquely determines an applicative structure, it is necessary to prove that this t is uniquely located in the list. If α_{n-1} is atomic, then t is the only trope with h_{n-1} as x-operand. Otherwise, t's `o` is either `cx` or `cc`. In the encoding of α_{n-1} on its own, h_{n-1} is atomic, so the first and only trope with h_{n-1} as x-operand has operator `o` as either `cs` or `ce`. In either case t is uniquely located as either the sole trope matching the pattern <h_{n-1}|c> for c either `cx` or `cc`, or, if no such trope exists, the sole trope matching the same pattern for c being `cs` or `ce`.

□

This argument considers only the simplest kind of applicative structure, corresponding to "pure" functions, in the terminology of functional programming languages. In these cases a function has no means of altering the state of values outside its immediate function body except by passing a return value, which is presumably bound to a variable. In real life, many or most functions are not "pure", and functions can alter exterior values in numerous ways, including via return values, mutable reference parameters, globally accessible mutable values, lexical variables which are available within the function body as a "closure", and throwing exceptions. For detailed analysis, it is necessary to identify all such possible connections. That is, we can endeavor to build a space of tropes with the form <f |o| v>, where f is a function-call, v is an instance of a value (a symbol or variable holding a value), and `o` is a relation specific to the *mechanism* by which f has this effect. Thus, we can extend the λ_4 Ontology into a list of relations pertaining to function-calls and their side-effects, which we can call *channels*. It then becomes necessary to define functional "effects" more rigorously.

Given a universe of typed values, we can assume a collection of *lexically scoped symbols* which are associated with a type and which hold values of that type (possibly allowing also for certain "special" values, such as a "null" value meaning a value which is not yet known or initialized). A symbols is uniquely identified by a character string within a given scope, but symbols in different scopes (including nested ones) may be different symbols while sharing the same "name". I will assume that symbols do not "change type", although their actual values can change between more specific types, and functions can be called on them to obtain their values as cast to a different type. Consequently, any change in state for a symbol is to reassign it to a different value of the same type. Technically, each such effect is an endomorphism in $\mathcal{T}$. Given a trope <f |o| v>, where `o` is an operator identifying a type of functional effect, we can associate f with an endomorphism on v's type. Since f can operate on multiple values, we can associate f with a collection of endomorphisms, capturing the totality of effects which f can have on "program state", or on the sum total of all symbols involved in program execution, including copies of lexical scopes as distinct "stack frames" during recursive function calls. The set of these effects, as well as the arguments which a function must (or may) take, can be modeled by its *signature*, generalizing this notion to functions "with side effects". That is, generalizing from the specific effects influenced by a given function call on actual values, the *types* of these values, and the manner by which this effect occurs (mutable parameters, return values, exceptions, globals, closures) is a general function "signature". Moreover, signatures provide a means of assigning types to functions. In the simple case of a "pure" function, the type is a map from a typed tuple of values to the return type; in this more general notion of signatures, we can still associate function values with types which are complex aggregate of other types, representing types of arguments to functions and/or of values which they can alter.

By this construction, functions themselves are just typed values, so they are contained within $\mathcal{T}$. This also suggests some ideas about how $\mathcal{T}$ itself can be defined. In particular, $\mathcal{T}$ must be closed under type-theoretic operations (like unions), but also under these complex functional operations. Given an ontology which includes a classification of functional side-effects, $\mathcal{T}$ is closed under signatures defined via this ontology. To be more precise, let $\mathscr{E}$ be a taxonomy specifically of this "list of effects", and assume a signature can list a sequence of affected values, as well as parameters. Much as function calls themselves are modeled as lists of tropes around λ_4 ("around" meaning drawing `o` from λ_4), where the position of the trope in the list is significant, similarly function signatures are modeled as lists of tropes around $\mathscr{E}$. If $\mathring{\mathcal{T}}_{\mathscr{E}}$ is the category of ordered trope-spaces around $\mathscr{E}$, then we can define a category $\mathscr{F}$ of function values embedded in $\mathring{\mathcal{T}}_{\mathscr{E}}$(or more specifically $\mathscr{F}_{\mathcal{T}}$, emphasizing that the operands around $\mathscr{E}$ operators are typed within $\mathcal{T}$). This is similar to how the category of applicative structures (for which $\mathscr{F}$ provides the domain of head values for semantically valid, non-atomic α) is (by the theorem) embeddable in $\mathring{\mathcal{T}}_{/\lambda_4}$. Moreover $\mathscr{F}_{\mathcal{T}}$ is contained within $\mathcal{T}$, so $\mathcal{T}$ is closed with respect to extension via $\mathscr{F}_{\mathcal{T}}$.

