Nathaniel Christen

Merging Full-Text Query with Data Sets: A perspective from compiler theory

Abstract

In data sharing and publishing contexts such as "Executable Research Objects", natural-language documents coexist with raw data files and, potentially, multimedia resources. In principle, these materials may be unified into a common search, indexing, and GUI framework. In the context of scientific research grids or other decentralized collaborations, the basic units of scholarly dissemination are no longer single manuscripts but rather composite packages including text, code, and data. According to FAIR (Findable, Accessible, Interoperable, Reusable) principles such content should be in available in a largely standalone fashion, hopefully usable as soon as downloaded. To this end, this paper will consider how compiler and Virtual Machine tools may be embedded within data-set code, built from source files with few external dependencies. The paper will consider how a self-contained environment for scripting or query language implementations can promote text search, raw-data deserialization, GUI integration, and other tasks to promote Executable Research Objects as insular applications helping scientists and investigators understand and assess published findings or theories.

1 Introduction

In the context of "publishing technology", one would traditionally understand a "publication" to be a natural-language document (e.g. an article, book, or book chapter). But several more recent trends in publishing and science have complicated this picture. For example: the idea of a "replication crisis" — concerns that results of influential experiments in natural and social sciences have proven difficult to reproduce during updated studies. Reaction to this "crisis" spurred the goal of "data transparency" and initiatives to document more rigorously the raw data, methodology, and analytic details behind published findings. Such initiatives in turn led to "Minimum Information" standards (e.g., Minimum Information for Biological and Biomedical Investigations, or MIBBI, which encompasses several dozen more specific specifications) and to mixed data/code/multimedia formats such as FAIRsharing (Findable, Accessible, Interoperable, Reusable) and Research Objects (including "Executable Research Objects", "Executable Papers", Research Object "Bundles", Research "Compendia", or — as with the name of one US "Data Infrastructure Building Blocks" (DIBBs) initiative — "the Whole Tale".[1]

A common theme of these various formats is that the basic unit of disseminating scientific or academic research is not (or no longer) a single document, but rather a collection of distinct files representing text, data, and potentially graphics or computer code. Natural-language text is still important — readers are most likely to learn about data sets or Research Objects via publications that introduce them — but in a Research Object context manuscripts are packaged alongside sibling assets with their own requisite functionality. This affects functionality intrinsic to text documents insofar as, to begin with, the various components of a data package can be mutually cross-referenced. For instance, terms and keyphrases within a manuscript could be annotated with links to data, code, or graphics content, if the same concepts appear as statistical parameters, units of measurement, data types, record fields, etc., within raw data files, supporting code, or visual materials (e.g., axes on a scatter plot).

Relatedly, researchers have explored text mining/Natural Language Processing tools to facilitate discovery of scientific work. In principle, computational methods can highlight which additional papers might benefit a particular researcher, given ones they have already cited or written themselves. As one example, the CORD-19 corpus was curated by the Allen Institute for Artificial Intelligence, early in the Covid-19 pandemic, as a large repository of open-access materials about SARS and Coronaviruses (much of which would ordinarily be paywalled but were made freely available by their publishers). Since Covid research was a matter of fitting different pieces of a puzzle together — especially early on, when the morphology, infectious mechanisms, treatment options, and vaccination strategies of SARS-Cov-2 were still under investigation — a computationally analyzable corpus held potential for scientists with one specialization to find useful information in articles they might not discover by mere chance.[2]

In effect, the goal of CORD-19 was not just to free up a large volume of documents to be browsed by human readers, but to augment their research via text mining. Un-

[1] For the "compendium" model, see [15]. For "executable papers", [18]. For FAIRdata, [10]. For the "Whole Tale", [13].

[2] For a general introduction to CORD-19 see [20]. Several other publications have described analysis that *used* the repository, in contrast to presenting computational details about its implementation. Also as relevant in this context (I hope), I wrote an extended analysis of CORD-19 in the second half of Chapter 3 of [8].

SOURCES Demo t code: https://github.com/scignscape/PNP JATS: https://scignscape.github.io/PNP/documents/A-perspective-from-compiler-theory.jats.xml .gt file (parses into C++ objects): https://scignscape.github.io/PNP/documents/A-perspective-from-compiler-theory.gt

fortunately, the most common format for dissemination of those publications — PDF — is flawed from a computational perspective, because the internal PDF representation of natural language often differs from how human readers would transcribe the same content (e.g., when seeing it rendered in a PDF viewer). Such details as ligatures, headers/footers, hyphenation, and font-face variations can introduce artifacts in text extraction (or "scraping") from PDF files.[3] In addition to problems with "ordinary" language, encoding errors also arose in the context of "special" character strings such as organic or chemical formulae; the CORD-19 documents did not use a consistent mechanism for notating these (potentially semantically significant) textual elements (or, at least, structured data was not preserved in the final PDF).[4] These problems were openly acknowledged by the team behind CORD-19, who described the steps they took to (imperfectly) convert PDF files to a machine-readable JSON format. They even issued a "call to action" encouraging publishers to adopt a more consistent text representation.[5]

Text Mining and Natural Language Processing are sophisticated use cases for machine-readable text encoding, but more mundane functionality is relevant as well, such as searching within PDF documents. Given the imperfections of PDF text representation, it would be reasonable for PDF viewers to run text searches first through more-precise data about text content — if PDF files provide as much — in lieu of PDF-internal text objects. For example, many PDF files are currently produced from XML formats such as JATS or BITS; if these intermediate assets (or at least links thereto) were embedded in PDF directly then they could be extracted and employed by PDF viewers as structured text encoding, in preference to the more "semi-structured" data one may obtain from methods such as text() in the Poppler library (which I've mentioned in footnote 3). Similar use cases might be compiling human-readable document indexes, or comparing two different versions of a manuscript during an editing/reviewing changes cycle.

For contemporary publishing, in short, natural-language content should be considered a computational artifact that often co-exists with raw data and computer code in Research Object contexts. Natural Language Processing proper — e.g., deriving parse-graphs or Syntactic Categories (parts of speech) for sentence components — is at least tangentially connected to these trends, but the (at best approximative) methods of NLP are outside the scope of topics addressed here. Instead, the instantiation of "natural language" content in this present context involves fully-deterministic encoding of textual material in the specifically *written* environment of academic and scientific presentation, which includes not only linguistic conventions but also normative practices for written manuscripts: inline and block quotes, citations, footnotes, figure illustrations, book indexes, and so forth. Collectively, these might be designated as facets of *discourse* rather than language proper. Given the co-existence of text with raw data and/or code within unified data packages, we can consider to what extent User Experience and structural models should be conceptualized as applying simultaneously to text and to data sets, as well as GUI components that present them respectively in the form of visual resources.

For example, we can isolate specific areas of potential GUI functionality in contexts where human readers are seeing renderings of text documents (or pages therefrom): copying the words of a sentence to the clipboard; searching for information about a single word, phrase, or term; examining the original text from which a quote was sampled; accessing a data set and/or multimedia resources that are indirectly included in text documents through figures/graphics (for instance, if a picture shows one frame from a video, finding and watching the video in its entirety). We can then identify related capabilities operational in the context of data sets and supporting/analytic code: the data fields of tuple-like records, for instance, are semantic objects potentially governed by analogous conventions as technical terms present in text; algorithms described in a manuscript can be studied concretely in the implementation bodies of data-set functions; readers could replicate simulations discussed in research papers with their own parameters.

[3] To clarify these remarks: PDF code libraries typically have some procedure that can be called to obtain text from one page and/or window area. In C++, for example, the Poppler library has a text method of Poppler::Page which optionally takes a rectangular sub-area (this allows scrapers to eliminate headers and footers, for instance). The strings returned by this method, however, will usually differ from the input sources captured as JATS, for example. The most straightforward contrast would be among hyphenated words, but other anomalies may occur, such as unexpected space characters distributed between letters of words in special fonts (small spaces may be present in the internal PDF text that are invisible to human readers, but methods like Page::text have no mechanism to report different sizes of text characters). Likewise, depending on the library being used, text-extraction routines may or may not "unpack" ligatures like "ffi" to their constituent letters.

[4] In my own analysis of CORD-19 mentioned above, I gave the example of a compound *2'-C-ethynyl*, which is also notated as *2 0-C-ethynyl*. With these alternatives, plus the fact that the JSON markup appeared sometimes to incorrectly parse the strings — i.e., breaking the character-sequence at the wrong location — the corpus overall has multiple different representations for the same formula. This obviously obscures the fact that each of these represent the same concept, and indicate how semantic annotation would be more reliable.

[5] Under a section explicitly titled "call to action" they counsel that "a number of challenges prevent easy application of NLP and text mining techniques to these papers . . . . [T]he primary distribution format of scientific papers — PDF — is not amenable to text processing. The PDF file format is designed to share electronic documents rendered faithfully for reading and printing, and mixes visual with semantic information. Significant effort is needed to coerce PDF into a format more amenable to text mining, such as JATS XML, BioC . . . or S2ORC JSON.... [T]he community must define and agree upon an appropriate standard of representation." [20, p. 8]

Perhaps it is not obvious that the same computational toolsets that might be used to implement functionality related to text documents would also be applicable to Research Objects and data sets. Certainly there are structural differences between text elements (sentences, paragraphs, etc.) and most forms of scientific data (especially table/record style formats). Nonetheless, the increasing interpenetration of text and raw data in Research-Object-like contexts implies that implementations regarding text and data as part of a common computational value-space should be seriously considered. For example, the code supporting an Executable Research Object might be engineered to treat both text elements (e.g., sentences and keywords/phrases/annotations) and data-set records as objects accessed and GUI-rendered via similar protocols. As a concrete example, similar methods might be used to populate context menus whether activated in the area of a keyphrase within a PDF window, or of a table-view (or any other intra-window control) that shows an individual data field, record/tuple, or statistical aggregate. Likewise, search protocols — whether triggered by an actual query language or rather by a query-like interface within a (for instance) C++ code library — might be engineered in parallel within text-manuscript and data-set contexts.

For the purpose of this paper, text/data integration will be envisioned to be performed via a dedicated Virtual Machine. This perspective remains agnostic as to whether the VM would be concretely utilized to implement a query language, scripting environment, middleware tools, or any other tactic in relation to published Research Objects. I *will* work under the assumption that authors and digital editors seek to align with FAIR priorities, which means in particular that data sets are *executable* with minimal external dependencies. In other words, as much as possible, interested readers should be able to obtain the content of an Executable Research Object *in source code* fashion and use those sources — e.g., by building an entry-point executable — without having to separately obtain and/or compile or install additional resources not packaged in the data set itself. Programmers producing a self-contained package in this sense may reasonably assume that readers have access to a working and relatively up-to-date development environment typical for their language of choice — for example, a C++-based bundle obviously requires a C++ compiler, and canonical libraries such as boost or Qt are acceptable external dependencies. Nonetheless, publishers should be careful to ensure that clear instructions are given for users who need to set up a prototypical development environment to begin with.[6] Source files obviously have *some* dependencies (at least, either compilers or scripting-language interpreters). However, I make the assumption here that Executable Research Objects are predominantly disseminated in source-code fashion. This is different from projects such as Kaggle (or other digital-notebook software, holding out the possibility that data sets might be distributed as Kaggle notebooks or similar), or **reprozip** [17] (which packages up all necessary binary components that can be identified as preconditions for data-set code to run on the developer's own computer).

I will not specifically address the question of whether binary or source-based dissemination is easier for end-users, either in general or vis-à-vis any particular data set. My goal is rather to examine requirements for a Virtual Machine that can function in the former context — in particular, that itself could be built with minimal external dependencies. I assume that, even in a context where Research Object code is published as source files in a mainstream programming language (C++, Java, etc.), there are some domain-specific or fine-tuning details that might usefully be coded in a special-purpose language. For example, the information encoded in a data set may be approached *de facto* as a database which accepts queries in an SQL-like (or some flavor of NoSQL) format — potentially extended to "full-text search" with associated text documents, via query specifications geared to text content (*cf.* [5], for example). In other scenarios, an Executable Research Object could employ something like a built-in scripting language to run analytic code, fine-tune GUIs, compose plugins for external scientific applications, etc. (The principle that Research Object code should be self-contained and source-based does not preclude users exploring data sets, or individual files therein, via discipline-specific applications that have analytic or visualization capabilities conducive to advanced research — the idea of Executable Research Objects is not necessarily to replace the tools that scientists use for their core investigations, but rather to provide accessible code for evaluative and pedagogical purposes and/or for anyone not familiar with or able to run the "professional" software).

1.1 Deserialization and Other User Cases

To clarify why a VM can have nontrivial roles in a Research Object context, I will identify potential VM use cases. One obvious example is deserializing data sets. Many subject areas have special-purpose data formats reflecting the unique kinds of information pertinent to their scientific field. Some of these are narrowings of canonical languages like XML or JSON, but even in those cases dedicated code is needed to traverse a DOM hierarchy, respond

[6] And external dependencies should be favored which are easier to use. For instance, if two C++ libraries provide similar functionality, developers should compare them based on how easy or difficult it is to install the libraries assuming they are not already present on one's system (while taking into account that the package manager for a user's linux distro, for example, may not be up-to-date). If one library is noticeably better in this regard, the Research Object should put more emphasis into code that works with that library as a dependency as opposed to the alternative.

to SAX events, or otherwise walk through nodes of the generic data structure to extract individual values that populate fields of domain-specific data types. Other formats have their own text and/or binary encodings, such that some form of parsing library is also necessary for deserialization (possibly in conjunction with post-processing logic analogous to, say, DOM navigation). Even a simple standard like CSV often does not map precisely to serialized objects, given that most aggregate values cannot be fully decomposed into tuples of scalar data points (instead, CSV circuitously encodes vector/array structures, or values sampled from enumerated classes, or special-purpose fields that are usually undefined or default-initialized and thus encoded as empty cells).[7] That is, code to deserialize CSV can be equally complex to XML or JSON (likewise for alternatives such as RDF, multi-dimensional arrays, multi-value SQL, and so on).

