

Infra: Structure All the Way Down

Structured Data as a Visual Programming Language

Christopher Hall
Computer Science
UCSB
USA
chall01@cs.ucsb.edu

Trevor Standley
Computer Science
Stanford
USA
tstand@cs.stanford.edu

Tobias Hollerer
Computer Science
UCSB
USA
holl@cs.ucsb.edu

Abstract

We present Infra, a new baseline medium for representing data. With Infra, arbitrarily-complex structured data can be encoded, viewed, edited, and processed, all while remaining in an efficient non-textual form. It is suitable for the full range of information modalities, from free-form input, to compact schema-conforming structures. With its own equivalent of a text editor and text-field widget, Infra is designed to target the domain currently dominated by flat character strings while simultaneously enabling the expression of sub-structure, inter-reference, dynamic dependencies, abstraction, computation, and context (metadata).

Existing metaformats fit neatly into two categories. They are either textual for human readability (such as XML and JSON) or binary for compact serialization (such as Thrift and Protocol Buffers). In contrast, Infra unifies those two paradigms. In order to have the desirable properties of binary formats, Infra has no textual representation. And yet, it is designed to be easily read and authored by end-users.

We show how the organization Infra brings to data makes a new non-textual programming paradigm viable. Programs that modify data can now be embedded into the data itself. Furthermore, these programs can often be authored by demonstration. We argue that Infra can be used to improve existing software projects and that bringing direct authoring and human readability to a binary data paradigm could have rippling ramifications on the computing landscape.

CCS Concepts • **Software and its engineering** → **Data types and structures**; **Visual languages**; *Programming by example*;

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Onward!'17, October 25–27, 2017, Vancouver, Canada

© 2017 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5530-8/17/10...\$15.00
<https://doi.org/10.1145/3133850.3133852>

Keywords human-readability, metaformat, structure editing, end-user development

ACM Reference Format:

Christopher Hall, Trevor Standley, and Tobias Hollerer. 2017. Infra: Structure All the Way Down: Structured Data as a Visual Programming Language. In *Proceedings of 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '17)*. ACM, New York, NY, USA, 18 pages. <https://doi.org/10.1145/3133850.3133852>

1 Introduction

Infra aims to make data more powerful and easier to deal with for both humans and computers. All types of data can be viewed, edited, processed, transferred, and stored entirely in Infra. Therefore, developers, runtimes, and end-users could theoretically share a common foundational medium across the computing landscape.

Infra is composed of a novel encoding and a novel type of editor/browser. These two components are intended to supplant the use cases of text encodings and text editors respectively, and since the encoding is compact binary, it also addresses the needs of transfer formats. Infra editors make reading and writing Infra's binary metaformat simple for end-users, and can even style the presentation and tailor editing in response to metadata, resembling a Web Browser or IDE. Beyond the common metaformat features, Infra's encoding includes three critical primitives: Metadata, Free, and Patch.

Metadata allows any data element, including other metadata, to be decorated with arbitrary Infra information to add context. For example, metadata is useful for providing IDs to support referencing values by name, style markup to assist presentation in an editor, or schema/type info to constrain or validate data.

Free allows encoded information to contain unallocated memory regions. This can be useful for aligning data to fixed-widths or improving the efficiency of localized edits to large structures on disk.

Patch elements are programs that can inline another Infra object and optionally modify the shallow copy, forming a generalization of graphs. This primitive turns out to be a powerful building block toward general computation in

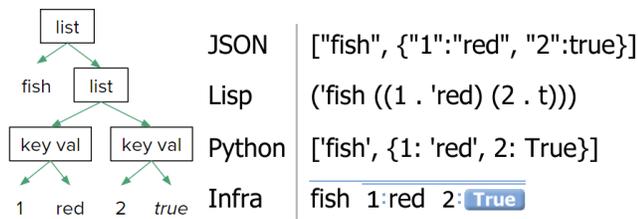


Figure 1. A toy data structure represented as a tree (Left). The data encoded in three textual formats, and Infra (Right).

the domain of data metaprogramming. In many situations, Patches and the Infra-encoded statements within them, can be conveniently authored indirectly via programming by demonstration.

This paper serves as an introduction of the ideas behind Infra. For a more comprehensive white-paper specification, source repositories, executables, video demonstrations, and tutorials, we invite you to visit Infra-Structure.org.

1.1 Motivation

Printable character codes are the sole building blocks of source code files, command line languages, form fields, Web formats (HTTP, URL, HTML, JSON, CSS) and all other “human-readable” formats. This is due more to the fact that early computers used electromechanical typewriters to interface with humans than because it is the only workable paradigm. Though all CPUs and runtime data-structures rely on binary-encoded quantities to structure information for random access, formats that need to be able to have a direct relationship with users are stuck with essentially one option - encoding their structure indirectly via contrived patterns of character codes. This is unfortunate because, text, as a UI paradigm, comes not only with compromises to efficiency, but functionality in general.

An entire class of problems with text arise from syntax. Editing structured data within text requires a fixed and often limited syntax riddled with reserved control characters. Not only does this limit the allowed schema of the data, but it requires that users be familiar with the syntactic elements used to control the structure of the data. Furthermore, syntactic elements, such as escaping, and the need for an unambiguous grammar, limit the readability of the data. With Infra, syntax is abstract, and structure is communicated graphically as each editor sees fit for the particular context.

2 Infra: Human and Machine Friendly

Figure 1 is a side-by-side comparison of some structured data expressed equivalently in four different languages. This is a toy example engineered to show off a range of Infra’s element types.

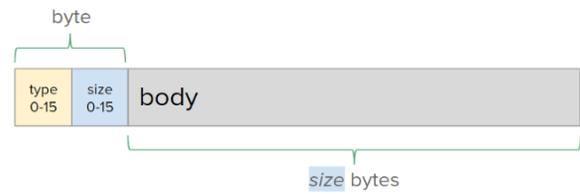


Figure 2. The byte structure of an encoded Infra segment.

Infra provides structured scaffolding for holding data, but it does not attempt to invent a new character encoding, so ‘fish’ and ‘red’ are encoded as UTF8 strings. On the other hand, ‘True’ is directly encoded as a boolean value, and is shown in blue. Typical formats communicate structure using characters such as ‘[’ and ‘(’. Infra communicates the abstract syntax present in the data using graphical elements. For example, the span of lists is indicated by a blue line above the items it contains.

We will continue working with this example throughout this section.

2.1 The Encoding

Infra’s metaformat encoding consists of a sequence of ‘segments’, each made of a header byte and a body of variable length. The header byte indicates the type of the segment and the length of its body. This pattern is sometimes called type-length-value, tag-length-value, or key-length-value. Due in part to its simplicity and processing efficiency, type-length-value is a common paradigm, used by many binary file formats such as Portable Network Graphics (PNG) [4], Audio Video Interleave (AVI), Matlab’s MAT, Protocol Buffers [8], MessagePack [12], and most modes of the complex ASN.1 [16] format. The finer details vary among these formats, but the most significant decision that differentiates the utility of these formats is the set of first-order primitives, or base types, that they define.

We define 13 base types that support direct authoring and are sufficient to enable a general set of applications. Only half a byte is needed to account for 13 base types (plus 3 unallocated). Segment headers can be a single byte when body lengths are no longer than 14 bytes. When the body length is greater than 14 bytes, additional bytes must be used to indicate the length. The value of 15 is used to signal this, and the header byte is followed by a variable length unsigned integer encoding. We find Dlugosz’ encoding [17] to be efficient and well-designed for this purpose. This is the encoding used for VLIs in the ZIP2 format.