To further specify $\mathcal{T}$, we can appeal to Spivak's analysis, which is presented in conjunction with an abstract theory of "database schema", but in a sufficiently general way that "scema" can effectively be considered as general types. This requires that $\mathcal{T}$ be closed under taking named associative tuples; that it, out of a tuple of types and a tuple identifiers, we can define a new type whose values include values from the corresponding types as "fields", accessed by the corresponding identifiers. In Spivak's theory, morphisms in $\mathcal{T}$ are precisely fields accessed through typed values: given $\mathfrak{v}$ of type $\mathfrak{t}_1$, a field f provides a morphism to the type associated with f. Spivak's demonstration that a universe of typed values satisfied conditions of a "Grothendieck" construction on $\mathcal{T}$ (in his paper, an analogous category of schema), essentially formalizes our intuition that an abstract specification of an aggregate type, in terms of the names and types of its fields, can be concretely instantiated in a value whose fields map to other values with appropriate types. That is, the following diagram commutes (where $\pi : \int(\mathcal{T}) \longrightarrow \mathcal{T}$ takes typed values to their types — cf. Spivak, p. 14 — ; $\mathfrak{f}_I$ is a field instance; and $\mathfrak{f}_S$ is a field schema; which is to say, its declared type):

$$\begin{tikzcd} \mathfrak{v}_1 \arrow[r, "\mathfrak{f}_I"] \arrow[d, "\pi"] & \mathfrak{v}_2 \arrow[d, "\pi"'] \\ \mathfrak{t}_1 \arrow[r, "\mathfrak{f}_S"] & \mathfrak{t}_2 \end{tikzcd}$$

. Because $\mathcal{T}$ includes (a very general class of) functional values, this definition models functions associated with types (including Object-Oriented "methods"). In practice, methods are not "fields" defined for individual type-instances, but are functions implemented once for each type; however, this can be modeled by associating fields with "default" values, which would include default implementation of methods or "instance functions". To model Object-Oriented types, both instance data and methods can be represented as typed values, so that in principle methods can be "redefined" (assigned to a different functional value) for individual objects, even if this operation is disabled, or restricted, in practice. This provides a straightforward way to represent more complex Object-Oriented structures, like class inheritance with override methods. It is also consistent with practical design of some programming languages, particularly those with "prototype-based" object systems, like JavaScript and E. In these systems, each type (or at least each Object type) has a prototypical value, which includes implementation of class methods. Reasoning about type properties — for example, about potential vulnerabilities, to build "security layers" guarding access to type methods — can progress in part by reasoning about these prototypes.

2.1 Views and Objects

The category $\mathcal{T}$, defined to include complex functional value types, encompasses a broad collection of values which can provide the data for real-world programming languages. In addition to "atomic" values (like numbers), and functions, these values include aggregate types, or name-indexed collections of fields. Having abstractly identified a realm of values, the next question is to make this more concrete: how are values digitally implemented, stored, visualized, and communicated? The more *abstract* formal semantics of values includes criteria of what constitutes a value and what makes values distinct; but the more concrete semantics must address such *processes* as representation and serialization. These issues can be illustrated for atomic values, say, integers. An integer is *represented* in computer memory through binary encoding; it is represented in a "serialization", a textual and lexical communication of values, often by a character string expressing its base-ten representation. Either format might be used for *persistent storage* of the value, in a file or database. A visual display, making the value known to a human user, may print the number in base-ten characters, or graphically convey its relative size (via a scrollbar, progress bar, dial, color intensity, etc.). In general, a semantics (and semiotics) of $\mathcal{T}$-values needs to recognize at least four different genres of representations, which we can call *aspects*: via databases or *persistent storage*; via active computer memory or *runtime storage*; via text representation or *serialization*, and via visual, often interactive display, which we can call *pictural*. Corresponding *aggregative structure* can be *persistent morphology*, *runtime morphology*, *lexicomorphology*, and *practomorphology* (for practical, interactive visualization).