Any researcher presented with raw data files that must be loaded as in-memory objects (or typed values generally) will hopefully not have to write deserialization code from scratch; instead, scientific communities often endeavor to publish code libraries to parse/traverse the relevant file types. Nonetheless, such libraries are frequently maintained by individual academic departments or other groups working with limited resources, and they can have problems related to versioning (the code being incompatible with newer, or older, representation specs), lack of breadth for different programming languages or operating systems, external dependencies (or other details that make the libraries hard to build and/or use for some people), and other issues affecting FAIR-style accessibility.

In principle, a VM in this context could provide a common environment for implementing deserialization libraries, at least as a supplement to those provided via traditional programming languages. For widely-used data formats, we can assume that many different published data sets will require users to parse data files sharing a particular format. The VM instruction set could be interpreted by multiple languages in variegated environments — i.e., the VM runtime itself could be ported to multiple language and operating-system contexts, and then distributed in source form alongside dataset-specific code (including deserialization algorithms in bytecode form). Alternatively, the VM runtime might be made available as a binary component (so long as it is tested against multiple platforms, to the extent that it serves as a "FAIR" component).

Assuming that this is indeed a primary use case, notice that the VM would not need to be highly optimized in terms of execution speed, multi-threading support, fault-tolerance, and so forth. Even as the compilation target of a special-purpose scripting format, the VM would not primarily function as host to a general-purpose language like Python or JavaScript. Instead, it could be designed from the ground up around requirements specific to data publishing and Executable Research Objects.

After a data set is loaded, it will typically be viewed and studied by the researcher who downloaded it. Potentially, this involves both dedicated GUI windows showing details about object-collections in aggregate and also individual objects/records (e.g., something like a GUI "form" showing the values of individual data fields in an organized layout, perhaps with list-views or hierarchical "accordions" for non-scalar information) as well as statistical overviews, presented via (interactive) graphics (charts, plots, diagrams, histograms, etc.).[8] This is another area where VMs might be pressed into service.

Suppose a Research Object is implemented in C++: although in principle all the GUI code could be built specifically for that one publication, to avoid reduplication of effort the authors might prefer to reuse components provided for the relevant book series, journal title, science grid, and so on. Assume, again, that such C++ sources are included within the data set itself; in principle, each Research Object could thereby reuse generic components but adopt the underlying C++ code for their own purposes, given the unique research methods and data profiles driving their investigation. At least vis-à-vis some functionality, however, such adaptations might fit predictable patterns conducive to a special-purpose script or query format rather than manually rewriting C++ code. For instance, context menus tend to need specific action options unique to the current data set, so "scripts" against an embedded VM might be employed to populate those menus, being called somewhere after the triggering event (a right mouse-button click or **ctrl**+left button) is registered. Similar comments could be made about areas of functionality such as setting up initial parameters for a simulation; managing text/data cross-references; logic for preserving application state and user preferences; etc.

For another example, suppose one graphic in a publication shows a video frame. An obvious context action for

[7] One example I am familiar with is the Toxic Release Inventory (TRI) project of the Environmental Protection Agency. The EPA publishes annual reports, available on a state-by-state basis, recording incidents of environmental contamination due to toxic leaks (from a list of several dozen known harmful substances). The raw data takes the form of CSV files with over 200 columns, but there certainly is not a one-to-one correlation between those columns and reasonable data fields of deserialized objects. Instead, some groups of columns serve to indirectly encode vectors of similarly-typed value, with up to perhaps five elements; others encode special conditions that are applicable to only a few records in the collection; still others encode text strings which have to conform to a restricted list of enumerated values. In short, a class type holding record-data would use a collection of vector-like collections, **enum**s, boolean flags, and so forth to represent many of the data-points. All of this obviously complicates the deserialization process ("CSV-to-Object mapping", so to speak).

[8] "Accordion" is a common term for a sort of GUI control that has multiple inner segments, usually stacked vertically (i.e., higher on the screen to lower) each of which can be independently expanded

menus near that image is for the user to watch the video in question (perhaps with an option to start shortly before the still that is depicted). Plus, not all science videos are from cameras: if the video actually records the run of one computer simulation, the user could instead be brought to GUI controls where they can (as needed) set up parameters and then execute a similar simulation on their own.

Here is a related case: suppose a publication names chemical compounds or formulae, and it is possible to obtain (from a service such as Protein Data Bank, or PDB) data files that certain applications are able to use for constructing molecular visualizations, modeling that substance's atomic structure. If the PDF viewer interoperates with such software/components (I will review possible strategies later) then launching the graphics' window with that specific molecular file loaded would be a useful context option, assuming the menu is activated in proximity to the annotated text (viz., where the name/formula is present).

Context menus are a good case in point because, in a PDF display, most of a reader's attention is focused on the document and not on GUI controls scattered by the edge of the window. Therefore, functionality requested in the document's context would typically be activated via a context menu rather than a GUI button or other clickable area (the menu could be localized to individual words/ text segments, particularly for keyphrases and indexed headings; or figure graphics, hyperlinks, and other components of a publication that might have extra programming attached). However, similar comments apply to other interactive controls: for instance, when a user enters a term into the PDF search box, windows operating in a Research Object context might well ask whether the search should be applied to the entire package, not just the current PDF file (exactly how to interpret searches over, say, raw data files depends on the data set in question). A PDF viewer designed specifically to embed in Executable Research Objects might furthermore adopt a standard protocol for adding GUI buttons/controls to interactive screen areas — here, individual projects could place clickable elements that invoke functionality more general than just what should be localized to particular areas of text/documents (and therefore less suitable to context menus).

Indeed, GUI programming overall evinces a variety of common practices that are relatively standardized and help users feel comfortable with a desktop-style software component. For instance, tooltips help users identify clickable regions when they hover the mouse nearby; conventions such as accordions, split windows (where one half can be enlarged while its sibling is contracted, by pulling a splitter left/right or up/down), or notebook controls (where one tab is visible at a time) help accommodate more information than can be present in a window at any one time. By expectation, functionality is distributed amongst screen areas of different size (for instance, notebook tabs — whose roles include switching from one tab-view to another when clicked — also typically have a small *x*-shaped close button at their top-right). Visual cues such as a thickened border around notebook tabs help users grasp application state (e.g., which tab is currently visible) with little effort (similarly, a common policy is to append an asterisk to file names when the files have not been saved in their current state). Cursor icons, likewise, convey whether clicking, dragging, or some other logistic is sensible at the current screen position.

For a similar example, consider how user gestures are implemented for more complex interactions than just, say, clicking a button. In 3D graphics and 2D (manual) image annotation there are often multiple possible interpretations of actions like dragging the cursor or operating the mouse wheel; applications often allow users to specify their intentions by augmenting their gestures with key-presses such as *ctrl* or *alt*. When constructing an arrow overlay, for instance, some gestures lengthen the shaft; others adjust the head; others rotate the arrow to a desired angle. Once certain patterns of interaction (through mouse and/or keyboard) are implemented in one part of an application, it is helpful for users to adopt similar conventions globally.

The overall suite of design patterns common to desktop GUIs are not just programming concerns, but also suggest a certain "semantics" for GUI components. For example, each window area can be associated with visual state-cues (e.g., whether a notebook tab or a file view is the one currently visible), cursor appearance, gesture logistics (what interaction are recognized there and how details such as key-presses modify them) and so forth, and this represents semantic information associated with controls (and their parts), potentially incorporated into documentation and user tips. One way to enforce consistency across GUI components is to defer certain tasks away from the compiled code and toward dynamically loadable scripts. For example, software can run in a dedicated "demonstration" mode that employs screen overlays to show information about particular window controls. That functionality could be implemented via VM scripts orthogonal to and developed after the main application.

1.2 Text Search and Data Analyses (or demonstrative summaries thereof)

Another potential VM use case involves data analysis. Not to imply that a dataset-embedded VM would be part of the ecosystem producing published findings; but analytic code might be separately produced that illustrates algorithms and information profiles in a more pedagogical and

or contracted (i.e., fully hidden). In the latter state only a (typically one-line) header remains visible.

demonstrative manner. More precisely, computer code can serve as a tangible resource for studying algorithms and data structures, because most programming languages are more readable than, say, inscrutable mathematical formulae. Even more so, source files may be written in a style that accentuates this expository role: code constructions or annotations are able to clarify details such as expected value ranges, coordinate systems, scales and units of measurement, cross-unit conversions, unit-decorated types, pre- and post-conditions, type invariants, etc. This degree of in-code specificity is not usually present in mainstream languages, so a Research Object VM could support scripting dialects which attempt to make code assumptions and pedagogical use cases as transparent as possible.[9]

Lastly, I mentioned earlier that researchers may prefer to load data sets with external software even if FAIR principles imply that Research Objects can be used on their own. So — although authors should attempt to provide sufficient internal resources that someone with access to a data set can find some value just with the materials explicitly included — one may reasonably anticipate that some scientists will seek out the additional capabilities of dedicated software for their field (at least if a Research Object involves information of a variety where such software exists). Data sets (without violating FAIR) could be designed to facilitate being loaded and visualized via such discipline-specific applications where applicable. Yet, in these cases, even when raw data files are encoded in a format that a given application recognizes, raw files alone are not necessarily sufficient to ensure that loading applications can access them properly (for example, any program with spreadsheet-like functionality can load CSV files, but such data rendered as table cells may not properly convey its structure and properties). To address these situations, many scientific applications supply a plugin mechanism such that data sets with special preparatory requirements can provide plugins to implement these, alongside the raw data. The problem here is that plugins can be tricky to compile and install. One solution is to develop a generic plugin that enables an application to execute VM instructions, which then serves as middleware such that Research Object plugins could be implemented as scripts compiling to that VM, rather than interfacing with the host software's plugin mechanism directly.

Having thereby outlined several use cases on the data-set side, I will also touch on how VM technology could support text-related tasks, such as indexing and full-text search. One obvious possibility is to achieve search functionality via a dedicated query language. If we assume that textual elements are stored as objects in some mainstream language (e.g., C++), then in principle queries such as "find the first sentence including the name of *this author*" (the author of a given bibliographic entry) could be satisfied by calling (in this case) C++ methods. However, for development and prototyping it might be easier to express searches through text files or string-lists that can layer over the relevant class methods.

Other possible use cases arise in the context of PDF viewers, in which could be embedded a VM interpreter/runtime. New functionality may then be introduced via PDF-related scripts; for instance, extra context-menu options, interactive features pertaining to figures/graphics, customizations for PDF annotations and comment boxes, working with indexes, or building domain-specific search capabilities (aware of semantics specific to a given scientific subject area, for instance).

The range of use cases addressing both text and data content suggest that VMs embedded in Executable Research Objects can be useful across their different asset-types (documents, code, raw data, graphics/multimedia). Under the assumption that a compiler and runtime environment (bytecode interpreter) is offered among the data-set package, then other components — GUI classes, a customized PDF viewer, deserialization code — could each utilize (e.g., link against) the VM implementation in turn, achieving consistent scripting/query design across the various components.

Having thereby identified the *purpose* of a VM engineered for the data-publishing context, the remainder of this paper will examine certain theoretical and implementational details relevant for these use cases in particular. The discussion will address both compiler and runtime concerns, and also consider the forms of information structures — both textual and data sets — that VM-compiled components would likely need to traverse/navigate, insofar as the relevant structural details may be directly incorporated into the VM's instruction roster.

2 Sentence Objects and Other Textual Elements

Conceptually, we assume that a Research Object VM will operate over a tableau of information structures, including both text and dataset content. That is, natural-language documents are computational artifacts that are coextensive with raw data files and other research assets, subject to similar algorithms (or even the exact same procedures, in contexts such as semantic searches applied both to documents and to data files).

Text-encoding specialists have devoted some attention to investigating the best formats for representing written

[9] To put it differently, the original analysis might be performed in C, Python, Matlab, fortran, etc., but all or some of that code would then get reimplemented in a more explicitly "pedagogical" language, as if developers seek to explain the code to an outside party.

text. In recent years, XML-based standards such as JATS and BITS have become popular, producing human-readable PDFs through techniques like XSL-FO while also enabling natural-language content to be analyzed via generic XML protocols, such as XQuery and XQSE (XQuery Scripting Extensions). JSON-based formats are also used, like the JSON-S2ORC protocol adopted by CORD-19 (mentioned earlier). Specifications such as the Text Encoding Initiative (TEI) may employ a generic representation for convenience (e.g., TEI-XML) but they are intrinsically a special-purpose data format optimized for natural language (for instance, TEI, established in 1987, predates XML itself). In a similar vein, special markup languages like TAGML (the Text-as-Graph Markup Language) — which are structurally distinct from straightforward tag hierarchies (and therefore not just a realization or dialect of XML) — have been proposed as more faithful to the inner structure of text content than generic formats (XML, JSON, etc.).[10]

Debating the merits of, say, TAGML versus JATS/BITS — or alternative XML languages like BioC, developed for PubMed Central [9] — is outside the scope of this paper. I will note, however, that the important details for questions about the most efficient text representation is less the actual structures modeled by a given format but rather how particular structural details affect *algorithms* that target the relevant information. Any encoded data structure is conceptually a *space* in the sense that we can use some terms of discrete topology therein: we have a notion of "movement" within the space; some parts are closer to each other than elsewhere; we can consider indivisible elements of the structure and picture them as a "foreground" or "cursor" that an algorithm visits before moving onward. Discrete information spaces are analogous to graphs in this sense, but they may entertain structural parameters that are more involved that ordinary graphs: for instance, "property hypergraphs" have nodes inside other nodes, plus key/value attribute lists on nodes and/or edges.[11]

The structural facets of any meta-model determines how algorithms transform and/or extract data from structure-instances. The issue is not the explicit structure of a single instance of a model, but rather the various criteria which may define a "structure" to begin with. For example, an important contrast between XML and TAGML is that the latter supports concurrent/overlapping markup, which can only be achieved in the former indirectly (e.g., via empty "milestone" nodes). Endorsing one approach over the other is a rather abstract question. A more concrete assessment can consider actual processing tasks and address whether algorithms that "quantify" over concurrent markup structures, or conversely purely hierarchical ones, are easer to implement (or, likewise, test, or maintain). Assuming that most algorithms navigate around a structure by tracking a "foreground" node — some kind of visitor pattern where one node at a time is the one visited — then TAGML-style overlap complicates the decision-space for node-to-node transitions. For instance, an element that begins in the *interior* of some quasi-parent node, but *ends* properly after the latter, does not have a single unambiguous parent. After completing whatever processing is needed for an element in this case, would one more profitably revert to the "quasi"-parent in scope at the *beginning* of the element or, rather, at its end?