2.2 Direct Authoring by End Users

With Infra, our first priority is supporting interactive authoring by end users. An Infra editor aims to fill the same role in computing that text editors and text-field widgets

Type	Name	Category	Encodes
0	Free	leaf	Unallocated gaps in serialization
1	Byte Array	leaf	Block of arbitrary bytes
2	UTF-8 String	leaf	A token of text
3	Integer	leaf	<i>n</i> -byte signed integer
4	Floating Point	leaf	<i>n</i> -byte IEEE 754
5	List	container	An internal tree node
6	Keyed List	container	First list item is treated as a key
7	Patch	container	Instructions for inlining a [modified] reference
8	Metadata	container (associative)	Set of markup associated with the following item
9	Continuation	container (associative)	More items for a previous list (segmented streams)
A		unallocated	
B		unallocated	
C		unallocated	
D	Bit Array	leaf (special)	Block of arbitrary <i>bits</i>
E	Nibble	leaf (special)	Efficient encoding for integers 0 through 15
F	Symbol	leaf (special)	Enum: <i>False, True, Void, Null, Parameter, Problem</i>

Figure 3. Infra’s 13 segment types.

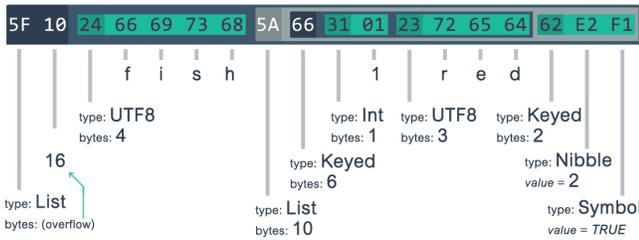


Figure 4. Infra’s byte encoding of the data structure shown in hexadecimal.

currently play. Like a text editor, which tries to be as adept and general-purpose as possible when it comes to enabling users to view and manipulate a buffer of character codes, an Infra editor tries to make authoring structured data easy. This includes having mechanisms to facilitate the authoring of spans, quantities, references, metadata, and padding.

Using an Infra editor feels like using a text editor. Unlike text editors, however, Infra editors have the opportunity to add useful structure as users type. Users can type their intended structure along with their content. For example, pressing ‘spacebar’ between words defaults to tokenizing the text into lists of strings. Furthermore, recognizable fields such as numbers can be parsed on-the-spot and converted into the appropriate Infra element type (such as Floating-point, which is binary-encoded)¹.

¹ To get a feel for what editing Infra is like, please take a moment to watch two short videos at the following links. Transcripts of the video demos are provided in appendices A and B.

Video 1 <https://youtu.be/L8VpCClxuME> (~3 min)

Video 2 <https://youtu.be/8k6n1m4leQo> (~2 min)

3 Lists and Keyed Lists

A list is a container to group zero or more data structures together. In our prototype editor, lists are represented simply as a line spanning over the items it contains. Lists can contain elements of any type, including other lists. In fact, trees can be built using lists of lists. Unlike text editors, which edit flat character arrays, Infra editors are designed to work well with hierarchical data.

The quick brown fox jumped over the lazy dog

In the above example, ‘quick’, ‘brown’, and ‘fox’ are grouped together in a List. ‘lazy’ and ‘dog’ are within another List. ‘The’, ‘jumped’, ‘over’, and ‘the’ are at the root level.

The selection cursor can also be hierarchical in order to edit at any level of granularity present in the data’s structure. In addition to moving the cursor between siblings, a central user-interface action is to move the selection down to a child or up to a parent container.

The quick brown fox jumped over the lazy dog

down

The quick brown fox jumped over the lazy dog

down again

The quick brown fox jumped over the lazy dog

In the above figure, the top row depicts selection of the second element as a whole. Moving the cursor *down* or *in* results in the second row, where ‘quick’ is selected, and moving the cursor to the right would now select siblings of ‘quick’ as opposed to siblings of the List (i.e. ‘jumped’). As seen in the third row, the cursor can be moved *in* again to operate at the character level in the familiar way.

Any tree can be displayed as a column-aligned table.

fox quick brown 5 female dog lazy 7 male

fox	quick brown	5	female
dog	lazy	7	male

Keyed List is a variant of **List** that associates the first child with the container itself. This is similar to the concept of a key-value pair where the first child is taken as the key, except Keyed Lists can have any number of values. Keyed nodes are also similar to Lisp’s S-Expressions and are used to encode Patch instructions, which we describe further in the chapter on Patch.

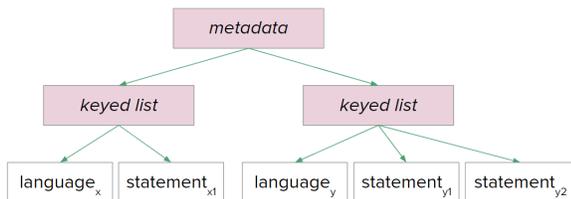
Our prototype editor displays Keyed Lists in either of two visual styles: a parentheses-like style or a colon-like style. Since it is only a presentation layer decision, the user has both on hand. The following figure compares the visual differences between Lists and Keyed Lists of zero through three items respectively.

List:	□	A	A B	A B C
Keyed:	()	A()	A(B)	A(B C)
	:	A:	A:B	A:B,C

We find that the colon-like style is more appealing for when there are exactly two items in the Keyed List (including the key), i.e. A:B. But we find that the parentheses-like style is generally less visually ambiguous for cases of fewer or greater than two items total. (To be clear, such ambiguities would be strictly user-interface-level issues, not encoding-level ambiguity.) In our figures, you will find that we mix the use of the two styles for best readability on a case-by-case basis.

4 Metadata

In *Infra*, metadata can be associated with any element, including other metadata. Metadata can be laid out in various ways or selectively hidden. In our prototype editor, metadata is shown in a smaller font over the element it is associated with. The first tier of elements within a metadata container are Keyed Lists so that all metadata ‘statements’ are associated with some language identifier. This not only provides a means to anchor interpretation of the metadata to some recognizable semantics, but also to allow any number of metadata layers to coexist on the same data node. If no metadata exists on a node, then the entire container is simply absent from the encoding. The capacity to have metadata costs nothing if it is not used.



There are two exceptions to the rule of metadata items being Keyed Lists:

- a UTF8 element found as a direct child of a Metadata node is taken to be ‘ID’ metadata, i.e. shorthand for the Keyed List **ID:string**.
- an Integer element found as a direct child of a Metadata node is taken to be ‘UID’ metadata, i.e. shorthand for the Keyed List **UID:number**.

In the following example, metadata has been authored onto the strings ‘fox’ and ‘dog’. The metadata values are keyed as ‘adj’ markup using Keyed Lists.

adj : quick, brown adj : lazy

The fox jumped over the dog.

This is equivalent to viewing the following HTML in a text editor (with HTML-specific syntax highlighting):

```
The <span adj="quick" adj="brown"
>fox</span> jumped over the <span
adj="lazy">dog</span>.
```

However, this HTML is malformed because repeated attribute tags are not supported. In practice, “quick” and “brown” would have to be combined into one value using a one-off syntax scheme to indirectly retain their boundaries. At that point, the HTML parser, syntax highlighting, and editor assistance stop helping, and custom parsing must be added wherever the values are used. Note that HTML attributes cannot themselves also have attributes (no recursive metadata).

4.1 Data-Driven Presentation

One of the many uses for metadata is to hint to *Infra* editors/browsers what abstractions are appropriate when displaying a particular piece of data.



On the left side of the figure above is a byte array of size three displayed in hexadecimal by the editor. On the right side is the same element after the user added ‘format’ metadata. As it happens, this editor recognizes ‘format’ markup, and the value ‘RGB’ gives the editor confidence to instead display the byte array as a color swatch, which can even be interacted with as a color picker, making editing the value much more intuitive.

Our prototype editor also supports a subset of the CSS standard, which makes use of color values, so let us combine this example with the previous one. In this scenario, the same

The **fox** jumped over the dog.

```
The <span style="font-style:italic; background-color: #ff9212">fox </span> jumped over the dog.
```

Figure 5. Top: editable Infra. Bottom: Editable HTML.

metadata exists on the color value in the “background color” property, which happens to itself be metadata. (The ‘format’ metadata is not shown here because it is at least two levels removed from the current position of the selection cursor. Moving the cursor to the first metadata level will expand it.)