For *aggregate* data values, regardless of representation aspect, it is necessary to encode compositional structure as well as each constituent value. Representations can be partitioned into subrepresentations for contained values, with aggregate representation structure reflecting the internal structuration of its represented value. That is, there must be a bidirectional correspondence between the morphology of the value as a *data aggregate* and the morphology of the representation as a *complex sign*, in some (visual, digital, textual) medium. Rigorous study of these morphologies and the isomorphisms between them belongs to the field of *algebraic semiotics*, pioneered (against a Category-theoretic basis) by Joseph Goguen [5], [6]. Obviously, it is also a foundational practical concern for software development, which needs to marshal data between different points, like a database, application objects, and User Interface. Perspector Spaces can serve both theory and practice, because both the structure and composition of data aggregates can be modeled as collections of perspectors. For the sake of discussion, I will call an *object* any aggregate data structure (which is a slightly broader definition than warranted for some programming languages, for which not all aggregate data types are object types)[1]. Type fields include function values, perhaps with discriminated "object" arguments, or *sigma* channels ("sigma calculus" [1] being an Object-Oriented extension to lambda calculus), ordinary function arguments being a *lambda* channel. Both channels, as means of functions affecting external state, should be modeled within the $\mathscr{E}$ ontology.

So any object is an aggregate of fields, which may be indexed by name or also by number — that is, an object may include a resizable "typed array" of values (allowing arrays to hold values typed to subtypes of the declared array type). Exposing named-field aggregates as Perspectors is straightforward, simply using the field name (including the type name if necessary) as an identifier for $\mathfrak{o}$. Exposing arrays is also straightforward using tropes (contextualized perspectors), and a "list sequence" operator $\mathfrak{o}$ to encode tropes in sequence.

[1] In practice, the principal distinction between objects and other aggregate values being that object types include functions in which a "callee" object is one argument, distinguished from other arguments — this is, methods. As I have proposed, the set of functions defined for a type, and internal to its interface, can be considered simply as a list of functional values associated with the type as fields, albeit ones which may have a single definition across all type-instances. I will call *any* aggregate value an object, although objects will only have Object Oriented interfaces if their types include some functional values using sigma channels. The aggregative form, rather than the type interface, provides the defining criteria for object values.

So, for each object $\mathscr{O}$ there is an ordered trope-space $\mathfrak{T}^{\circ}_{\mathscr{O}}$ which can be introduced into larger Perspector Spaces. This also means that a Perspector Space $\mathscr{P}$ may include collections of tropes that are the encoding of a specific object. From $\mathscr{P}$ we can select only those tropes which collectively encode a single object — in Josh Shinavier and Marko Rodriguez's terms [8], this is a "dilation" of any single trope in that collection, which assigns it a semantic context. More generally we can define a selector $\mathfrak{s}$ on $\mathscr{P}$ as a mechanism to isolate a set of tropes in $\mathscr{P}$ according to some criterion. If the goal of $\mathfrak{s}$ is to recover the encoding of a single object from within $\mathscr{P}$, we can refer to $\mathfrak{s}$ as an *object* selector. Alternatively, a selector may seek to recover the Perspectors originating from a given data source, or reflecting some ambient semantic structure of origin, such as an HTML file. That is, a selector can extract those tropes from which a document in a formal language can be recovered, and potentially a visual representation (or persistable or serializable representation, recalling the above list of representation aspects) thereby constructed. For example, tropes can be selected which can be assembled into a valid HTML document and displayed in a web browser. We can call a selector which provides a visual representation of some data inside $\mathscr{P}$ a *view* selector. Representing structured data within $\mathscr{P}$ is often a matter of using both view and object selectors to yield a view template woven with object data.

Rather than committing to a single visualization formation, applications can use general Perspector or Trope spaces to hold data, and on their basis build different types of visual and User Interface components so as to the adapt to the environment where application are used. This is consistent with Biezunski's understand of Perspector Spaces as data spaces which permit multiple *perspectives*. It also reflects his insistance that perspective is an intrinsic aspect of semantics: "once some meaning has been uttered, there is a perspective behind it, whether we recognize it or not. It is by looking at the rules that have been applied while creating this piece of information, and by disclosing the rules that are at work — rules for establishing identity of meaning, rules for applying merging to identical subjects, and derived rules, if any — that we can provide hooks that can be used as binding points for other, similar, pieces of information" [2, p. 19].