Moreover, TAGML recognizes two distinct forms of de-facto parent/child relations, namely "containment" versus "domination". In the latter case, one node has a logical or semantic nesting inside another; the parent forms a context which is semantically intrinsic to the child (e.g., a "last-name" field in an overarching "proper-name" tag). In *containment*, on the other hand, one node's content lies inside another's, but there is no logical mandate for this relationship (for instance, paragraphs often do end up as contained in a single PDF page, but paragraphs are not intrinsically situated within pages). There are then, at least four different varieties of nesting: overlap either vis-à-vis beginning and end, and interiority either via domination or containment. By contrast, XML has a single parent/child relationship. The more complex TAGML architecture may or may not make any particular algorithm easier to implement, but at least we can be sure that the structural differences between the two formats are tangible in how code targeting TAGML has to keep track of a more complex space of parent-like internode connections.

Conversely, XML algorithms would presumably be more complex in situations where concurrency actually is a natural representation of an underlying semantics. Falling back on zero-width elements as start/end landmarks for structures crossing against a DOM hierarchy is not a seamless translation of node-overlap semantics to a non-concurrent

[10] For TAGML, see [3] or [4]. This is one of the more recent proposals for markup languages that incorporate SGML-style features such as concurrent/overlapping markup, and on theoretical grounds might be considered a better conceptual match to natural-language text than XML. Of course, although XML was originally conceived as a more practical and standardized version of SGML, not all SGML features were preserved, which to some extent limited XML's expressiveness.

[11] For property hypergraphs, see [1]. There is extensive literature on both property graphs (conventional graphs with key/value tuples added) and hypergraphs (where hypernodes contain other nodes, and hyperedges contain other edges); the property hypergraph paradigm merges both of those representational devices. I have an overview of how different graph formats extend one another, in terms of expressiveness and structural features [8, pp.134–143]. The property-hypergraph model can itself be subsumed into others. For instance, (hyper)edges may be associated with contexts which render them valid on some occasions but not others. The collection of edges leading from a node can be ordered into a priority or sequence ranking. Connected hypernode and hyperedge sets may form subgraphs which are also available for key/value properties. All of these variations are meta-modeling factors which here I try to convey via the concept of "topomorphic regimes".

environment.[12] If an XML file does indeed use milestones to emulate overlapping markup, then the presence of landmark tags would have to be tracked by XML processors as state data or a parallel hierarchy separate from the XML structure itself. As such, algorithms would need to maintain a separate stash of parsing/processing objects, which presumably complicates the implemented procedures.

Added complexity is not necessarily a bad thing, but these examples suggest that markup formats come with different tradeoffs. Features that make some processing routines easier to implement can render others more difficult, and vice-versa. But in the context of text encoding, discussing such tradeoffs solely with respect to *generic* formats like XML or TAGML skirts the key point that here we are primarily concerned with the representation of *text* content, not generic data structures. It is one thing to serialize and deserialize information in general through the nodes of a hierarchical or concurrent document format. It is another to focus specifically on the structural properties of natural-language texts.

Insofar as XML, TAGML, JSON, RDF, and so forth are *structurally* different, such variation may more properly be described as "meta"-structural, or concerning those parameters that articulate the individual properties of an XML (etc.) structure in general. Any given instance of XML (etc.) structure takes its identity from the full set of details endemic to the XML format: element names, attribute keys and values, text content (if any), and nodes' parent/child and sibling relationship. The parameters constituting a TAGML instance are different, given the latter's alternative approach to nesting (discussed above), attributes (which here are more complex than key/value tuples), and text content (potentially split into hypernodes). It is difficult to compare XML and TAGML structure *instances* because what it means to be a structure in the former case is not quite the same as in the latter. We can, however, compare "meta" structures in that we can present the suite of parameters through which any single XML instance is individuated, and by comparison the analogous parameters constituting the TAGML architecture. Similar comparisons can be made vis-à-vis JSON and RDF-like formats (e.g., text encoding via property-hypergraphs).

For purposes of discussion, I will use terminology here to the effect that different markup/representation formats evince different "topomorphic regimes", where "topomorphic" literally means "shape of place". This term, found occasionally in chemistry and mathematics, does not appear to be defined in a computer science context; but I think it is a useful neologism to address the overall class of algorithms and data structures subject to "visitor patterns" and variegated constructions as to criteria of individuation for individual structures. That is, a given representational protocol is associated with a distinct topomorphic profile insofar as data is encoded via an aggregate of "sites" with specific navigational possibilities from one site to another. A topomorphic *algorithm* is one that extracts information by navigating to a site of interest, being able to obtain specific data points in the context of any one foreground site — e.g., the text content and attributes values (per key) of an XML element — before traversing to a different site. The "neighborhood" of a site groups all data that can be extracted directly as well as nearby sites accessible via a single transition-step. For instance, the neighborhood of an XML node includes its text, attributes, next and previous siblings, parent node, and first child; potentially expanded to its full child-node set and parent-ancestry.[13] The neighborhood of a TAGML node is similar but more complex due to TAGML's looser attribute and containment protocols. The neighborhood of an RDF-style site — for instance, a hypernode in a property hypergraph — would be (in this case, more involved than simpler Semantic Web triples) any contained nodes inside the hypernode; its key-value properties; and other hypernodes reachable via labeled graph edges. The neighborhood of a JSON "node" would be a tuple of values (for an array), or key/value pairs (for an object), or a single scalar value, depending on where the given node was positioned in a JSON hierarchy.

When text content is encoded via XML, TAGML, JSON, etc., it takes on the "topomorphic" properties of the underlying representation. For instance, navigation between consecutive paragraphs amounts to navigating between sibling paragraph elements in an XML tree, assuming that paragraphs are the basic units of encoding. However, natural-language text has its own topomorphic profile which is only approximately captured by "topomorphic regimes" *in toto* of generic formats adopted for text encoding. In this section I will consider topomorphic properties of text on its own terms, separate and apart from how it is "serialized" via markup.

2.1 Examples of Analyses Requiring Sentence Objects

The following cases may help to clarify why markup formats — when they are used not just for the interchange of text or metadata, but as intrinsic components of publishing workflows — can be difficult to use for certain kinds of search and content-management requirements. These are situations where computing over text/discourse elements represented as objects in computer memory is more effi-

[12] Milestone tags, in their structural (emulative) purpose, are not intrinsic parts of tools such as XPath and XQuery — apart from their basic status as zero-width elements, which does not by itself convey their special role.

[13] An algorithm might only ever need to visit child nodes in sibling order. However, at least some XML processors present the option of moving to any child node via a numeric index

cient than traversing DOM nodes and hierarchies. As such, a framework such as a "Sentence Object Model" — where sentences and other discursive units are already converted to code objects — is more convenient than (for example) XML files where nodes must be mapped to objects as a preliminary step.

Categorizing Inner Documents via Annotations You are working with a large manuscript assembled from many smaller documents. Their authors have contributed PDF comment boxes associated with specific sentences, paragraphs, or sections of their text, and have employed a "handle" syntax to mark themes relevant to these content units, according to a standardized vocabulary. Your goal is to scan each of the comment boxes so as to identify all inner documents which address specific themes. That is, you want to build a mapping from theme labels to lists of inner documents (or chapters, etc.) which could be pressed into service after splitting the larger manuscript into smaller parts; or utilized to assemble glossaries, enhanced ToC pages, web navigators, and so forth.

To complete this task it is necessary to execute procedures that can obtain the content of all PDF annotations, indexed by page number. One would then search each annotation-text for every characteristic handle that indicates relevance vis-à-vis individual themes, building up an std::map style data structure whose keys are theme objects/labels. Following the comment box page number, one could moreover identify the chapter or inner document including that page, to serve as a value set for the aforementioned map (depending on the set up, more granular locations such as paragraph ids may be available as well). On the basis of that map data structure, the algorithms could generate additional code (e.g., for a glossary) or manipulate secondary files: suppose, for instance, that one goal is to split the manuscript into one file per chapter, or per inner document. The algorithm would then traverse the map structure to build a second one whose values are file names (while creating those files as necessary). These coding requirements — working with object-based data structures and local filesystems — call for programming in full-fledged languages such as C++ rather than queries or XML transforms.

As a related use case, consider how comment boxes might be leveraged to construct a printed index (or also, secondarily, a search-engine/vector-database sort of index) with attention to rhetorical details. When the text of a manuscript provides a citation or names a person or theory, authors could be encouraged — either directly through input sources or via PDF annotations — to summarize positions taken vis-à-vis that subject matter (e.g., endorse, critique, or neutral/evaluative). When the sources or comments are analyzed to extract those notations, the printed index could be extended by incorporating that material into index entries. For example, instead of an undifferentiated heading locating "Keynesian economics", the index might produce subheadings such as "Keynesian economics, arguments for"; "Keynesian economics, arguments against"; "Keynesian economics, mathematical models"; "Keynesian economics, and macroeconomic theory" and so on.

Extracting Linguistics Annotations You are working with a linguistic article that contains a list of sample sentences/passages, according to a common linguistics presentation style. These examples are intended to demonstrate specific claims (about parsing structure, conceptualization, cognitive schemas, etc.) or to serve as a basis for subsequent analysis. Such samples — which might be obtained from written corpora, audio records, or simply invented by the author to highlight specific linguistic topics — are commonly used in this discipline and typeset in formats that visually distinguish examples from primary text. Moreover, the samples are often printed with id numbers, contextual comments, and sometimes special symbols representing phonology, agglutination, word order, morphosyntactic patterns, syntactic categories, etc.

Because this mode of presentation is so common, researchers could benefit from a searchable repository of these examples, as a special kind of language corpus. That is, multiple linguistics papers may be scanned for listed samples and each of those examples extracted into a common format, to be folded into a multi-publication database. Interested readers could then browse the samples to find articles that address specific syntactic or semantic themes, particularly if each item is classified according to the linguistic subject they exemplify (or serve thereto as a basis for analysis). Moreover, it is not uncommon for the same case study to be analyzed by multiple authors, potentially creating interesting contrasts in theories and paradigms. The whole process would be accelerated if publications employ a common format (e.g., distinct XML tags) to notate the examples; but the full extracting and database persistence logic would need specialized programming.

Embedding a Publication Viewer in a Chemistry Application You are working with an article about biochemistry, and the text includes discussions of compounds identified either by a common name or a chemical formula. The authors and/or editors have been attentive to encode these special terms in a structured format (e.g., special XML tags) and have also added informative suggestions via PDF comments. Your goal is to find all chemical terms/formulae which can be associated with molecular-structure files in formats such as PDB, MOL, CML (Chemical Markup Language), or similar standards. Those files may then be read — and converted to GUI displays — via science applications dedicated to visualizing and analyzing chemical structures in three dimensions (showing positions of atoms, atomic groups/arrangements, bonding

strength, sub- and inter-molecular forces, and so forth). A similar example might be a chemistry textbook, with custom classroom software.

For this kind of context, special programming is needed to extract textual data about chemical names and formulae, and potentially to search for molecular files via science APIs. Furthermore — in the hopes of best leveraging 3D visualization capabilities — the PDF viewer has to interoperate with science applications, either through a common message-passing protocol or by embedding the entire PDF code base itself as a plugin or extension. Ideally, one or more molecular visualization applications themselves can be modified to work directly with these special PDF components, because — once files are loaded into the software — the specific chemical compound being studied, and its properties, could be cross-referenced to locations in the original research manuscript (or, analogously, the students' textbooks).

Along similar lines, suppose a chemistry paper includes various terms for named proteins/compounds, amino acids, etc. (just considering the molecular-visualization context). With these specific technical entities, special-purpose supplemental data and annotations can be relevant, such as PDB accession numbers, launch-requests for external visualization tools, transcriptions between names and formulae, etc. It would be reasonable to include access to such information among context-menus activated both in PDF window areas proximate to names/-formulas in text form, and in software tools that might be drafted for interactive displays or scientific analysis. In other words, context-menu implementations — or at least hooks into points where menu options are populated — will ideally be synchronized among different parts of a Research Object, including text (e.g., PDF) viewers.

Papers with Geospatial Datasets Suppose a manuscript takes its place among a Research Object whose information content includes geospatial data, with GIS "attribute layers" encoded via columns in the so-called XYZ format (an alternative to latitude/longitude; one more directly usable by digital map displays). It is entirely possible that the main publication will include some discussion about this format, which establishes cross-reference points between the text and data sets. Meanwhile, included code — if it does indeed seek to make the raw files usable for analysis — will almost certainly include procedures for XYZ to *lat*/*lon* conversion, as well as distance metrics in the former, computations related to Mercatur projections, and so forth. These calculations thereby yield cross-reference points between the source code and components both of the textual and raw data varieties.

This and the prior example might even be combined. Data sets addressing environmental contamination (such as the EPA's Toxic Release Inventory, mentioned above in the context of CSV files) include both GIS and Cheminformatics objects. Harmful substances can be detected in soil, water, or accident reports, and each data point involves both the geographic location/area where an observation occurs and the specific compound inolved. The former lends itself to digital maps; the latter to molecular visualization displays (to help explain visually those qualities that make a substance dangerous to people or ecosystems). Class types and GUIs for both of these genres would then be applicable to Executable Research Objects surrounding the publications.