CSS { font style : italic
background color : **fox** }

adj (quick brown) adj (lazy)

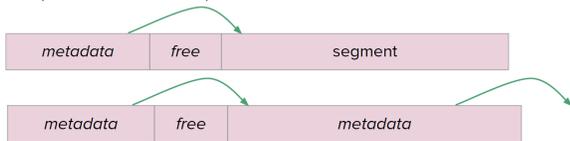
The **fox** jumped over the dog.

There are several noteworthy aspects to this structure. Several grammatical constraints are relaxed relative to typical CSS due to Infra circumventing the bottlenecks of a tokenizer. The style property names can have spaces in them, rather than being forced to use hyphens to separate words. The byte encoding of the color value is in binary, which is more compact than “#ff9212” by a factor of three and moves the parsing to author time rather than render time. As we will explore later, Infra encodings can also use its Patch base type and metadata layers to bring string de-duplication and value computations to CSS or any other application.

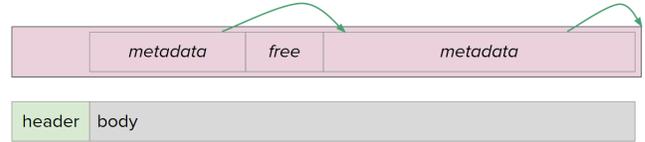
If an editor displays metadata layers in a separate panel on the side, or the user toggles off the display of metadata entirely, the rendering of Infra with style markup will resemble rendered HTML yet remain editable. For a visual comparison, see Figure 5.

4.2 Metadata Association Rules in the Encoding

Metadata segments associate with the segment immediately following it in the stream, skipping segments of type Free. Metadata can be associated with other metadata with no issue (meta-metadata).



If the last segment in a span is metadata, it associates with its container. If it is in the ‘top level’, it is treated as metadata for the tree itself.



4.3 Schemas

We find it useful to define an optional system for ensuring that data conforms to a specified type definition. The metadata channels ‘schema’ and ‘child-schema’ and are defined for this purpose. Metadata in the ‘schema’ channel is treated as an exemplar constraining the allowed shape of the data. Likewise, when ‘child-schema’ metadata occurs on a list, the list’s children are constrained to take the shape of the exemplar. These constraints, as well as the defaults defined by the exemplar, help editors to provide context-aware editing functionality, such as auto-complete.

5 Case Study 1: URL Syntax

Let us explore a hypothetical alternative reality where computing’s text infrastructure never consisted of only characters codes, and was built up around a parametrized syntax such as Infra. The UI widgets used for everyday tokens of input/output (such as Text Fields) would be Infra editor widgets and literacy around using keyboards would think in terms of authoring structure along with values. In this section, we explore the effect this would have on the nature of computing by focusing on an everyday unit of structured information - a URL.

The following URL is a link provided by a Google search result. It is the link to the Wikipedia article on Uniform Resource Locator, but the actual URL is a redirect through Google’s servers for accounting purposes. This kind of URL is chosen because it is representative of complex stateful URLs as well as the fact that it contains a URL inside itself.

```
https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=6&cad=rja&uact=8&ved=0CDkQFjAF&url=http%3A%2F%2Fen.wikipedia.org%2Fwiki%2FUniform_resource_locator&ei=tE8sVe-iF4m8ggSZi4T4AQ&usg=AFQjCNFVKoOa_HlCuDeUu8wYS_g70me4Kw&bv m=bv.90491159,d.eXY
```

Note that this URL is not very readable, and that the embedded URL is escaped and does not work if copied to a browser address bar as is. The bulk of the characters in URLs like this are Base64-encoded bit strings. Base64 encoding is born out of the fact that human-readable formats are unfriendly to binary data, and in the web world, there is even further need for compromises to encoding in URLs, to avoid having to escape plus and forward slash characters.

Now let us jump to looking at how URLs could have formed differently if Infra boxes existed before text boxes did. Eight notable improvements are listed after the image.

```
https www google com url
sa:t rct:j q: escrc:s source:web cd:6 cad:rja uact:8
ved: [Data Matrix] ei: [Data Matrix] usg: [Data Matrix] bvm {bv:90,491,159 d:eXY}
format: URL
url: http en wikipedia org wiki Uniform_resource_locator
```

- The elements of a domain name do not have to be separated by punctuation. It can simply be a list.
- This applies the same way to the path component of a URL.
- The query fields are key-value pairs and can be grouped together.
- Numbers stay binary encoded. (As they were in the memory of the computer that constructed the URL.)
- Bit strings can stay bit strings without the need to use indirect representations such as Base64. These bit strings are displayed as a Data Matrix (one of many possible visualizations at the disposal of the editor UI such as Chroma Hash). The use of a data matrix allows for a compact display of a binary value that does not necessarily need to be readily deciphered by a human, while giving some ability to judge equality. In this case of reverse engineering Google's URL, it is not obvious if the values 't', 'j', 's', and 'rja' are also meant to be treated as Base64. With Infra, such an ambiguity would not have to exist.
- The nested URL does not have to be escaped, in fact, it is also parsed, and even labeled as being a URL with 'format' metadata.
- Underscores do not have to be used as a substitute for spaces.
- The 'bvm' value can have the substructure it seems to want. In this case it was parsed into two values separated by comma, and then sub-split into key-vals by period. The '90491159' portion is numeric and is encoded more usefully as a binary integer - able to be displayed according to the user's preference for localized digit-grouping.

Since the query fields are grouped together, they can conveniently be displayed in a tabular arrangement at the request of the user. From this layout, the information structure is quite clear, and it is easy to notice that 'bvm' is the only field to have more than one associated value.

Not only is unescaping an escaped URL by hand tedious and cryptic, but it also requires using an ASCII table for reference. However, in the hypothetical case of Infra-based URL syntax, extracting the forwarding address is trivial, as

is any other kind of quick editing. In daily routine, we find ourselves often needing to manually tweak video links (such as YouTube) to either remove the playlist portion (so it only links to the specific video) or to nudge the timecode (because we hit pause a little late before copying the generated link). In the pure-text world, doing these simple kinds of things requires familiarity with URL's specific meta-characters, rather than just being the same kind of structured editing across all user interfaces.

6 Patch

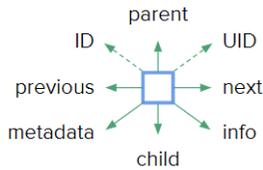
A Patch is a program that returns a data structure. The simplest type of Patch simply references another node in the tree, returning a shallow copy; this expands the domain of infra from trees to fully-general graph structures by providing crosslinks. Patch programs can also make modifications to what they return (without modifying the original); this enables pure-functional programming at the encoding layer. In the encoding, Patch nodes are containers like List nodes, except their children are interpreted as instructions for assembling a return value.

The Patch execution model is centered around the metaphor of a virtual cursor, equivalent in nature to the cursor in an Infra editor. There are instructions to move the cursor around and instructions to modify the return value at the cursor's location. Each Patch instruction is a **Keyed List** made up of an opcode (key) and a set of arguments (values). The set of available opcodes is the same as the set of edit operations available in the Infra editor. Not only is this set of opcodes general enough to make any modification to a return value, but it allows for a user-friendly way to author simple Patch programs. A modified reference can be authored by demonstration without ever needing to see or write Patch code. A user's modifications to the Patch output are appended to the Patch's instructions like a macro recording².

A Patch's virtual cursor begins execution at the Patch's own location in the data tree. Starting at that point, Patch instructions navigate to the desired node, and then describe the modifications that should be made to the returned version of the referenced node. A Patch's edits operate on a private overlay of the structure so that edits are reflected only in its return value and not in the original source material, such as in immutable or persistent data structures. Later, we will see that Patches can return Side-Effect objects which can, in turn, perform controlled side effects.