By now, the technology to generate markup, like HTML, from structured data spaces — often using template formats like JSP — is very well-established. However, more powerful and interactive visual displays can be achieved by going beyond conventional markup, using application GUIs (coded in languages like C, C++, or Java), and/or rich graphics (including 3D and Virtual Reality as well as complex 2D images). Constructing graphics and GUIs from Perspector Spaces involves using view-selectors to isolate important tropes and then, if necessary, dilating them with sufficient additional tropes that a visual representation can be algorithmically constructed, whether as just an image or as generated code which programs a User Interface. Because markup languages, GUI code, and binary graphics are very different as formal languages, there has been relatively little research into the underlying connections between these media as sign systems and as spaces for data integration, merging explicit data with the ambient semantics of the format through which a representation is engineered. However, recent research in the "grammar" of visual forms — such as *computational visualistics* [7] — offers systematic integration of different visual formats and how they convey information. In particular, this emerging field offers a rich theoretical basis for *perspectives on Data Spaces*, the environment within which Perspectors are created and/or understood. I hope to leverage this theoretical understanding with practical tools for generating visual frameworks — graphics, markup, and GUIs — for structured data, through the medium of Perspector Spaces.

3. CONCLUSION

The Semantic Web "is not very Semantic", according to linguist Peter Gärdenfors [4]. As he then argues, logic and set theory, in themselves, are not sufficient to capture realistic semantics, formal or otherwise. Biezunski has also emphasized that the "formality" of web semantics does not exempt technology from being sensitive to the complexity of real-world concepts. Ultimately, software provides tools which must work within people's lives and understanding. The meaning of concepts, as they are modeled by computer code and by data structures, is determined by the situations in which users' will feel that the concepts should be applied, by the kind of information they expect to find alongside instances of a concept, and by how users expect to visualize and modify this information. These criteria can be given formal treatment, but a simplistic association of Ontologies to "sets of concept instances", for example, does not suffice for a formal semantics which is also cognitive and semiotic, grounded in real-life practice. Perspector Algebra, Computational Visualistics, Cognitive (and Algebraic) Semiotics, and other approaches to Semantic Networks and Domain Ontologies, allow us to sustain the rigor of formal semantics — which is not unimportant, especially considering that formal reasoning is needed in conjunction with, for example, cybersecurity — without departing too far from realistic conceptualizations. In the end, meaning is cognitive, enactive, and pragmatic, produced by the interaction between minds and their environments. Our environing worlds, "lifeworlds", are social and technological, not just natural, *constructed* according to deliberative, scientific, and political norms and values. All of these realities have a place in Semantics.

4. REFERENCES

[1] Martin Abadi and Luca Cardelli, "An Imperative Object Calculus" http://lucacardelli.name/Papers/PrimObjImpSIPL.A4.pdf

[2] Michel Biezunski, "A Matter of Perspective: Talking About Talking About Topic Maps". http://web.dfc.unibo.it/buzzetti/IUcorso2007-08/materiali/EML2005Biezunski01.pdf

[3] Michel Biezunski, "The Data Projection Model: Toward Auditable Information Systems through Unified Declarations on Data" https://www.infoloom.com/media/misc/dataprojection.pdf

[4] Peter Gärdenfors, "How to make the Semantic Web more semantic" http://yaxu.org/tmp/Gardenfors04.pdf

[5] Joseph Goguen, "An Introduction to Algebraic Semiotics, with Application to User Interface Design" https://cseweb.ucsd.edu/~goguen/pps/as.pdf

[6] Joseph Goguen, "Semiotic Morphisms, Representations, and Blending for Interface Design" https://cseweb.ucsd.edu/~goguen/pps/uid.pdf

[7] Jörg R. J. Schirra, ed., "Computation Visualistic and Picture Morphology" http://www.gib.uni-tuebingen.de/own/journal/pdf/buch_image5b.pdf

[8] Marko A. Rodriguez, Alberto Pepe, and Joshua Shinavier, "The Dilated Triple" http://arxiv.org/pdf/1006.1080.pdf

[9] David Spivak, "Database Queries and Constraints via Lifting Problems" http://math.mit.edu/~dspivak/informatics/LiftingProblems.pdf