These hypothetical examples (which actually are inspired by projects I have worked on in the past) are cases where extra programming is needed for some important part of the document-prep workflow, one that stands apart from familiar publishing pipelines. Such applications as comparing manuscript versions side-by-side on a sentence-by-sentence basis, coding up customized search algorithms to help construct a print index, and running searches to locate stretches of text that should be subsumed under tags (or other specialized object) are a different category — requirements that could well apply to many or all of the documents typically handled by publishing software.

The presence of textual elements that do demand special treatments may be notated by conventions like domain-specific XML tags, but actually implementing the cross-component interop needed to leverage them almost certainly requires programming over textual content as well as data-set materials as a whole. In other words, these are concrete User Interaction features that illustrate the principle that text content (including objects modeling rhetorical/discourse structures) is now situated as a computational artifact among variegated information spaces, like fibres of a differentiable manifold.

2.2 Sentences are the Preeminent Units of Discourse

I will assume that the basic unit of intrinsic text representation is a single sentence — even if this is not how most encoding standards are designed. Among common formats, only BioC and TEI appear to employ distinct sentence tags, and even here these are optional — thus, documents in those formats may have `sentence` or `s` elements (respectively) but there is no guarantee that documents will identify textual boundaries with that level of granularity. The far more common approach is to model documents as series of paragraphs.

Nonetheless, it seems obvious that written text is fundamentally a stream of sentences and that paragraphs are a separate, aggregative layer. So, the "base" layer for text-representation is that of individual sentences. Here I am concerned not with how text is recorded in an encoding

file — e.g., XML or TAGML — but internally as a computational artifact, which we might reasonably assume exists in an Object-Oriented context. As such, text content (however conveyed via markup) should be conceptualized first as a collection of sentence objects. From that basis we can then model intrasentence entities (e.g., keyphrases and annotatable word-sequences) and larger-scale units (paragraphs, sections, etc.).

Characterizing inter-sentence relationships is a little subtle, because *usually* sentences follow one another in a straightforward sequence, but sometimes documents can take on more of a hierarchical character. For instance, a footnote may contain multiple sentences, each in a sense "nested inside" the sentence where the footnote marker appears (although conversely footnote contents may be a clause on a scale smaller than that of single sentences, akin to a parenthetical comment). Similarly, all the sentences of a block quote might be considered "children" of the sentence that introduces it (which in turn, potentially, ends with a colon — or no punctuation at all — rather than the usual period or question/exclamation mark). In short, sentences are not *directly* nested in others, but they can have containment relationships through intermediaries like footnotes and block quotes.

Note also that written discourse has constructions that go beyond just transcribing verbal speech. Footnotes, block quotes, citations, parenthetic material, and special organizational patterns outside normal paragraph flow (e.g., enumerations and bulleted lists) all embody conventions of published manuscripts that have no direct correlations in spoken language. From the topomorphic perspective, such details inform how we model text content as a "space" to be navigated within and traversed across. This, in turn, is dependent on the goals of algorithms that operate against textual objects.

An obvious class of procedures are those which attempt to locate matches to query strings (typically words or phrases). In the general case, it is reasonable to assume that successful hits will lie on the interior of single sentences (rather than straddling sentence boundaries) and that query results will be a list of sentence objects where the desired phrase is found (perhaps augmented with PDF page coordinates for the sentence as a whole or the page-area spanned just by the matched text — which is how results are typically presented by PDF libraries' own search routines — assuming PDF coordinate data is available).[14]

In the absence of any restrictions, every sentence in a document should thus be checked, with potential branching to account for "nesting" relationships (as with footnotes and block quotes, potentially) as well as divergent possibilities for "next sibling" (consider a footnote marked just after the end of one sentence: we can visualize the discourse as continuing on two tracks, one within the footnote and one continuing the main text.

In some cases, however, whomever initiates a search might explicitly narrow the scope to exclude or include specific areas of text. They might filter searches to consider only main document content, not (say) footnotes or endnotes; conversely, perhaps they *only* want to search within footnote lines, quotes, bulleted content, etc. From a topomorphic perspective, the relevant details here are not so much the explicit "tags" or other markup via which special segments like quotes and footnotes are introduced, but rather the distinct varieties of textual material that are woven together into an overall document. The property of *being* a footnote, block quote, inline quote, citation, etc., is a *characteristic* of text subsumed under the relevant element, and users restricting query scope should be understood as narrowing focus to content with one or more specific *facets* (not specific *tags* per se). This can apply to presentational features as well, such as italics or sans-serif. Such restrictions might come into effect if a user is seeking to find a passage they had read earlier, for instance, remembering certain words that were present and recalls some of their styling details.

In a written context, moreover, some words or phrases have semantic and/or structural properties that can be modeled apart from just notating the underlying verbiage. For instance, JATS/BITS has tags specific to personal names, and their presence in article text allows search tools to disambiguate proper names from their homonyms ("Robin Hunter", an actor, might also be a description of someone bird watching, or a predatory cat). Visible citations should be backed by structured data allowing the current document viewer to search for the referenced resources. Quotations, similarly, follow conventions with ellipses and bracketed paraphrasing that have to be accounted for by algorithms for searching and related tasks (should bracketed words be included if they permit some string to match a query, and if so, how should their special status be notated in the match-result context)? Meanwhile, specialized segments such as chemical formulae, genus/species names, phonemic transcriptions (for linguistics papers), etc., might need both distinct representations and customized treatment for query matching.

[14] The page-view rendering engine would then take area coordinates and apply styling — usually background shading — to visually mark off search matches. The complications arise when we intend to override the internal PDF search algorithms to match against encoded text instead (given the problems of PDF queries from hyphenation, ligatures, etc.), as well as inability to restrict matches to such contexts as quotes or footnotes. If users expect highlight cues and a search runs against embedded or sibling text-encoded assets, then matches in the latter context have to be correlated with the former. In special cases, however, PDF page coordinates of preselected items of interest (keywords, index entries, etc.) can be preserved via mechanisms such as LaTeX pgfsavepos.

It is similarly non-trivial to define the scope of individual letters and symbols. Conceptually, a reasonable assumption is that every character in a document (at least for English and most other languages) can be reported as a two-byte UTF-16 code point (albeit maybe *stored* with *one* byte). But, expressive though it is, Unicode is arguably flawed in the context of representing glyphs that are visibly identical but functionally distinct, such as end-of-sentence periods in contrast to dots for abbreviations, acronyms, decimal numbers, etc.; or, say, apostrophes versus right-single-quotes. This may be a case for Unicode Private Use areas, to ensure something closer to a one-to-one match between symbols' code points and their discursive roles.

Such details of *internal* representation, furthermore, are distinct from user functionality like copy-paste or PDF annotations. For example, consider requests to copy the current sentence to clipboard. Presumably the user then intends to paste those contents into some other location, but that could be many things — LaTeX, MS Word, email, a note-taking program, etc. With no separate specification, "exported" content for things like copy/paste should be as simple as possible,[15] perhaps even restricted to Latin1 (so that "curly" single and double quotes, say, are replaced by their straight and non-chiral versions).

On the other hand, multiple forms might be present in query/search strings. For example, suppose a desired match is one single word enclosed in double quotes. Quite plausibly the "input" to that query could have either the straight/ASCII form (if the user is typing from a keyboard) or the curved left-versus-right forms (e.g., if they've copied from some other document). Internally, meanwhile, the encoded text elements might not use quote delimiters at all, but rather some form of quotation object. Thus, testing whether an input string matches text/discourse objects at a particular point in the character-stream is more involved than simply comparing numeric glyph-codes. Likewise, a text-query engine should assume that results might be requested in a variety of formats (Latin1, UTF-16, HTML, etc.) such that how text is conveyed, in a variety of potential reporting protocols, becomes data for each text element (e.g., sentence objects) that might deviate from how such content is internally represented. Similarly, how should results note the presence of footnote markers; of citations (in the form of numbers in square brackets, for instance); of ellipses and emendations for quotes; etc.?

In short, there are structural differences between *query input* formats, *internal* (object-based) encoding, and various result *output* formats; and nontrivial algorithms might be needed to map both the first to second (and evaluate potential matches) and the second to the third. All of these details should be considered by a VM whose roles include serving as a compilation target for a text-query language. Suppose that all discursive content is stored internally as C++ objects, and that methods on those objects handle all the complicating factors related to whether a specific part of a given sentence (canonically) is a successful match to a query string. Robust query-evaluation algorithms would presumably encompass a variety of parameters/flags and protocol variation, considering the range of discursive contexts which have somewhat different match-evaluation characteristics, both vis-à-vis fairly canonical writing conventions (block quotes, citations, etc.) and special-purpose symbols, formulae, or character strings that might be commonplace in a particular academic/scientific discipline. A VM could be engineered such that calls to a few general-purpose C++ methods are reified as built-in bytecode instructions, but such calls would need to be preceded by steps to initialize all data relevant for the C++ evaluative procedures.

Meanwhile, a text-query VM might have opcodes specifically focused on navigation between/within sentence objects and related text elements. Defining a rigorous "topomorphic" model of written discourse's structures is a non-trivial task because of complications to the straightforward sentence-sequence norm, such as how the words directly preceding a block quote appear to form the first half of a sentence which is in turn completed by the beginning of the quote: should this be modeled as one sentence or two? Can sentences straddle boundaries between ordinary text and special areas like quotes, or enumerations? Here I am not interested in concrete answers to these questions, but merely the more abstract observation that these are the kinds of issues that have to be resolved for a "topomorphic regime" supporting traversal across text documents, with a detailed model of what constitutes a single discourse "site"; what data is immediately accessible from the site-neighborhood; and what criteria come into play when navigating from one site to another.

In a Research Object context, a VM could be applied both for text-search queries against documents and queries in general (or scripting) for data sets. In this sense, apart from topomorphic capabilities specific to the structures of written discourse, the VM's operations might also navigate through various raw data formats, depending on how the Research Object is encoded. VMs, that is, may need to be adaptable for multiple kinds of "topomorphic regimes". But algorithms specific to text searches may also be applied to these different data formats, insofar as a query might be applied against data entities (column labels, text which is an individual field for some record) and/or computer code. Questions about character encoding and converting from query-input formats are relevant to text-search in this more general context as well as against document manuscripts specifically.

[15] There is no guarantee that the program wherever the copied content is pasted will understand anything other than alphanumerics and basic punctuation.

With that said, I will defer further discussion about text searching for now and pivot to an examination of Research Object VMs in more general data-encoding contexts.

3 Building a Self-Contained Compiler and Virtual Machine Runtime

For reasons already discussed, the kind of VM relevant for this paper would be designed for an Executable Research Object context and, in particular, an environment where most or all components are distributed in source-code fashion. Special attention is therefore necessary to ensure that a technology stack sufficient to support (at least modest) scripting and query-language capabilities can be developed in an accordingly standalone fashion, with limited external dependencies, and including compilers for front-end code and whatever logic is relevant for processing and transforms applicable to intermediate representations. For obvious reasons, these requirements preclude heavyweight Software Language Engineering environments such as LLVM or the JDK.

In the landscape of programming languages, a few are mostly self-contained in the sense that installing the language from a source package is relatively straightforward, and does not require users to first have something like LLVM or libClang on hand. Two examples that I am familiar with, and which might be relevant as case-studies, are ECL (Embeddable Common Lisp) and AngelScript [16], [11], both of which seem relatively easy to build from source. These languages still are slightly different from what I envision here, partly because — despite being well-designed from the perspective of (and to the extent that such is desired to be begin with) the languages in their totality, as computational artifacts (i.e., the full stack needed to execute scripts, with compilation and bytecode interpretation as necessary) are comparatively straightforward to compile — these components are not primarily intended as providing scripting/query layers for other code which is *also* distributed from source. For whatever reason, almost surely ECL and AngelScript (and other lesser-known languages with similar benefits) are not utilized very often for code supporting (and packaged within) data sets or Research Objects.

Moreover (and granted this is a more subjective point) these languages do not appear to be implemented in a fashion that is explicitly oriented toward customization and adaptation for individual data sets. The kind of compiler and VM I consider here might well have special instructions and/or front-end syntax rules that are unique to the specific research being published, such that front-end parsers, IR and bytecode formats, and native-function interop are designed to be extensible in a relatively straightforward manner. This is admittedly a speculative issue without a compiler and VM tested in real-world context, but at least by *intent* the concepts I propose here yield architecture and paradigms that offer more accessible "hooks" for customization in contrast to the internal code for languages like ECL and AngelScript.

In this discussion I will assume that the language and any components it interacts with are written in C++. This is partly because the prototype compiler and VM I will occasionally refer to are written in C++, and partly because I assume Executable Research Objects will in general include certain GUI components (potentially including PDF viewers themselves) and the most straightforward path to effective cross-platform GUI applications is via C++, particularly in conjunction with the Qt libraries. At the same time, as far as I am aware, very little of the implementation details I discuss involve techniques specific to C++, so it is entirely possible that similar VMs, potentially executing the exact same bytecode, could be programmed in other languages (Java, JavaScript, etc.) — which of course would be more conducive to some use cases I identified above, such as cross-platform deserialization.

To be clear then, I envision Executable Research Objects where *both* an internal scripting and/or query language and *also* major GUI and analytic components are coded in C++ and distributed within the data set as source files. This need not entail everything built from scratch; instead publishers or grid collaborations could provide a suite of mostly-usable classes which authors (or programmers on their behalf) may customize for their own data publications. Nevertheless, if each Research Object has its own specialized C++ code, it can be unclear what role a VM would play. This is not analogous to large-scale scientific applications that embed a scripting interpreter for customizations, like Python in the context of ParaView or ArcGIS, JavaScript vis-à-vis OpenCV.js, or C++ itself (as an interpreted language) in CERN-ROOT. Clearly the entirety of ParaView, ArcGIS, OpenCV, or ROOT (not to mention LLVM and Clang for the latter) and all their dependencies cannot feasibly be included within a single data set. These are conceived instead as tools scientists install (probably from binary) and use on many occasions for their research work. Published investigations could well be conducted through use of the same software, so for assessment or reproduction other scientists might indeed wish to work in an environment similar to how the stated findings originated. The relevant platforms therefore embed script interpreters to handle the relatively modest fine-tuning that might be needed to sync the research tools of the authors with those of potential readers (at least, those seeking to go so far as replicate or analyze the published material in depth).