² Please take a moment to watch a video demonstration of the Patch mechanism in action at the following link. A transcript is provided in appendix C. **Basic Patch demo** <https://youtu.be/hs42TeFytEk> (~2 min)

6.1 Opcodes for Navigation



parent(*n*) shifts the focus cursor up the tree by *n* tiers. If the argument is omitted, the behavior is equivalent to `parent(1)`. If there is no *n*th parent, the Patch instead evaluates to the Problem symbol with metadata describing the issue and execution is halted.

child(*i*) shifts the focus cursor to its *i*th child. Or if the argument *i* is a Keyed List, the focus will shift to the first child Keyed-List with a matching key (the same key as in the argument). If the argument is omitted, the behavior is equivalent to `child(0)`. If multiple arguments are provided, each will be considered an index for successive applications of `child(i)`. In other words, `child(2 0 1)` would be equivalent to the sequence: `child(2) child(0) child(1)`. A negative index value can be used to index backwards from the end of the list. Thus, the last item of a list can be focused with `child(-1)`, and the second to last with `child(-2)`, etc.

previous(*n*) shift focus to the sibling with the index *n* less than the focus' own index in its parent.

next(*n*) shift focus to the sibling with the index *n* more than the focus' own index in its parent.

metadata(*channel*) shifts the focus to its associated metadata container, and then to a Keyed List within it whose key matches *channel*. If the argument is omitted, focus just moves to the metadata container in general.

ID(*id*) jumps the focus cursor to the 'nearest node' with an ID-metadata value matching *id*. The search order resembles classical scoping rules for identifiers in most programming languages. To start, the first level of children of the focus are searched. If none have ID metadata that matches, siblings are searched. And then, the search resorts to siblings of the parent, grandparent, etc.

UID(*uid*) jumps the focus cursor to the unique node with a UID-metadata value matching *uid*. UID-labeled data have their own namespace and do not have to avoid name collisions with ID-labeled data.

info() shift focus to a synthetic tree populated with information about the element that was currently in focus, such as its number of children, its index position in its parent container, and its encoding type.

6.2 Opcodes for Modification

The secondary role of Patch instructions is to perform edits, modifying the value being referenced, but only from that Patch's perspective. This is akin to concepts such as: copy-on-write, persistent data structures, and 'modifiable references' in [1]. Since the original reference material is guaranteed not to be modified, and that material is the Patch's only input source, Patches behave like 'pure functions'.

insert(*v*) modifies the focus' parent to contain *v* at the same index as the focus cursor. If multiple arguments are provided, all will be inserted at successive index positions.

remove(*n*) removes the next *n* items starting at the position of the focus cursor. In the argument is omitted, the behavior is equivalent to `remove(1)`.

write(*v*) overwrites the value at the current focus with the value *v*. This acts like a `remove()` followed by an `insert(v)`.

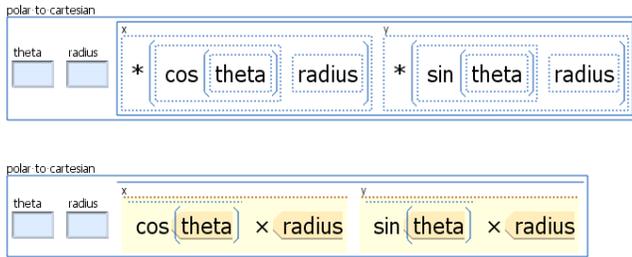
sync(*label*) halts execution and causes the Patch to evaluate to a Side-Effect object. (See the following section: [Effect System](#)) In our prototype editor, we represent a Side-Effect object as a clickable button labeled with the value of the *label* argument. When the user clicks the button, the Patch is resumed in a context where it is safe to mutate the subtree it references.

6.3 Patch as Function Application

Infra has no need for a native concept of a function or a function call. Since Patches can be defined inline, Patch semantics act simultaneously in the role of a function *and* a call site. Conceptually, function application is the process of taking an instance of a function and substituting argument values in for parameter placeholders. That very process can be performed *by* a Patch using the building blocks we have already introduced.

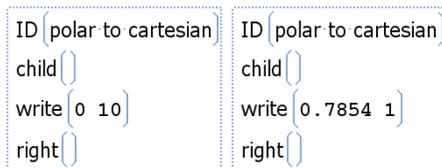
The convention for mimicking a traditional function is to have a list of default parameter values followed by a result value that is composed of one or more Patches that reference those parameter values. Instead of providing a default value for any particular parameter, Infra's Parameter symbol can be used as a valueless placeholder. ID or schema metadata can optionally provide names and type constraints and on any or all parameters.

Let us look at a concrete example function. To do this, we are going to have to get slightly ahead of ourselves and use an arithmetic object for multiplication, which is not introduced until the section on Native-Service Objects. The following example is a function that converts an angle and radius to an x-y pair.



The top row shows the Patch tree without any evaluation, and the bottom row is the reduced (semi-evaluated) form that would appear in the editor (see the discussion on Native-Service Objects for clarification). The pale blue boxes labeled 'theta' and 'radius' are *Parameter* symbol elements, which are meant to be used as placeholders for a future value - perfect to play to role of an input parameter.

Next let us look at two different elements that each call this 'polar to cartesian' function with their own arguments.



And here is what these Patches evaluate to:



There is an air of machine-code level programming to this use of Patch, but note that users will be viewing the evaluated output of Patches most of the time and author them indirectly through direct manipulation of their result value. Taking advantage of live interaction and iterative authoring of input values can enable a Patch to dispense its own documentation dynamically in response to the partial values as they are being assembled.

A general benefit of leaving function application as an emergent ability of Patch semantics, is that it itself is programmable. A variety of function-application semantics can be supported while keeping the number of first-class concepts in the Infra specification minimal.

- **default arguments** are achieved simply by not overwriting some of the hard-coded values in a function interface.
- **call by value** is achieved when writing literal values as input arguments.
- **call by name / need** is achieved when writing Patched values as input arguments. (lazily evaluated arguments)
- **currying** is achieved when writing values to only a subset of locations and leaving the rest to be referred to and written by other Patches.
- **named arguments** are achieved when using ID metadata on arguments to locate them.

6.4 Effect System

As described above, a Patch's edits normally only effect the result value of the Patch, leaving the referenced source material unchanged. On the occasions that it is useful for a Patch execution to have a stateful effect on the world, an effect system helps this to happen while keeping side effects explicit and controllable. Infra has an effect system made up of three components: the Side-Effect object-type that can be returned as the result of a Patch, the sync() opcode that exports a Patch's attempted mutations as a Side-Effect object, and a permissions system to regulate automatic execution of the side-effects described by a Side-Effect object.

A Side-Effect object is analogous to a Pull Request in the popular Git version control system. They are inert return values until they are applied/triggered. They encapsulate the edits that a Patch has made to the data it was referencing, such that those edits can be applied to the original (a destructive change) at the discretion of the runtime system. Because Side-Effect objects are represented as an interactive button in our prototype, they also resemble and behave like toolbar or drop-down-menu buttons in a graphical user interface, which trigger specific useful state changes on demand. An Infra editor UI allows users to trigger Side-Effect objects directly. Any Patches that have a dependency on a value being mutated are invalidated, and will be re-evaluated as needed. This same mechanism already has to be in place for normal edits made directly by the user.

When a Patch contains within it a Patch that returns a Side-Effect object, the top-level Patch will also evaluate to a Side-Effect as well. Logically, this is because it is not only roadblocked waiting for *its* side effects to occur before proceeding, but it also needs to have a context for synchronizing mutations for the nested one to inherit. If schema metadata appears on a Patch that evaluates to a Side-Effect, the schema value itself needs to be a Side-Effect object in order to be consistent.