While acknowledging such scenarios as entirely proper and common, my point here is that there are other purposes for research publications wherein avoiding dependencies on large, external platforms is desirable. It is preferable for

an interested reader to examine research on Image Processing, for instance, *without* needing to install OpenCV, or for scientists to assess multidimensional data spaces without having ParaView (or comparable software) on their computer. This can be achieved if Research Objects provide more light-weight, perhaps scaled-down, but functionally similar components (as source code) to the tools offered within larger platforms (ParaView, ROOT, etc.).

But with that said, having to micro-customize scientific C++ code can still add extra development time to a data-publishing project. Again, I am envisioning data sets including C++ classes that are derived, at least in part, from libraries intended to be used for many different projects. Research Objects should avoid *depending* on external software such as ParaView (though *facilitating* its use is fine), and this may be achieved by leveraging simpler components for tasks such as interactive 2D visualization (based on QtCharts and/or QGraphicsView, for instance). Such components may take the form of a class or "widget" library made available to authors and readers by publishers, book series, journals, grid maintainers, etc. (similar comments apply to digital cartography GUIs in lieu of ArcGIS, image graphics to interactively demonstrate processing steps originally executed via OpenCV, and so forth, just to reuse those examples). If the components are included as source code within data sets, it is possible for all project-specific customizations to be implemented simply by modifying the relevant C++ code. Nonetheless, adapting them to a given data publications may be less time-consuming if done via a scripting or query language explicitly built for components like those being fine-tuned.

I therefore see numerous use cases for a VM to buttress scripting or query languages that express customizations that would be more complex to achieve via manually editing C++ code itself. At the same time, computationally intensive routines — or simply ones that for stylistic reasons are better suited for it — might still be coded (specifically for the data set) in C++. One consequence is that interfacing with C++ should be easy. Making calls to C++ procedures (and marshaling input/return arguments) should require less effort that evident in languages like Python, JavaScript, or Haskell, which serve more as generic languages in their own right.

Conceptually, a "foreign-function interface" (FFI) to C++ procedures is relatively straightforward. The called procedures themselves can be identified via function pointers (in the case of pointer-to-members, a trivial proxy class may stand in for any method-declaring type; i.e., supplying the left side of a *CLASS::*METHOD-PTR* construction). Methods and procedures can be called, in general, through type-agnostic function pointers so long as the byte-lengths of all arguments match what the procedure expects.[16]

Details like mutable references, exceptions, the presence or absence of return values, and static versus object methods complicate the picture a little, but still for *most* C++ procedures it is possible to compress all data needed to call the function indirectly (through a pointer) to a single two- or four-byte integer, which (for the lack of an agreed-upon term) I will call a "micro-signature". "Registering" a C++ procedure with a VM/script engine thereby reduces to providing a name (potentially including class and namespace scopes), function pointer, and the micro-signature code (by contrast, note that the analogous FFI step is noticeably more complex in ECL and AngelScript). The code library published along with this paper demonstrates how micro-signatures map to steps for arriving at the action function-call, via a series of switch blocks — in this context AngelScript defers to inline assembly code, and other languages fall back on "smoke" bindings or other pre-processors, or else load in a large dependency such as **libClang**. So the approach described here, by comparison, is more aligned with the idea of a compiler and VM being both standalone and (fairly) easily customizable.

Some of the demo code which works off of micro-signatures was generated automatically, and it does have the limitation that not every semantically meaningful micro-signature code "works" with the switch-branching mechanism (this is a necessary compromise, because too many switch cases can make a binary uncompilable). The goal however is that *most* C++ procedures that a data set would want to expose to script engines have signatures that reduce to a supported microsignature code. For those that do not — considering that the Research Object is expected to have some C++ code of its own — it should not be difficult to implement wrapper functions. Overall, any collection of exposed procedures can be grouped into equivalence classes via their micro-signatures. If two procedures have the same code, they can called via the same cast of a function-pointer (or pointer-to-member). Wrappers are needed only for those functions whose codes do not fall within any of the supported equivalence classes.

Another idiosyncratic aspect of the demo VM is what the code calls "channels". The goal here is to design an

[16] A terminological note on words such as "procedure" and "function". In most contexts I prefer the former, because there is less confusion with the more mathematical sense of "function" as a value-to-value mapping. Conversely, when discussing language implementation and specification for the likes of C++, "function" is the more common term. I believe a useful contrast is as follows: "procedures" can typically be understood as functions with implementation bodies actually written in source code. Note that functions sometimes have algebraic relationships wherein they may be considered individual values: the composition **f**∘**g** means to call **g** first, then **f** with its result. Here "composition" acts like a binary operator on two function objects. For languages that support similar notation, certain functional values could be expressed without an implementation body on their own. This leaves open the possibility of construing functions as a superset of procedures, where the latter are always written out in code while the former may be designated indirectly.

abstract/generic model that could be adopted to different calling conventions and ABIs relevant to the VM environment, both the host language (which, in the general case, is not necessarily C++) and calls to procedures implemented via the VM itself (i.e., as opcode/argument sequences). Instead of a single program stack, procedure inputs and outputs are instead split into a series of channels, each with specific roles and semantics. Every call is performed through a "channel package" that models the transformation of values (and references) present at the call-site to parameters of the callee. A "stack" of channel packages plays a role akin to stack frames, but in contrast to a basic push/pop stack for machine code these packages are (by design) much less optimized, and can take a lot more memory (this is deemed acceptable because scripts for this kind of VM are expected to be relatively small and have only modest resource usage).

Although other use cases will be mentioned later, the main purpose of channels is to supply a larger perimeter of "hooks" which observer code can identify (and maybe inject other code) either via static analysis or at runtime. For one thing, channels have names, and the VM places no restriction on the strings chosen for these. As an example, a separate channel can be allocated for "this" objects (which the demo code calls "sigma" after the sigma calculus) such that calling conventions could be implemented (and honored) where one may have two or more distinct "message receiver" objects (use cases would be multiple-dispatch across multiple arguments, as well as overloads for binary operations). For another example, a separate channel could be named for result values returned from *constructors* as compared to ordinary procedures. This would be a way to focus specifically on constructors as the point of origination for any values handled by a program, in contrast to already-initialized values simply passed between procedures.

More generally, the channel system yields well-defined execution points pertaining to phenomena such as value-reassignment (for mutable variables) and type-casts. Channels are technically deques (double-ended queues) wherein values (called "carriers") may be pushed in front or from behind. The operation to do so is an execution point that can involve "effects" such as a value being read (possibly relevant for race conditions) or cast (relevant for narrowing type-conversions). The channel's names, value type(s), variable names, and their relevant lexical (or global) scope are all pieces of information that could be used to isolate these points in particular, and potentially install runtime handlers or monitors. It is worth noting that the VM "instructions" are themselves actually calls to ordinary C++ methods, which can be extended, reimplemented, or debugged (or "breakpointed") like any other C++ procedure. This gives the VM greater operational transparency than other scripting languages, I believe.

Another purpose for channels is to support a wider range of inter-procedure communication behind the scenes, some use cases for which will be discussed in the next section.

3.1 Proof-Scope and Contract Overloads

As discussed, one purpose for scripts within Executable Research Objects is to document analyses and data structures pertinent to published investigations. The resulting programs are not necessarily exactly the same as originary code, but they can introduce or summarize that code for illustrative or pedagogical reasons. To put it differently, the demonstrative code should foreground details such as preconditions, type invariants, units of measurement, and value ranges, whether or not these factor directly into algorithm implementations. The code serves not merely computational ends but also expressive roles as concrete tools for human examiners to understand theoretical models and programming techniques visible through the sources.

There are multiple ways of handling such details as algorithm preconditions, including "assert" statements, boolean checks performed prior to a procedure entering its primary logic, exceptions, comments, and code annotations. Insofar has code has a documentary/pedagogical role, precondition requirements (and/or axiom/postcondition when applicable) should potentially be handled in a more structured manner via (for instance) typestates (subdividing types' extensions into functionally distinct regions), decorations (where details like scales and units of measurement become explicit parts of a class interface, so that procedures expecting values scaled to one type of unit cannot be called with other types), and Design by Contract (DbC). In this section I will consider how VMs can facilitate scripting languages that incorporate these features.

Consider DbC in particular. The fundamental insight behind code contracts is that procedures' implementation bodies will often make specific assumptions about their parameters (or other relevant values) that are more detailed than ensured by data types alone. For example, writing to a file only possible if the file exists; and dividing by an integer is only possible if it is non-zero. Some invariants are extensional properties of types themselves (for instance, a signed one-byte integer can never have values above 127 or below -128). However, many properties of interest are true only of *some* instances for a given type. In most file-system libraries, say, there is no separate type specifically for files that are known to exist (i.e., their path is checked against the local directories); and integer types almost universally include zero as a possible value. Preconditions such as demanding existent files or non-zero numbers cannot, then, be enforced simply via procedures' type signatures.

For another concrete example, consider a procedure whose role is to initialize a "double-associative" array or "reversible map"; that is, a tuple of pairs that uniquely map each key to a unique value, so the association may be inverted (keys can be obtained from values as much as vice-versa). To build such a container, one must naturally start with two lists of instances, the first corresponding to the key type and the second to the value type. For invertibility, the two lists must be of equal size. The most direct implementation — uncluttered by sanity checks — would simply assume this as a precondition.

Procedures creating a double-array might therefore take two lists and moreover stipulate that the lists are the same size. One question is then how to enforce this. Such a procedure could simply check the two sizes and refuse to continue if they are unequal; the question then is what to return (perhaps an empty map, but such would also be the result of a call with two empty lists, and in that case the same-size requirement is technically satisfied). The procedure could throw an exception. Or, it could simply assume that the lists are indeed same-sized.

Explicitly checking preconditions via **ifthen** tests is better than blind faith, all else being equal, but that convention does violate "separation of concerns": arguably, checking that arguments agree to stipulated contracts is a separate process than actually carrying out the procedure's computations, and should be factored into a different function. In other words, some distinct code should perform the "gatekeeping" role of rejecting contract-violating inputs, and the visible separation of that logic from the main subroutine-body can help one understand and maintain the code.

Steps to guarantee that contract-annotated procedures are *only* called with preconditions duly satisfied can be taken either at runtime or compile time (or both). In general, compilers can assess whether it is *possible* for a contract to be violated at a given call-site. If this *is* possible, then the compiler can insert an extra runtime call that serves as a "gatekeeping" check. If it is evident that a contract necessarily *cannot* be upheld, then this could be a compiler error immediately. Otherwise, if the compiler can ensure that for all execution branches, external states, or value-combinations a contract is *guaranteed*, then the compiler can emit code for the call without further intervention.

A related question is how to handle execution paths when we get to the point where it is established that a precondition is not honored. The most common approach appears to be falling back on a language's exception mechanism, or something similar. For example, recent C++ contract specifications imply that the typical results of a contract violation are either calls to a "handler" (which might implement some alternative sequence that avoid depending on the precondition givens, or potentially throw an exception) or immediate program-termination [2]. Structurally this is not too far off from the handler being a *de facto* "catch" block that could either rescue or propagate an exception. As far as I can tell, these semantics are similar to those employed by most DbC supporting languages.

This all seems wrong. I believe a more intuitive framework would employ what might be called "overload by contract", in which the presence of absence of violations serves as state-variations suitable for overloading. I hesitate to claim definitively that something along those lines is indeed rather counter-paradigmatic in programming language design, because its omission is puzzling, which might signal that I'm missing something. But in any case, a VM use case I wish to examine here is to support "Overload by Contract" (ObC) as one version of DbC.

Consider again the double-array example. At any point in time, there are essentially three possibilities: we might have enough information to confirm that the list-size precondition is satisfied; or, we might know that it is *not* satisfied; or, that status might still be unresolved. At the moment when we do need to call the relevant procedure, conceptually (and perhaps literally) we are in effect overloading on that information-state. In other words, the procedure needs three versions. If the contract is violated, there are several options (short of exceptions or termination). The result could be an empty map, or the code could scale the longer list down in size to match the shorter, or supply default values for unpaired keys (or vice-versa).[17] A VM compiler and runtime should not attempt to guess which approach is better for a given program. Instead, it can stipulate that whenever a procedure is annotated as governed by a contract precondition, it must be paired with an implementation that addresses violations. By analogy, when inheriting from an abstract base class, all overloads of virtual methods have to be implemented for the derived class.

On the other hand, if we are in a program-state where the precondition is not known to either pass or fail, an obvious solution in general is simply to test it, and then call one or the other "overload" in response. In a same-size list case, logic for that test is fairly straightforward. In other cases, it may be more complex, and the compiler might insist on the code-author providing an explicit implementation for the test. That test function would then be analogous to an entry-point for multiple-value dispatch, returning some code that indicates which of several possible follow-up procedures should be called.

Continuing the double-array case, suppose the main procedure is called within the scope of a statement-block wherein the size of the two lists is explicitly checked. If

[17] For the map to be invertible, these cannot just be default *constructed*.

they evaluate equal (as in **x.size() == y.size()** or similar) and no intermediate statement could alter them, then the compiler may safely choose the version of procedure governed by the contract.

We should avoid, however, designing a compiler where too much subtle reasoning is expected. For one thing, compilers in this lightweight context should be as simple as possible. For another, it is helpful for preconditions to be explicitly notated in code — both at call-sites and within signatures — rather than enforced by invisible machinery behind the scenes. In short, I advocate for more transparent approaches than formal proof systems whereby pre- and post-conditions are chained as if computing over states and effects (wherein, for instance, the length of an array is part of its type, and any operation to add a value increases the length by one and thereby changes its type).

In practice, then, we should not assume even that the compiler would recognize an if-expression like **x.size() == y.size()** as testing the exact same state of affairs that is reported necessary for a contract. Instead, accept the notion that programmers need to hold the compiler's hand somewhat, both to make assumptions visible and to simplify compiler implementations. In the present context, one way to achieve this is via what might be termed a "reified boolean". This term hopefully connotes how the result of **x.size() == y.size()** is not just a boolean true or false, but a more substantial object that *evaluates* to boolean in the first clause of the **if**/**then** expression, but also persists as an object in the subsequent code blocks (whether the true branch or, if present, a false one). I assume that some added syntax is available to indicate where programmers intend apparent boolean expressions to evaluate not only to a boolean return value but to a "reified" object in this sense.

When an **if**/**then** construction does involve a *reified* boolean I assume this object remains "in scope" for the associated code blocks; and that similar branching or loop formations could have similar conventions. Conceptually, **if-then-else**, **do-while**, etc., are not necessarily special constructions in the language; they could possibly be expressed as ordinary procedures, taking code blocks as arguments. In this case, the language could have a mechanism such that procedures which *call* code blocks have the ability to attached reified booleans to them (by analogy to how a closure captures variables from enclosing scopes).

The notion of "channels" also comes into play. Conceptually, channels capture data from a call-site scope. If a procedure declares its reliance on or assumption of a contract — e.g., two of its parameters having equal list-size — then in the current context I interpret this as requesting a "proof" object to be passed in through some specific kind of contract-verification channel. Obviously that channel would not "look" in code like regular input arguments, but a compiler could observe the presence of that channel in the callee signature and endeavor to populate it with objects in the current lexical scope. A reified boolean such as one passed into the scope by an **if**/**then** test could serve this purpose. That object would need to be declared and initialized in such a manner as to clarify that it does indeed prove the contract's validity, which depends on both the proof "type" and the values involved. For example, the relevant proof-type here indicates that two lists have the same size; this information, together with pointer values or some other confirmation that the lists in question are indeed the ones that factor into the procedure's preconditions, would supply the data requisite for a compiler to recognize that the specific proof object (here a reified boolean) which it finds in the caller's scope is a genre of proof object that matches the callee's contract.

Notice that operations performed in the same scope as the proof object could render the proof invalid; if, say, new values are added to the key-list but not the value-list paired with it for the double-array. In principle, one might imagine the compiler trying to prevent such invalidation from happening. That could be achieved simplistically by refusing to allow any call that could modify values involved with the proof-object in any way (in other words, permit only calls by constant reference while a proof object is in scope). A more refined approach is to reject only mutable calls whose effects are (or include) changing the size of one or the other list. But one might certainly imagine cases where, although a given method *might* change a size, it would do so only in circumstances which the programmer knows do not apply in the current scope.

A reasonable alternative is to assume that a proof in scope will remain valid through the duration of the scope. It is then the programmer's responsibility to avoid conflicting calls. The point here is not to perform all analysis on the programmer's behalf, but rather to make the proof object a visual part of the source code and thus a reminder of contract obligations that must be fulfilled.

There are also cases where a codewriter might be confident that the kind of proof embodied by a reified boolean applies even in the absence of a test like **x.size() == y.size()**. If the second list is obtained from the first by something like an **std::transform**, they will be guaranteed to have the same length. Likewise if they are constructed from initializer lists that in the source code obviously line up, or built by pushing values pairwise. In these cases, the called procedure might still be looking for a proof in scope. Here that proof would not be supplied by a reified boolean, but the programmer could explicitly declare such a proof as invariant in the current scope.[18]

[18] And, if desired, the expression which achieves this could, under debug modes, also serve behaviorally as an assertion.

This arrangement — coupled with an Overload by Contract convention that procedures requiring contracts are overloaded in several versions (at least, those for pass, fail, and testing of a relevant contract) — may not be the most powerful engineering of a DbC system, but I think it makes the compiler's role straightforward — how to issue the relevant VM instructions for contract-enforcing logic. I believe this serves as an example of how channel-based calling conventions provide a roadmap to guide compilation steps that might be convoluted otherwise.

3.2 Variable Lifetimes and Memory Management

Another situation where channels can have a similar role comes into play in cases where procedures return values that are too large (in number of bytes) for return-by-value semantics.

Consider return and/or initialization options in C++, on the assumption that we are not just returning by value, due to size concerns. One possibility is to allocate a pointer in the callee, ownership of which is taken over by the caller. Another option is for the caller to pass a value of the relevant type (presumably trivially initialized) by mutable reference, trusting the callee to (fully) initialize it. One limitation of the second alternative is that it only works with types that can have a default or preliminary initialization, with their "real" values (or internal data fields) supplied later. This cannot be assumed for all types.

A more exotic solution that circumvents the issue just noted is for the caller to allocate a buffer with the proper memory-size and the callee to initialize *that* via placement new. The memory itself could then come either from the stack or the heap, whereas there is no way for a callee to allocate values on the *caller's* stack. Unfortunately, there is no option to implement this kind of arrangement in ordinary-looking C++ (this mechanism is similar to how compilers implement Return Value Optimization).[19]

Which alternative is most convenient from a caller's point of view? Pass-by-reference is a common C++ pattern: simply declare an object, which (assuming it is a class instance and not a primitive type) will trigger calling to a default constructor. The only imperfection here — apart from cases of non-default-constructible types already mentioned — is that there is a visible separation between the point of declaration (which is also a minimal initialization) and the "de facto" initialization effectuated by the called procedure.

[19] For Return Value Optimization (RVO) see [14] or [21]. Readers might be interested in an overview of user-visible RVO-related techniques that I present as a C++ code excerpt at https://cppinsights.io/s/cac923c9 which also addresses lifetime issues as discussed next.

This might seem to be a minor annoyance, but admittedly a form where a procedure returns a pointer, which is assigned to a variable initialized at that point, is closer to how the code is conceived: the purpose of the call is to initialize that pointed-to value. Initializing a mutable reference is behaviorally similar, but it less clearly documents the relationship between the callee and the initialized variable. Moreover, sometimes the caller *wants* a pointer return because the value should outlast its lifetime.

From the callee's perspective, the question of how to package its result is tangential to the main concern, which is to perform whatever steps and calculations are needed to fully initialize the sought-after data structure. This leaves open the possibility that a compiler could manipulate procedures in such contexts to support multiple interfaces, depending on the call site. That is, a procedure could be engineered to to work on a value/object *either* passed by reference, or constructed on the heap prior to the main body, or constructed (wherever the caller wants) via placement new. This idea essentially generalizes Return Value Optimization, which uses memory buffers behind-the-scenes to eliminate the construction and copying of temporary objects.

Apart from illustrating one use for channels, these points are relevant to the general protocols and style of a scripting language. Observe that passing an object/ value by (non-constant) reference can serve one of several purposes:

1. The primary goal of the callee is to initialize or modify that object
2. The object is intended mostly to provide information the callee needs for its main purpose, but cannot be made constant either because one of the callee's secondary roles might be to update some field in the object, or else it may need non-constant members even if in this case the object is unaltered
3. Modifying the object is a primary role of the callee, but a return value is separately needed (perhaps to provide information about the status of that modification or other side-effects associated with it)

Woven with these alternatives are questions about potential modifications: e.g., whether the callee is *guaranteed* to modify (as when a default-initialized object is in an invalid state). All changes are not the same: some may be necessary to render the object usable; some may be important to keep the object in sync with its peers or overall program state; others can be secondary and just represent a report on the various effects a procedure has triggered.

Likewise, consider a procedure that has certain side effects and then returns a value. It may be that the *primary* purpose of the procedure are those effects, and that the

return value is mostly for sanity checks. Conversely, it may be that the primary purpose is to compute that value, but side effects are unavoidable while doing so.

These various possibilities probably are not clearly demarcated, and it might be a subjective judgment as to what purposes or effects are "primary" and "secondary". Still, programmers and maintainers will reason along these lines when understanding source code. One might then argue that programming languages should use syntactic cues to help clarify these issues, more so than in C++, for example. For instance, with pass-by-reference there is no visible mechanism to distinguish objects that will *necessarily* be modified, or will be so assuming the callee returns normally, or instead only *might* be modified.

In C++, moreover, pass-by-reference serves as a workaround against both multiple-value-return limitations and reluctance to return large byte-sizes. In other words, often objects are passed by reference even if, behaviorally, the role of the callee vis-a-vis the object is more like that object being *returned by* the callee.

A scripting language — with more freedom to tweak its syntax — could do a better job of flagging its caller/callee semantics. For example, assignments (in C++, the left side of a basic operator equals) may be *initializations* (for a variable that, beforehand, was only declared) or else "re-initializations" of objects/values already in a valid state; there is no syntactic difference. A variable declared and then passed by reference will first be *uninitialized* (for built-in types) or *default initialized* (for class types). There is no visual distinction of the two cases.

A scripting language could be more profligate in using separate syntactic forms to differentiate these various semantic possibilities. In the case of pointer-return versus pass-by-reference (or RVO) one option would be what might be called "Janus" (metaphorically "two-faced") channels that (unlike the typical case) may be *either* input or output. For sake of discussion, suppose we coin the phrase "Janus pointer" to be an object that, depending on context, could serve as a pointer, reference, or non-initialized memory buffer. Here "Janus" is not some pointer typeclass but instead a formation placed inside a Janus channel. When assembling the "channel package" for a procedure-call, the compiler can check whether an *initialized object/value* or else a *memory buffer* has been pushed into the channel. If so, that gets passed as *input* to the callee (in the latter case the callee is orchestrated to call placement new prior to its main execution). Conversely, if a Janus channel is present but empty, the compiler expects the callee to allocate on the heap (again, perhaps prior to its running its actual implementation block) and push that into the Janus, to serve as a return value.

My overarching point is that, for procedures whose signatures include a Janus channel, their interface subsumes both pass-by-(nonconst)reference and pointer-return semantics. An important detail here is the lifetime for Janus values. Supposing an object is initialized within the preamble portion of an **if**/**then** or **for**-loop expression, one obvious question is whether the object is associated with the *inner* code block or the surrounding scope (and, hence, when its lifetime ends and destructors are called, if relevant). In C++, the specifics are forced by syntax. I envision languages where (perhaps with inobtrusive notated cues) programmers can request specific lifetime/storage-duration options in the context of procedure-calls that look mostly the same otherwise. For example, **for**-loops should be visually consistent whether or not a value declared in the preamble is bound to inner or outer scope; this is merely one detail among numerous points of attention (terminating conditions, iterators, etc.).[20]

Procedures' "interface" is more than just an implementation detail, furthermore, because at least some routines within a code library are "semantic" entities that should be searched and documented alongside other textual and data content. An Executable Research Object, we can assume, provides some mechanism to search over its variegated components — and perhaps an introduction or "manual" outlining its content. Among text materials, searching might include keyphrase or index-based queries over sentence objects, along lines discussed previously. For data, searches could be performed over individual records and (at least for table-like profiles) column labels (plus columns' types, units, etc.).

The analogous documentation for computer code would be searching data types, fields, procedures, and so forth. We can assume Research Objects will try to compile a sort of concordance listing (at least public-interface) procedure names, signatures, and pre-/post-conditions. Many projects automate this kind of step via (e.g.) Doxygen, but for data-set code a concordance should be more fine-grained. For instance, details such as functions' assumed units should be both notated in code and described in documentation in a structured manner, rather than relying on source comments.

One possibility is to run scripts at the point where the "executable" resource(s) of an Executable Research Object start up; scripts with the responsibility of building a search/documentation concordance, carrying more information than just declarative code comments (for example). Because the VM is designed to support formations such as contract overloads and special-purpose channels in the

[20] A good introduction to C++ lifetime issues — and solutions via "lifetime annotations" (often associated with the Rust language) — is [7]. I believe some of these issues can be addressed by offering programmers more control over lifetime termination, in particular allowing variables to be associated with one of several lexical scopes present at a given code point (their "retirement" aligned to their scope's end) or aligned with sibling variables.

context of procedure-calls, it could recognize the concomitant annotations and incorporate them into search and documentation assets.

3.3 Dependent Types and Template Metaprogramming

Constraints such as two lists having the same size, which here I have examined via contracts, are also studied in terms of dependent type theory. A type whose instances are lists of length *l* is "dependent" in that its specification depends on a *value* (by contrast to *meta-types* that depend on a *type* to be specialized, as in vector<T> where T could be int, double, etc.).[21] If the aforementioned *l* is a compile-time constant, then the relevant type may be called *weakly* dependent.[22] However, the running example of lists for a reversible map would involve a *strongly* dependent type if the relevant constraint is expressed via the type system alone (without the added machinery of contracts or something similar). Specifically, the type of the second list would depend on a length *l*, but the actual value of *l* is unspecified until the actual procedure-call, and derived from the list's sibling argument in the channel package.

It is *not* a matter of strong dependence to construct a type whose instances are *pairs of* lists having the same size. Here one would merely check in the constructor that this invariant holds, and ensure that any methods modifying the lists in the pair do not alter their sizes differently. In short, what makes the hypothetical procedure-signature with a second type strongly dependent on a first argument is not just the constraint semantics, but also that the two arguments are conceived as distinct parameters. With normal type systems, types express constraints on single values. A constraint on two distinct function-parameters is not typically understood as a phenomenon of either type, but rather a more holistic limitation involving the *procedure* being called (and so implemented via contracts or other precondition enforcement).

Expressing relevant constraints via "just" the type system is an interesting possibility because we then have a single framework for expressing a wide variety of preconditions, rather than dividing this into types (for single values) and contracts (for disconnected values passed to a procedure). However, reasoning through dependent types is not straightforward. For one thing, it is important to stress that a fundamental distinction exists (in the context of Software Language Engineering, if not mathematics) between types and their extensions.[23] The fact that a hypothetical value **v** would be one instance of a type **t** does not confirm that **v** belongs to **t**'s *practical* extension at any point in time (i.e., the values that are actually available for use by a program, given limitations with memory size, filesystem credentials, network traffic, etc.). In general, in computer code the *purpose* of a type is not simply to offer a shorthand expression for some extension-set. Rather, types' primary role is to enable function overloads. Function signatures are distinct if two parameters have different types, even if those types are extensionally equivalent.