6.5 Native Service Objects

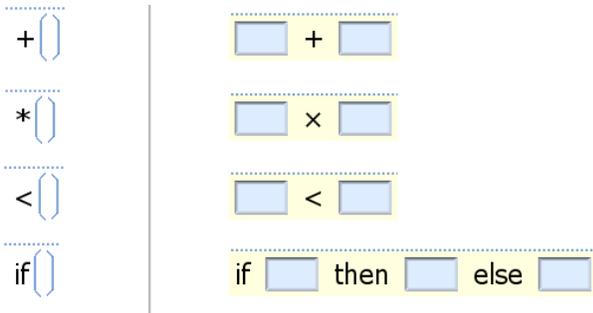
As we have seen thus far, Patch opcodes just perform tree navigation, insertions, and deletions. On their own, those operations can not perform computation that is sensitive to the values they operate on, which is to say, they are not Turing complete. However, those operations *are* sufficient for performing 'function application'. The right primitive functions just need to be available in order to bootstrap a capacity for computation. This is where the native service objects in the standard library come in.

Since their logic cannot be expressed in terms of virtual cursor manipulations, these built in functions must be built in to the standard just like the primitive operations in other programming languages are. They are loaded by name, just like any other named entity, but they each override Patch evaluation or mutation semantics with their own native logic

rather than execute their contents as standard Patch cursor instructions. This allows them to use their contents merely as a presentation layer for their parameters - free to act like a domain-specific language.

We have explored Native Service Objects for performing logic and arithmetic, for performing operating system input/output (with the help of Infra's effect system), and for inspecting and manipulating Java runtime objects and methods.

Logic and Arithmetic The logic and arithmetic entries in the standard library include boolean operators such as conjunction and disjunction, mathematical operators such as addition and subtraction, and control flow structures such as if-then-else.



In the table above, the first column contains four example Patches. The second column displays their corresponding evaluations, all of which are native-service objects. In our prototype editor, subclassed types are given a yellow background tint to remind the user that they evaluate according to overridden semantics. These native-service values cannot be serialized directly in the Infra encoding. This means that they are always a result value of a Patch that references their *a-priori* existence in a way that can be serialized.

Note that the text elements in these objects are purely decorative for the sake of their user interface. They are not necessary for the object to perform its function, but would be nondescript without them. In the case of the math operators, the interface is able to resemble a familiar infix notation without the need for any explicit support or syntax for distinctions between prefix, infix, and postfix operators. Also note that, in the case of the multiplication example, the decorative elements can help expressions to use more appropriate Unicode characters without requiring the user to deal with the round-about ways to type them manually.

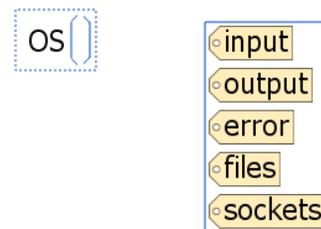
As usual, the parameter values can be written in with Path's modification opcodes, or the shorthand notation can be used if it is sufficient to fill parameters in depth-first order. The following table depicts the shorthand notation of listing argument values in the body of the instruction.

+ (394 167)	394 + 167	561
if (True)	if True then [] else []	[]
if (True Hi)	if True then Hi else []	Hi
if (True Hi Bye)	if True then Hi else Bye	Hi
if ([] Hi Bye)	if [] then Hi else Bye	!

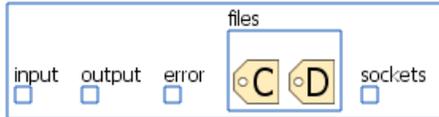
The first column is the directly encodable form of a call to a native object with argument values. The second column shows the values after one evaluation. The third column shows the values after a second evaluation.

In the prototype editor, Patches are displayed only as evaluated as they can be without error. In other words, the editor automatically evaluates Patches up to a point. This refers to Patches that evaluate to a Patch, that in turn evaluate potentially to a Patch. Once an evaluation chain results in a Problem symbol, the previous stage is the one displayed. Therefore, the example in the fifth row would be displayed in the form of the second column, while the others would be displayed in the form of the third column. This assists the user in filling in missing values or addressing errors. The lazy evaluation of Patches means that even deeply-nested issues could be easily addressed on the surface, one at a time, without clutter.

Operating System Integration Native-service objects can provide external forms of input and output. Operating System integration is addressed by four categories of native-service object: standard input/output console streams, a file system tree, executable process interface, and socket binding interface. All instances of these services all grouped under a single object registered as "OS".



On the left is a Patch that jumps to the OS tree and stops. On the right is its evaluated value. These five items look the way they do because they have ID metadata values and our prototype defaults to displaying named elements as their ID value. To help avoid confusing the ID as the actual value at that location, it is rendered to look like a luggage tag. This is an example of an editor providing alternative view modes for subtrees. With the default Face we can see the actual values, and the tree would look like:



Each of these items are Lists that keep their contents in sync with the external reality of their corresponding data model in the operating system. Each are a special Native-Service Objects instantiated as singletons inside the OS Service Object. If an element is inserted into ‘output’, the element is also written to the process’ standard output stream. Bytes written to the process’ standard input stream get interpreted as Infra elements and appear in the ‘input’ list. If input bytes do not successfully parse as Infra, they appear as plaintext or byte arrays, depending on a human-readability heuristic used.

Now we have the building blocks we need to write a true “Hello, World!” program in Infra. The following Patch results in a button that, when clicked, prints the string to the standard output stream. (Reminder: printing to standard out is a state mutation and therefore requires the use of the ‘sync’ opcode.)

```
comment: instantiate a view of the operating system tree
OS ( )

comment: navigate to the standard output node within the OS tree
ID (output)

comment: add data to the end of the standard output node (nothing will print yet)
append (Hello, World!)

comment: return Side Effect object to commit changes to the original output node
sync ( )
```

In the figure before last, ‘C’ and ‘D’ represent drive letters (the root file system objects on our machine). These are actually List elements with ID metadata and, just as before, they are being displayed in a mode where their ID represents their whole. In the UI, the actual children of a directory will be rendered when selecting or drilling down into that element. To avoid implicitly rendering the whole file system tree at once, it is important that Infra editors delay the loading of sub-trees until they are expanded.

We leave discussion of the sockets sub-tree for future work.

Runtime Language Reflection Since Infra is an infrastructure based around directly authoring and editing structured data, there is a natural mapping between the internal data structures of a programming language runtime and

human-readable Infra data structures. For programming languages that feature runtime reflection, these mappings can be automatically supported without having to prepare an adapter for each data type in advance. Reflection makes it possible for the library to dynamically assemble representations for any object without the need to run pre-processors on source code or be involved at compile-time. Runtime objects can be visualized on demand, by the Infra medium. The Infra medium also naturally brings with it the interactions necessary to manually assemble argument values into and invocation of native functions/methods. This essentially allows Infra to act as a visual debugger for the runtime environment that the editor implementation is running in.

We leave the details of these Native-Service Objects for future work.

7 Case Study 2: Plain Text at Scale

This section briefly explores the cost of storing abstract structure within content in the way that Infra proposes all data be authored and stored. There are varying degrees of structural breakdown, hierarchy, and interconnection possible with any kind of data. For starters, we will look at just a basic first pass of sub-structure that can be given to most plain-text content - tokenization.

We have tokenized a sample of English texts and source code files within Infra to measure an average byte overhead introduced by Infra’s element headers, which segment each word. Infra editors display whitespace padding between elements, so actual space characters are not needed between words. Elements of fewer than 15 bytes only require a 1-byte header, and so most of the time, the presence of the header byte is made up for with the lack of need for a space character. However, newline characters are a common occurrence in textual data and are not often preceded by whitespace, but on some operating systems, they are accompanied by a carriage-return character. In all cases tried, the byte overhead was less than 4%.

For the full text of Lewis Carroll’s “Alice’s Adventures in Wonderland”, the byte size increased from 163,815 bytes to 169,096 bytes when tokenized simply by splitting on space characters. This is an Infra overhead of 3.2% to have structure at the word level. But, now that there is word-level structure, Patch can be used to de-duplicate strings by encoding a common word once and referring to them from the locations where they are used. As long as the byte size of the Patches themselves is smaller than the word they reference (minus the one-time cost of metadata to number the word), memory will be saved. In the case of Alice in Wonderland, the storage size can be reduced by 44,206 bytes (26.1%) through basic string de-duplication.