In a language such as Idris, where dependent typing is built in intrinsically, types become essentially "runtime" phenomena not substantially different from values as such. This is an alternative perspective on type theory, and while it certainly has a formal foundation there are numerous non-obvious concerns that differentiate this latter kind of type system from (what we might call) "non-extensional" type theory. Once we fundamentally distinguish types from sets, any set-theoretic reasoning becomes more or less inapplicable in the former context.

Consider what happens when a dependent type is passed to a procedure — e.g., a list **y** of length 5. This is a *runtime* detail and therefore that specific value cannot be manipulated at compile time. Instead, one presumably would need something like a *variable l* that holds **y**'s size. Such *l* functions as a constant only within one execution of the procedure's implementation body; it works *de facto* like a parameter. The procedure needs to be prepared to handle any possible *l* value within its overall range (which in turn depends on the maximum legal size for **y**). Any dispatch or fine-tuning which relies on particular values for *l* has to be implemented in terms of runtime checks (if/else and so on). By extension, the same applies to **y**, at least in the context of behavior affected by **y**'s size.

However, if we end up treating this size as only a runtime detail, we aren't really exploiting **y**'s *dependent* type

[21] For the general motivation and theory of dependent types a good source is [6], which looks particularly at the Idris language — one of just a small group of relatively general-purpose languages to support them. An interesting look at potential applications for languages in general is [12].

[22] Although "strong" and "weak" sometimes appear in related contexts, my proposed usage is not the same. But in the general case — considering just types themselves — a dependent type is one of a type *family* associated with some other "index" type, such that each instance of the index corresponds to a unique member of the family. When a value in computer code is assigned a dependent type, then this general picture takes on different variations. If the actual indexed-typed value for a specific variable/symbol is itself obtained from some *other* variable, then the former also *depends on* the latter; aside from its type depending on some index-type instance, the information needed to ascertain this type depends on reading some external value. This extra layer of dependence can be referred to as "strong" dependence; these are usually the situations motivating dependent type systems. For example, C++ certainly allows one to declare templated types where (at least some) template parameters are values rather than other types; but those values must be compile-time constants.

[23] This is not true of all types; one has no issue equating the extension of a byte type with unsigned integers less than or equal to 255.

in the first case. In effect, any procedure that accepts an apparently dependent y in this scenario has to treat y non-dependently, as if it were actually assigned a more general type (e.g., the relevant list type without any size data). We can certainly obtain a runtime variable like *l* from, say, y.size(), but that's not doing anything we wouldn't do with an ordinary type system. Having such an *l* precomputed seems basically syntactic sugar. At best the dependent specification is ornamental, for documentation but not computational purposes.

One way to see this explicitly is to consider the procedure being templated in terms of y's type. In a VM context, we can assume that each different template instantiation yields a distinct sequence of bytecode instructions. So, for any potential type of y, the procedure will have a corresponding (distinct) bytecode block. Obviously, it would be infeasible to duplicate such blocks with thousands of copies. So the space of variation for y's type has to be restricted. If one still wants to conceive of y as having a range of possible types — comparable in cardinality to a range of ordinary values — then the specific runtime type y inhabits must be modeled as a runtime value. That is to say: within the bytecode for a template instantiation, there must be some dynamic symbol or holder representing the runtime granule of y's fine-grained type. But from a VM perspective we are entitled to argue that details represented via dynamic bytecode symbols are not *types* but rather some runtime data associated with an object/value. In other words, although a language could give these dependent constructions the flavor of a type system, details like y's "granular" characterization are not types per se, but rather a sort of annotation or characterization asserted relative to y.

None of which precludes developing something like a dependent type system, but the class of "dependent" types has to be recognized as fundamentally different formations than types themselves. In particular, in no construction would bytecode use a runtime value to stand in for some range of types (at least directly; indirection is of course possible with a pointer-to-void, let's say, reinterpreted to something else). Any typed symbol present in bytecode has one fixed type; if it is necessary to work with a range of possible types, the relevant code has to be templated and then compiled to different bytecode blocks, one for each "tapset" (we might say, meaning "type-attribution per symbol"). For purpose of discussion, I propose the term "type *facet*" for an (in principle) runtime type attributed to some value or variable, more precise (in the typical case) than its actual (compile-time) type.

The idea of using types to express runtime conditions is plausible because there may be some details that are guaranteed within the logic of one type but not another. For instance, any float (well, any rational/decimal) may have fractional part zero, and so be (losslessly) convertible to integers. The fact that a float f could also be int, say, is a detail whose logical form is elegantly captured via types. This appears, indeed, to be a contract or precondition, but a special kind. Here one way of stating the contract is that f is "compatible" with int in some sense.

But what sense? We need a systematic way to discuss how types themselves enforce preconditions (or invariants across their extension). One tactic is to consider the overall contract machinery, but apply it specifically to "constructors" (or factory functions — specifically, procedures that serve the purpose of constructing new values, whether or not they are literally constructors with associated special semantics in a language like C++). More precisely, in a channel system we can define a constructor in a general sense as any procedure with an output channel marked for a *constructor* role. Then a "constructor contract" could be any contract defined on such procedures.

But with this setup, we are free to abstract constructor contracts from constructors' implementations. Consider a type of all floats whose integer value is exactly n. The check for a constructor-contract in this case would presumably just test an f's being at least n and less than n+1. Insofar as some f passes the test, then the hypothetical type can serve as a *facet* even if no actual constructor is called. This facet is just a runtime value, and can be represented in bytecode as a dynamic entity whose specific value(s) are not known at compile-time (in particular, multiple instances are representable without some bytecode sequence being duplicated).

In short, some form of dependent types could be allowed if they are implemented more as value-annotations consistent with templating mechanisms (given an "instantiation equals block" paradigm) than as *de facto* types attributed to their bearers. Not only does this approach clarify dependent-typing possibilities, it also indicates how template metaprogramming might be supported in a (comparatively lightweight) VM context.

Of course, I have implicitly made assumptions about the VM in question. Clearly the VM would work in general with full type objects and rarely if ever practice type erasure. Implementing templates via duplicate generated bytecode-blocks is not the only tactic. But, for VMs in the Research Object milieu relevant to this paper, these are reasonable assumptions.

4 Extensibility Vis-à-Vis Grammar and Intermediate Representation

Here I address some further details about how such VMs could employ robust type objects. This will lead to points about language extensibility, that belong to a broader issue spanning front-end parsers as well as back-end code generators.

4.1 Decorated Types and Axiation

Similar to contracts and channels, units and scales of measurement straddle implementation and documentation. The key goal in this context is to ensure that procedures are called with quantities scaled in a consistent manner; at the least, that two different function-arguments are scaled the same. These concerns are endemic to dimensional analysis and (for instance) SI units in a scientific context, but they also extend to areas which do not obviously employ numbers measuring such clearly physical magnitudes as size, time, temperature, etc.

Consider the case of 2D graphics. There are many (e.g.) C++ graphics and image libraries, and they are inconsistent with respect to what the horizontal and vertical axes are called, as well as which quantity comes first in a pair. is constructed either by naming two points (diagonally-opposed corners) or by giving one point and two scalar values representing width and height. QPoints are integer pairs (QPointF is floating-point instead) in **x**-then-**y** order. Here the dimensions are conceptualized as **x** and **y** axes; but in other contexts pixels are treated rather as "cells" as if in a table, and coordinate-pairs are in *row-then-column* (i.e., *vertical-then-horizontal*) order. In the case of QRectangle, one similarly must commit to memory that the first scalar is "width" and the second "height".

For both implementation and documentation, we can argue that aggregating magnitudes into coordinate groups is more rigorous than employing lists of individual scalars, so procedures' interface should nudge toward coordinate rather than scalar types wherever possible. For instance, "moving" a point some distance left/right then up/down can be notated by convoluting the point with a transform pair rather than disconnected scalars (as with QTransform, affine translations are one part of a larger transform matrix). Similarly, arguably a better rectangle constructor would take, if not two opposing corners, then a width and height merged into a single "size" parameter. If coordinates are just integer-pairs, there is no way to overload these two rectangle constructors, but that could be alleviated with pairs being explicitly decorated as "*width-then-height*" — a different type than "*x-then-y*". More substantially, we can consider a spectrum of distinct 2D coordinate types representing how horizontal and vertical axes are conceptualized: *x-then-y*; *row-then-column*; *first-then-second* (as with generic pairs); *width-then-height*; or *horizontal-then-vertical* (plus inverses of all these). The strings "**s4wh**" or "**u2cr**", say, might encode signed 4-byte *width-then-height* pairs and unsigned 2-byte *column-then-row* pairs, respectively ("negative" width or height could cover such cases as initializing a rectangle via its bottom-right rather than top-left corner).

These concerns in general address how coordinates project onto axes (they could be called "axiation" details), which is a separate matter from lengths' units of measurement: pixels, typographic points (1/72 of an inch), millimeters, window-area percentages, etc. Particular types can synthesize both coordinates and units details. There is no reason why a VM could not internally work with types along these lines, especially the sort of VM I am considering here which is far removed from assembly language and the physical substratum of computers (the instruction set is much different from machine code). For example, in the demo VM, channels are populated in general by first loading a type object and then pushing a value that gets associated to that object. Type objects may be fairly robust C++ structures that cover numerous internal details, including potentially a mashup of byte-length, axiation, and unit decoration ("**u4xy**-*in-pixels*" and so forth); or dimensional aggregates in the sense of mp-units and similar (e.g., boost::units) libraries.

A straightforward use case would be as follows: suppose a procedure-call is being prepared and two successive values are pushed to a channel, preceded by loading type objects. A check could be performed (either at compile or runtime) that the two type objects are dimensionally compatible (not necessarily exactly the same, since we might accept something like casting from two to four bytes for raw magnitudes). The callee might specifically declare that it expects types associated with a particular axiation, coordinate system, or unit of measurement; in that case, it could also be checked that the channel's type objects are compatible with the procedure. Alternatively, the callee might declare that its calculations are not affected by dimensional minutiae, but still reject channel packages wherein two parameters have incompatible coordinates.

Modeling coordinate details explicitly may avoid certain programming errors, but a further benefit is how dimensions become an intrinsic part of a procedure's signature, propagating to documentation, unit tests, GUI design, and so forth. For example, a Research Object whose multimedia contains videos, and with code included to play them, is intrinsically working with scales such as Frames per Second (FPS) or Hertz. Presumably, at least some internal procedures would thereby take and/or return arguments with analogous dimension, and so those procedures and their signatures would be relevant matches for searches given such terms as "FPS", "kilohertz", etc. Insofar as a code-documentation "concordance" is exposed to the package's overall query system, granular typing such a unit-decoration facilitates the integration between data-set code and the overall Research Object.

Similar points could be made about value-ranges. Annotations might declare that the value held by some variable (or reference, object-field, etc.) will/should always fall within some range (narrower than its type's overall ex-

tension). Channel semantics offer a roadmap toward the execution-points that have to be checked to guarantee such restrictions. For instance, the demo code uses the term "carrier-handoff" to refer to actions such as transferring a value from one package's "return" channel to another's "lambda" channel (which is the default for inputs). This would be a natural place for runtime "hooks" confirming a range.

In brief, in a channel-oriented framework *most* of a compiler's operations involve pushing values into a channel and then transferring control to a callee. That comment might sound underwhelming, because for normal compilers one could equally say that much of their work comes down to pushing and popping values to a program stack. However, channel packages are more complex data structures than a single stack, and they offer more sites for analysis or intervention, such as when outputs from one context become inputs to another. The VM instructions which are *not* focused on channels tend to be engaged with lexically-scoped variables and steps to enter and exit code blocks, including lifetime management and the proper moments to call constructors and destructors. In general, if someone wants to customize a compiler to perform extra checks (or schedule them at runtime), implement new kinds of contracts, etc., the extra code would generally apply either to channel-manipulation (especially points where values are pushed, carrier-handoffs, and validations executed immediate prior to a control-transfer) or to code-block operations (enter, leaving, and registering lexical symbols within a local or parent scope).

These comments apply mostly to *semantic* customization. They pair naturally with the issue of extending *syntax*, which I discuss next.

4.2 Language Grammar and Intermediate Representation

External tools like LEX and YACC make compilers less self-contained, so here I assume that initial parsing is performed via techniques that could be built into source code directly. In particular, I assume that language grammars are defined via regular expressions, which are matched against input files via conventional Regex libraries. Several of these exist for C++; within Research Objects employing Qt for cross-platform GUI, the QRegularExpression classes are adequate.[24] Meanwhile, most C++ regular expression libraries take ordinary strings as initializers for regex patterns; hence we can add features to a basic pattern language, so long as these are expressible via ordinary regex patterns that are computationally generated.

Here (consistent with the demo code) I make the following assumptions: first, all grammar rules are expressed in terms of patterns that are implicitly anchored on the current point in a left-to-right parser (corresponding to a "slash-A", i.e., \A, in Perl-compatible expressions). That is, no rule will "skip over" characters, and rules will not in general shift the "current" parse-point except via the matches they consume. Fallback rules may implement logic to skip over characters or parts of the input that really are extraneous (e.g., superfluous whitespace). Also, rules are declared and tested in sequence: the first rule whose regex matches at the current parse-point takes effect. More precisely, each rule is associated with a callback function (typically an anonymous procedure, like a C++ "lambda") and the first matching rule's callback is executed, pre-empting other potential matches (since all patterns are anchored at the same place, there is no need for a "ranking" or backtracking due to some matches being further forward than others). Finally, grammars are assumed to be context-sensitive, and it is possible to implement contexts and/or flags that enable or disable rules. A deactivated rule would be skipped over in the test-sequence.