For an example of what this kind of Patch usage looks like, let us take the famous quote from JFK:

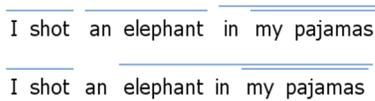
ask not what your country can do for you - ask what you can do for your country

ask not what your country can do for you - 0 1 7 4 5 6 2 3

The second row shows ‘UID’ metadata and Patches unevaluated. (Reminder: Metadata and Patch both have shorthand for ID and UID when using strings and numbers respectively, which is why the metadata does not appear as “UID:4” and why the Patch commands do not appear as “UID(4)”.)

String de-duplication is a simple form of data compression, but importantly, this is not a compression scheme that obfuscates the data format. Patches are referentially transparent, and so substituting an element for a reference to the same value is a non-disruptive transformation.

Hierarchies can be added to make explicit the sentence structure or ‘parse tree’ of the text. For example, here are two possible parse trees for the same ambiguous sentence:



We added Stanford’s open-source NLP parser to the editor prototype for automatically adding best-guess grammatical structure. Ignoring metadata to additionally label parts-of-speech, embedding sentence structure into the text of Alice in Wonderland required an overhead of just over 10%. Note that the deduplication saved more than enough bytes to make up for the tokenization and parse trees, totalling a savings of 15.8%.

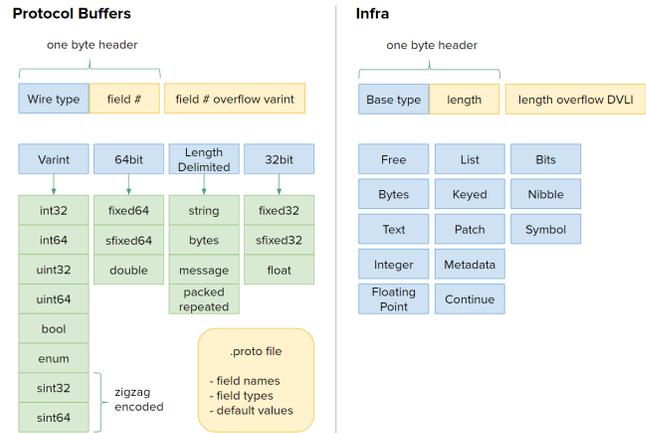
Being able to include extra structure such as a natural-language parse has a multitude of uses for downstream processing. In this case advanced users have the opportunity to correct bad parses.

Degree of structure applied to the text of Alice In Wonderland	Bytes
Traditional monolithic string	163,815
Basic tokenization (on whitespace)	169,096 ↑3.2%
Basic word deduplication	124,890 ↓26.1%
With embedded NL parse trees	137,953 ↑10.5%

8 Case Study 3: Protocol Buffers Replacement

As far as compact high-efficiency serialization formats go, Google’s Protocol Buffers [8] are, by our estimation, the most widely known, used, and supported in a modern setting. In this section, we will refer to it simply as ‘Proto’. Overall, Infra has roughly the same byte efficiency as Proto. Both Infra and Proto precede elements with a one-byte header split into a type enumeration portion, and a scalar quantity portion. Also in both cases, the header is conditionally followed by

a variable-length unsigned-integer encoding to allow the scalar quantity to overflow into more bits.

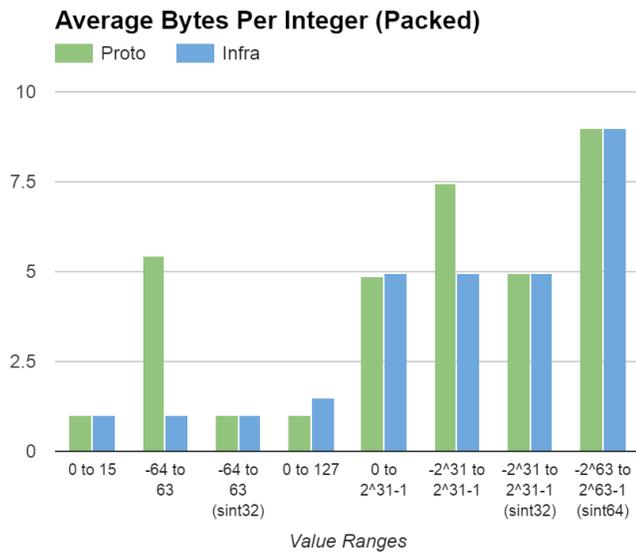
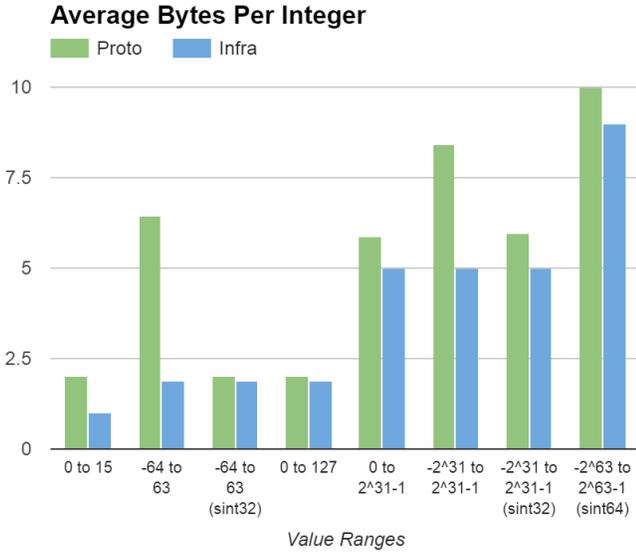


The performance of Infra and Proto are tricky to compare directly because they are designed for nearly opposite circumstances. Proto was designed to be manipulated procedurally by pre-compiled code, and to eliminate as much unnecessary exposition of the data on the wire as possible. Infra was designed to be viewed and authored directly in its encoded form, and to allow for as much exposition of the data on the wire as the user wishes to include. That being said, Infra can still be used in a constrained way as not to embed any more than the bare minimum necessary for Proto-like use cases.

8.1 Integer Encoding

In this comparison, we focus on the efficiency of encoding integer values, since that is where bit widths are the most dynamic, and where the bulk of the design complexity resides in Proto. Infra has two integer base types (Integer and Nibble), while Proto has ten (int32, int64, uint32, uint64, sint32, sint64, fixed64, sfixed64, fixed32, sfixed32). Infra can get away with effectively one integral base type because Infra’s headers are parametrized by byte length, whereas Proto’s headers are parametrized by field number.

For the following measurements, various trials of encoding a list of ten thousand integers in each encoding were performed. The integers were randomly chosen from a flat distribution. Trials vary in the range of random integer values chosen (small, large, negative) in order to exercise various phase changes in the encodings. The list is serialized using each encoding, and the total byte length of the serialization is divided by the number of elements (ten thousand) to arrive at an average number of bytes per integer. This averaging amortises away the one-time-cost portions of their byte overhead.



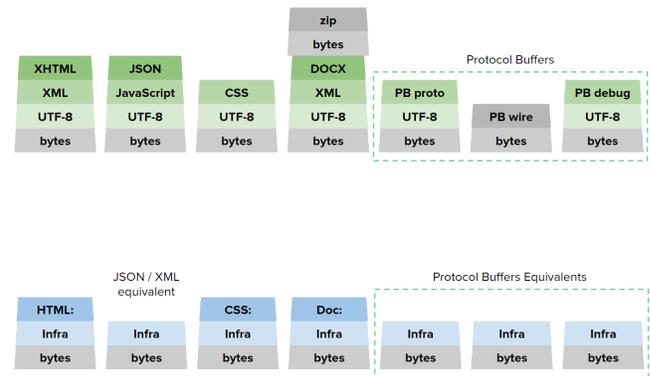
Value Ranges: Proto has unsigned integer types, signed integer type, and *sign-unspecified* integer types. Proto’s variable-length integer encoding ends up being highly inefficient for negative values (two’s complement) if a *signed* type is not specified explicitly. It falls on the user defining the .proto file to make a judgement call regarding the frequency of negative values that will appear in future data. This is the issue responsible for the measured spikes of inefficiency for Proto in the second and sixth value ranges. Those two ranges are run again with ‘sint32’ and ‘sint64’ specified in the .proto file respectively.