Conceptually, the goal of an initial parsing run would be to construct an Intermediate Representation or "parse graph". Schematically, the match-text of the successful regex (or some capture group therein) becomes the basis for a parse-graph node, and the rule's callback is charged with constructing the node and adding it to the graph. That's a good mental image, but practical implementations may well deviate from this paradigm. This observation may be anecdotal more than theoretical, but by trial-and-error I've come to believe that implementations that first work through all parsing steps to build a complete parse-graph, before doing anything else, are more difficult to maintain than those which mix graph-construction with operations that, in theory, are logically subsequent.

[24] For the record, I'll add a few comments however based on practical experience applying QRegularExpression for this kind of grammar. One nice addition (which is not hard to implement via a string preprocessor for regex patterns) is to name common subpatterns and insert them (by name) into larger regexes (the point is not, necessarily, to create a subpattern *group* but simply to reuse any part of a regex expression that occurs multiple times in a grammar serving similar roles). In some places, also, variable-width lookbehind (rarely supported) would be nice to have. Another consideration is how to merge the functionality of capturing (via group) and positive lookaround [19]. Regular Expressions in a (one might say) "unfolding" grammar premised on rules anchored to one "cursor" position — which is continually advanced forward — have their own techniques, and one frequent issue is where to situate the cursor after a successful match. Normally this is directly after the overall match end, but that is affected by lookahead assertions; in general, it would be nice to control the post-rule positioning in regex patterns directly. Note that capture groups generally have two properties: first, all of their content is contained within the overall pattern match; and, second, in a system where you advance the "cursor" just beyond that overall match for the next rule, the new position will be further ahead than any group. But there are occasions where one or both of these results is not what you want. For example, you might intend to leave content for some later match (which is a purpose of lookahead, rather than just consuming the relevant characters) but you also want to "read" that content to handle the *current* rule.

It is worth noting here that the parsing techniques under discussion are considered both for markup and scripting/query languages; that is, they might be used for an input language to create textual documents as well as programs in a scripting or query format. Suppose that one intends manuscripts eventually to be shared as LaTeX files. Sometimes the desired output is evident immediately from a grammar rule, and waiting for some future graph-traversal step to produce such LaTeX code makes the overall pipeline more obfuscated.

For sake of discussion, then (this is the approach taken by the demo code) each rule's callback works primarily by emitting code for a kind of preliminary or preparatory VM. This is not the VM whose bytecode is the overall compilation target, but rather a kind of miniature VM whose operations are oriented either to building up a parse graph or emitting instructions of the "real" VM (or both). In general, particular nodes of the parse graph can be associated with lists of VM statements, and the eventual bytecode output is produced by walking these nodes and setting out the instruction in sequence. Prior to that step, however, middle-end transformers can be given a chance to manipulate and/or expand the graph by editing nodes or adding new ones. Conversely, the workflow might proceed by creating a "rough draft" of the eventual bytecode while also notating intermediate points associated with parse-graph nodes. Middle-end logic working off the nodes could then add material to the bytecode at those insertion points.

The bytecode statements themselves need not be very complex (again, we need not engage directly with things like x86 registers). Both for the "preliminary" and "primary" VMs a single instruction can take either a single value or no value at all. The instructions in general could be interpreted by a C++ class, such that each instruction maps onto a C++ method. For development and testing, the "bytecode" files could actually be stored as human-readable strings naming instructions together with characters representing the argument, if present (and irrespective of its actual type). That is, one version of the VM interpreter can be engineered to read two-part statements with a textual representation of the desired method and a string encoding of a possible single parameter (a string-to-method table can convert the instructions to pointer-to-member-function values). For optimized bytecode, of course, stringized method-names would be replaced by numeric opcodes (with a similar method-pointer table for the interpreter) and arguments expressed as byte-sequences rather than literal strings.

Given this setup, extending the VM should be conceptually uncomplicated. Developers could work first off the human-readable bytecode versions (which, admittedly, are not really bytecode in themselves, but the optimizations are automatable). New instructions may be introduced just by implementing new methods in the interpreter class, and adding the name to a dispatch table. New syntax leveraging those instructions can be recorded as custom grammar rules with their own callback handlers. New "preliminary" VM instructions may likewise be implemented similarly via methods in the pre-VM interpreter, and the rule-callbacks can work by emitting instructions for the pre-VM extensions. Custom channels might be employed to implement special-purpose calling conventions and ABIs. The kind of overall compiler/VM stack outlined here has, I believe, a relatively clear "roadmap" for where extensions and customizations could be woven into the basic code, with the effect that scripting and query languages could be adapted to individual Research Objects more fluidly than general-purpose standards like Python or JavaScript (or even fairly "insular" languages such as AngelScript and ECL).

4.3 Script-Enabled Parsing for Deserialization

The pure-C++ parsing system described above could be isolated from the rest of the workflow sketched in this section. In so doing, one option is to register a VM interpreter with the grammar directly, so that rule callbacks may invoke scripts or script-implemented functions instead of C++ ones. That is, some or all parts of a grammar could be implemented in a scripting language compiling to the relevant Virtual Machine.

This would be a step toward implementing deserializers via VM bytecode, where the VM in question is distributed as source code within Research Objects themselves. That solution would address the requirements discussed in Section 1, insofar as data sets could thereby obviate the need for external deserialization libraries.

As with scripting, query, or markup languages, for reasons touched on in this section, deserializers could proceed either by assembling all data (i.e., text strings) extracted from input files into a graph structure before then traversing this data to create runtime objects; or by responding to rule-hits in a more localized fashion (or some combination). In the former case, the algorithms are essentially topomorphic, and the serialization format could be modeled in terms of site and navigation criteria. Of course, if generic formats like XML or RDF are employed for encoding, the "topomorphic regime" is already implicit in the representation standard.

In any event, it makes sense for a VM serving these sorts of use cases to have built-in instructions pertinent to topomorphic computing, i.e. to "visiting" and "navigating between" sites (e.g., graph nodes). Perhaps such generic instructions could call C++ methods overridden for different kinds of input sources and their "regimes"; in particular, desiderata for moving from one site to another. Insofar as such techniques may apply to both raw data files and to sentence objects (and text/discourse elements

in general) from associated publications, the VM has the ability to visit and traverse both raw data structures and textual/linguistic content present in a Research Object. Both sorts of materials are tractable to the VM in a common search and front-end protocol. In short, the tools for working with raw data — via a VM implemented internally in an Executable Research Object — encompass manuscript content as well, bridging different data set components into a common computational space. Here I also circle back from data-related topics to textual encoding as discussed in the first section.

5 Conclusion

The first half of this paper focused on text encoding, whereas the second was involved with VM and compiler topics. For clarification, I should review the connection between these two subjects.

First. computer code in a Research Object context is (hopeful) annotated and documented, so it is subject to searching and indexing analogous to textual content. One consequence is that types' and components' "public interface" stands in an environment where synergy between code-related searching and querying over text or data sets is one design goal. The structure of procedures' input and output parameters, as well as how their preconditions get expressed and verified, influences how code-components may be presented and queried in consort with other Research Object facets. I believe that Overload by Contract and lexically-scoped proof objects facilitate such integration, insofar as implementing multiple versions of a procedure differing by contract-verification state — and visually marking the presence of precondition-satisfaction assumptions/stipulation within scopes where they apply — makes contract details explicit in program code, while also exporting their declarations to documentation tools. Employing a template framework similar to C++ — with value as well as type template parameters, thus supporting weak dependent types in a basic sense, but also explicitly connecting divergent template instantiations (even in the same specialization) to distinct bytecode blocks — similarly implements metaprogramming in a manner accessible to code/text/data integration.

Let me add that interface design is not only a matter of documentation; it affects actual component implementation as well. Scripting engines, routines populating context menus, "*en situ*" documentation (i.e., documentation not through external files but via tips or interactive information available as functionality within GUI displays), and so forth, depend of code-introspection features whose capabilities are shaped by sources' syntactic and semantic properties.

Similar points apply to those places in an interface where something other than return-by-value is intended. Merging pointer-return, reference-initialization, and Return Value Optimization into a common interface design can help, I believe, yield interfaces that cover variegated coding requirements in a unified and intuitive manner. Similar techniques also permit an expressive system for manipulating variable lifetimes through explicit notation in source code, which can clarify often-opaque programming details.

Focusing instead on text encoding, in Research Object contexts similar to ones envisaged here — where raw data files are paired with deserializer code such that data sets' contents get converted to runtime type-instances — then textual materials are subject to similar programming conventions. In particular, linguistic and discursive phenomena such as individual sentences, publication elements (quotations, footnotes, citations, and so forth), special character-sequences (chemical formulae, annotations for linguistics, terms governed by controlled vocabularies, etc.) and others are modeled by runtime objects in languages like C++.

That is, both textual phenomena and raw data records/observations are merged into a common runtime (and, canonically, Object-Oriented) information space. Under this assumption, it is reasonable to implement search and manipulation capabilities — plus potential script or query languages that access them — in a unified manner. Within an Executable Research Object, the topos of runtime values includes both raw-data and text objects.

6 Bibliography

[1] Bansal, Aruna, "HgMed: Hypergraphs Mediating Schematic Translations Between Data Models". *Proceedings of the 27th International Database Engineered Applications Symposium (IDEAS '23)*, May 2023. https://dl.acm.org/doi/pdf/10.1145/3589462.3589476

[2] Berne, Joshua, *et al.*, "Contracts for C++". ISO-C++ (2025). https://isocpp.org/files/papers/P2900R6.pdf

[3] Bleeker, Elli, *et al.*, "Between Flexibility and Universality: Combining TAGML and XML to Enhance the Modeling of Cultural Heritage Text". *Proceedings of CHR 2020: Workshop on Computational Humanities Research* (November 2020), pp. 340–350. https://ceur-ws.org/Vol-2723/short39.pdf

[4] Bleeker, Elli, *et al.*, "Texts as Hypergraphs: An Intuitive Representation of Interpretations of Text". *Journal of the Text Encoding Initiative*, Vol. 14 (April 2021–March 2023) https://journals.openedition.org/jtei/3919

[5] Botev, Chavdar, *et al.*, "Expressiveness and Performance of Full-Text Search Languages". *Proceedings of the 10th International Conference on Extending Database Technology* (March 2006), pp. 349–367. https://ecommons.cornell.edu/server/api/core/bitstreams/68074eab-f0fa-441c-81ea-0b4debf66f87/content

[6] Brady, Edwin C., "IDRIS — Systems Programming Meets Full Dependent Types". https://www.type-driven.org.uk/edwinb/papers/plpv11.pdf

[7] Braenne, Martin, *et al.*, "Lifetime annotations for C++" (Request for Comment), 2022. https://discourse.llvm.org/t/rfc-lifetime-annotations-for-c/61377

[8] Christen, Nathaniel and Neustein, Amy, *Innovative Data Integration and Conceptual Space Modeling for COVID, Cancer, and Cardiac Care*. Elsevier, 2022. https://shop.elsevier.com/books/innovative-data-integration-and-conceptual-space-modeling-for-covid-cancer-and-cardiac-care/neustein/978-0-323-85197-8

[9] Comeau, Donald C., *et al.*, "BioC: a minimalist approach to interoperability for biomedical text processing". *Database*, Vol. 2013. https://dash.harvard.edu/server/api/core/bitstreams/7312037d-13b7-6bd4-e053-0100007fdf3b/content

[10] Garabedian, Nick, *et al.*, "A Framework to Generate, Store, and Publish FAIR Data in Experimental Sciences". *Proceedings of the 19th International Conference on Semantic Systems* (September 2023). https://ceur-ws.org/Vol-3526/paper-01.pdf

[11] Grinbergs, Harijs, "Production Quality Decision Support Using Real-Time Computer Vision Framework". *Proceedings of Engineering for Rural Development* (May 2016), pp. 442–447. https://www.iitf.lbtu.lv/conference/proceedings2016/Papers/N081.pdf

[12] Harisanker, Pottayil, *et al.*, "A Practical, Typed Variant Object Model: Or, How to Stand On Your Head and Enjoy the View". *19th International Workshop on Foundations of Object-Oriented Languages* (Tucson, AZ) 2012. https://www.cs.swarthmore.edu/~zpalmer/publications/fool12.pdf

[13] Ludaescher, Bertram, *et al.*, "Capturing the 'Whole Tale' of Computational Research: Reproducibility in Computing Environments". *arXiv* 1610.09958 (2016). https://arxiv.org/abs/1610.09958

[14] Murphy, Sean, *et al.*, "Return value optimization (RVO)". *Special Interest Group on C++*, 2020. https://sigcpp.github.io/2020/06/08/return-value-optimization

[15] Peer, Limor, *et al.*, "Reproducible Research Publication Workflow: A Canonical Workflow Framework and FAIR Digital Object Approach to Quality Research Output". *Data Intelligence*, Vol. 4, Num. 2 (2022), pp. 306–319. https://www.sciengine.com/DI/doi/10.1162/dint_a_00133

[16] "Simplex Numerica User Manual", 2023. https://www.simplexnumerica.com/app/download/10205099171/Manual.pdf

[17] Stevens, Vicky, *et al.*, "Using ReproZip for Reproducibility and Library Services". *IASSIST Quarterly*, Vol. 42, Num. 1, pp. 1–14, https://iassistquarterly.com/index.php/iassist/article/view/18

[18] Strijkers, Rudolf, *et al.*, "Toward Executable Scientific Publications". *Proceedings of the International Conference on Computational Science (ICCS)*, 2011 https://www.academia.edu/109735244/Toward_Executable_Scientific_Publications?uc-sb-sw=37138629

[19] Uezato, Yuya, *et al.*, "Regular Expressions with Backreferences and Lookaheads Capture NLOG". *arXiv* 2404.17492 (2024). https://arxiv.org/pdf/2404.17492

[20] Wang, Lucy Lu, *et al.*, "CORD-19: The COVID-19 Open Research Dataset". https://openreview.net/pdf?id=0gLzHrE_t3z

[21] Zhilin, Anton, "Guaranteed copy elision for named return objects" (Published Proposal), 2020. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2025r2.html