‘Packed’ Mode (Second Chart): Proto has a ‘packed’ option for repeated fields, in which it forgoes repeating the same tag header for every element in a list. ‘Packed’ can be specified in the definition of each repeated field of a scalar

type or by declaring “proto3” syntax mode. Infra can do something analogous using the Bytes base type. Infra is able to have packing metadata associated with the byte array to inform how to decode it, but since Proto is built to only assume that any reader of the data also has the corresponding proto definition on hand, we make the same assumption during this phase of our comparison.

8.2 Proto-Definition Encoding

Infra defines one encoding while Proto defines three: its C-like .proto definition language, its wire encoding, and its JSON-like ‘debug’ strings. The .proto definitions and debug strings are human readable, the former is meant to be authored directly, and the latter is meant only for reading. The following diagram outlines how various formats layer their encoding semantics. The three Protocol Buffer encodings are grouped on the right. Other familiar formats are included for context and comparison.



The second row describes a kind of alternate reality, where Infra is the encoding foundation of these formats instead of UTF-8 or a one-off binary scheme. In this arrangement, formats such as HTML and CSS are simply metadata channels on Infra-encoded data structures. Metaformats such as JSON, XML, and Protocol Buffers would have much less reason to exist if Infra were an existing baseline offering both the high performance of binary encodings and even greater human-readability than textual encodings. Thus, vanilla Infra is shown as equivalent rather than imagining JSON, XML, and Protocol Buffers semantics on top of Infra. Infra can mimic any one of the three encodings defined by Proto simply by virtue of what layer(s) of information is included in a structure.

- Infra instead of **PB’s proto**: a data structure acting as a prototypical data instance (data that can later be used as schema metadata)
- Infra instead of **PB’s wire**: a stream of data instances containing no copy or reference to a prototypical instance (data without schema metadata)

- Infra instead of **PB's debug**: a stream of data instances with a copy or reference to a prototypical instance as schema metadata

9 Related Work

The problems that Infra aims to solve require expanding the notion of human-readability beyond only character codes by generalizing it on two levels: encoding and editing. As a result, our contributions touch several domains. Existing metaformats are related work because we provide a general-purpose data encoding upon which higher-level formats can be built. Structured Editors are related work because we provide structure-informed display and editing tools. Software systems that merge programmable presentation with computational elements are related work because our goal is to scale seamlessly from raw data to high-level user interfaces within the same medium. We have organized related work by domain.

Textual Metaformats Human-readable metaformats such as XML [6], JSON [5], Comma-Separated Values (CSV) [25], and Lisp's S-Expressions [22], are the most popular and supported formats for structured information. Yet, it would be a far cry to imagine any one of them ever becoming the universal default for all text written by all end-users. In other words, it would be overly dysfunctional if every form field, search query, and command-line expected all users to type only valid XML structures. Even in the case of wide standardization, providing structured editors to abstract away the syntax would not really be worth doing since the encoding is not machine-friendly to begin with.

Binary Metaformats Binary metaformats such as Abstract Syntax Notation One (ASN.1) [16], Thrift [27], Google's Protocol Buffers [8], Cap'n Proto, MessagePack [12], Binary JSON [7], and Extensible Binary Markup Language (EBML) [23], swing so far in the other direction that they forgo the ability to be easily edited at all. The vast majority are explicitly designed as RPC frameworks, prioritize only byte-efficiency, and require predefined schemata or at least formal field names before any data can be encoded. Even if one of these formats had editors that would help them mimic freeform editing, few of them are designed to encode graphs and none of them support recursive metadata or fragmenting a block of memory with unallocated byte-gaps between elements. The former is critical for data-driven processing and the latter is a critical part of all programming language runtime heaps for tracking dynamically allocated memory.

Structured Editors Structured editors have a long history of repeated attempts to assist users in the syntactic tasks of editing formal languages. Systems from the 70s and 80s, such as Emily [14], Gandalf [13], Centaur [3], as well as contemporary systems such as Subtext [10], TouchDevelop [28], and

Prune [19], are billed solely as source code editors specializing in a particular language. JetBrains' Meta Programming System (MPS) [18], has generalized this by using meta grammars to allow the same system to be used to program in additional languages. However, structured editors end up limiting themselves by being so high-level. They constrain edits to prevent data from ever entering grammatically invalid states. In these systems, source code must be modified such that it is compilable before and after each atomic edit. This is cited as the main cause of the usability problems that have historically plagued structured editors [20]. Programmers' editing habits routinely find that the path of least resistance for compound edits passes through grammatically invalid intermediate states. This issue is amplified further in *Source Code in Database* (SCID) systems that customize even their storage representation to a particular language. Syntactically invalid programs do not have a way of being represented. Intentional Programming [26] utilizes such a system. In contrast, our goal with Infra is to apply the concept of structured editing to a general-purpose metaformat that languages can be built upon.

Programmable User-Environments Programmable user-Environments such as The On-Line System (NLS) [11], the Smalltalk and Squeak User Environments [15], Berkeley's Boxer Project [9], and Wolfram's Computable Document Format (CDF), all empower end-users with authorship of and access to the descriptions responsible for the entities and abstractions present in front of them. However, each still use text characters as their only fundamental building blocks within those descriptions. This means that their 'liveness' and helpful abstractions bottom out at the source code level. This is unfortunate for users and developers alike because both programming learning curves and API documentation could benefit from those properties - recursively. A partial exception to this is Boxer, the spiritual successor to Logo [24], which includes 'boxes' of three varieties³ as distinct primitives in addition to text to give structure to raw data and source code statements. This still leaves out basic types like binary-encoded quantities, which are critical to the internal representations in all software systems.

10 Backwards Compatibility and Adoption

Though the ultimate goal of Infra is to be a better *alternative* to the classical plaintext infrastructure rather than a *supplement* to it, Infra can still provide value along side existing technology. The following paragraphs each outline progressively deeper adoption scenarios.

Infra-Agnostic Use An Infra library can offer a suite of parsers (and renderers) to convert existing flat text data to structured dialects for easier manipulation and to render

³'Data', 'Doit' (code), 'Port' (transparent reference)

them back out in the original encoding. This category includes using Infra merely as an alternative to JSON, XML, YAML, and Protocol Buffers, for encoding and transferring data.

Stand-Alone Use A fully isolated use case is to use Infra as a generalization of spreadsheets. Beyond the usual grid of flat (unstructured) values, its semantics offer trees of any depth. Infra’s Patch references function much like cell formulae do.

Semi-Integrated Use Using Infra as an alternative to XML or YAML, Infra can serve as a beneficial format for directly-editable configuration files. Also, since metadata can be used to include CSS markup, Infra can bring richer web-like presentation and interaction to textual system reports, such as in console logs.

Transparent Adoption If Infra widgets replaced text field UI widgets, but continued to be used only for plaintext uses, Infra could be made to be ‘invisible’ while its extra capacities were available on an optional basis. The widget API only needs to offer automatic ‘flattening’ to plaintext strings (perhaps dropping metadata and concatenating list items using spaces) for applications that must use the entered data in a strictly character-array form. This would allow Infra to be leveraged more and more over time with a pre-existing install base.

Social-Media Adoption If a user-centric team-communication app (e.g. Google Hangouts, Slack, FlowDock, Twitter) were to adopt Infra as its medium, the situation would be similar to the ‘transparent adoption’ above, but would take place within the context of a community that is accustomed to authoring and exchanging constructive elements beyond text on a regular basis.

Neo HTML Infra itself is a more presentable markup language than HTML, and could be used as an always-binary-encoded alternative. Either, Infra editors can be used in the role of a web browser, or Infra extensions could be written for existing web browsers. This would unify the concepts of viewing the source and viewing the rendered page. Parsed dialects of CSS and Javascript content can exist in metadata. HTTP headers can also be switched to be Infra-based. Since the majority of an HTTP header consists of numerical values, this would save an especially appealing amount of parsing and byte overhead.

Post-Web Web Stack Since Infra editors are already similar to web browsers, and Infra’s encoding comprises a markup language, binary transfer format, and dynamically allocated heap memory (using the ‘Free’ element type), it is a small leap to use Infra for every layer of a typical web stack, including database storage.

Patch semantics lend themselves well to mimicking an HTTP request. With the addition of a new opcode for ‘navigating’ the virtual cursor to a remote host, a Patch can

continue executing on the remote host, acting exactly like a URL as it specifies the rest of the path with the rest of its instructions to name a resource. The Patch result is the reply from the host, performing the equivalent of a GET request. And a Patch that performs modifications would be the equivalent of a PUT, POST, or DELETE request.

To complete Infra’s ability to act as a back-end database, an additional native service object can be added to act as a database driver.

Full Adoption With full adoption we envision a compiled programming language designed with infra at its core. Infra would make up the code, the data structures, and possibly the heap. New programming language semantics would be invented that leverage Infra and Patch.

11 Conclusion

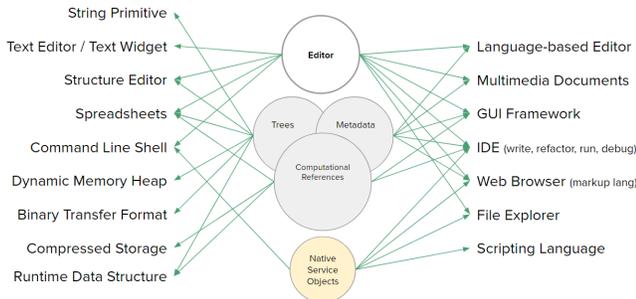
Infra is designed to make working with data more direct, consistent, and efficient for both humans and computers. Before Infra, developers had to either choose a human-readable format for their data and forgo processing simplicity and efficiency, or choose a binary format and forgo human readability. With Infra, developers can have both. Furthermore the organization Infra brings to data makes Patch viable as a new type of programming language targeting the domain of in-stream data metaprogramming.

In summary, Infra defines a dozen base types, a handful of Patch opcodes, a range of native-service interfaces, and an effect system to mediate side effects so that untrusted data can at least perform computation while trusted data can be useful for general-purpose programming. It provides the components needed to provide modest free-form data input widgets that can scale into a user environment and programming environment on a whim (since Patches can exist anywhere and metadata can embellish any data with appropriate editor abstractions). A capacity for structure, metadata, and computation can become an ambient part of everyday computer usage.

Infra, as a single lightweight tech stack, can be leveraged to various degrees to perform in a wide variety of critical roles across the computing landscape. To highlight a few, Infra can provide:

- text-editor-like availability for users to quickly read, write, and manipulate Infra-encoded information
- a spreadsheet-like experience: by displaying hierarchical data in a tabular layout, Patch’s data-flow-like programming model functions like cell formulae
- a rich-text document-like experience: data-driven styling and presentation (via ‘CSS’ metadata)
- an IDE-like experience: data-driven structure and type checking (via ‘schema’ metadata)
- a GUI application-like experience: high-level interfaces and abstractions of specific data models (via ‘format’ metadata)

- a Web-browser-like experience (via ‘HTML’ metadata): Infra naturally provides the opportunity for a binary-HTML based Web (and binary-HTTP headers), but in the meantime, a traditional parsing step can be used to interface with the existing Web.
- a command-shell-like experience: the Native-Service Objects for operating system integration allow inline browsing of the file system, invocation of executable files with input arguments, and standard I/O streams (mediated by the effect system).



We find that Infra, as a medium, enables opportunities to bring richer direct-manipulation and user-interface support to much lower-level layers of computing than are normally available.

11.1 Closing Statements

Because a structured encoding opens the door to explicitly-demarcated metadata, and structured metadata opens the door to new heights of extensibility, we believe Infra is the best kind of root-striking evolution computing can make right now. In an ever more connected world with accelerating trends towards ubiquitous computing, Internet of things, semantic web[2], and an active push for broader cultural reach in Computer Science education, Infra is well poised to make the needed kind of impact. Infra raises computing’s currently-low bar for what is expected from the rawest-of-the-raw tokens of encoding, in terms of their ability to standardize the building blocks of structured (and interdependent) information, as well as the richness of the means for interacting with them.

A Transcript for Video 1 - Building Blocks

Let me demonstrate the nature of authoring data in our prototype editor for Infra.

Infra is a novel binary-encoding that can be edited as freely at text, while providing the capacity to express arbitrarily complex structure, computation, and include metadata.

Our prototype editor is meant to look and feel like a text editor so that it can be used everywhere that users would normally interact with text, but is actually working with machine-friendly data structures.



In this case, we have created a list of nine UTF8 encoded strings. The spacebar naturally tokenized our sentence as we typed it, rather than writing a space character like it normally would in a text editor.

Let me paste in some text from the human-readable-format comparison example in the paper. This is the equivalent data structure, represented in JSON... Lisp... and Python.

```
["fish", {"1":"red", "2":true}]
('fish ((1 . 'red)(2 . t)))
['fish', {1:'red', 2: True}]
```

It is some simple structured-data that covers a range of syntactic elements: Strings, numbers, lists, key-values, and a boolean value. For all of the similarity in their structure, they are far from being interchangeable between parsers.

Now let’s type the same structure using Infra.

```
["fish", {"1":"red", "2":true}]
('fish ((1 . 'red)(2 . t)))
['fish', {1:'red', 2: True}]
fish 1:red 2:True
```

In this case, the values, 1 and 2, are not being encoded as text characters. They are binary encoded integers. And this True has been authored explicitly as Infra’s boolean symbol for true.

```
["fish", {"1":"red", "2":true}]
('fish ((1 . 'red)(2 . t)))
['fish', {1:'red', 2: True}]
fish 1 | red
      2 | True
```

With Infra, data can stay parsed in this way right from the time of authoring. This means that editor UIs can have richer, more precise, interactive dialog with the author about the content, its parts, metadata associated with its parts, inter-relationships, and its presentation.

Let’s add metadata to this string element to give the editor more confidence that this is in-fact JSON.

```
format: JSON
["fish", {"1":"red", "2":true}]
('fish ((1 . 'red)(2 . t)))
['fish', {1:'red', 2: True}]
fish 1 | red
      2 | True
```

This editor happens to have a JSON parser and supports metadata labeled as “format” metadata, so here we can see the editor now offering to parse it as JSON.

```
format: JSON
["fish", {"1": "red", "2": true}]
('fish' ((1: 'red')(2: t)))
['fish', {1: 'red', 2: True}]
fish 1 | red
      2 | True
```

```
fish 1:red 2:True
```

```
('fish' ((1: 'red')(2: t)))
['fish', {1: 'red', 2: True}]
```

```
fish 1 | red
      2 | True
```

Next, to demonstrate more editing, let’s manually parse a second one of these representations. But first, let me rig up a side-by-side view of the byte encoding for this object so we can see how the representation changes with each keystroke.

We can drop the cursor down to character-editing level and start cleaning things up.

```
fish 1:red 2:True
('fish' ((1: 'red')(2: t)))
5F1D2F1B28276669736820282831202E2027726564292832202E2074292929
['fish', {1: 'red', 2: True}]
fish 1 | red
      2 | True
```

From Infra’s perspective, a traditional text editor is just an editor whose cursor is locked down at the character-level, only edits String objects, and has tunnel vision on one string at a time.

```
fish 1:red 2:True
```

```
fish 1:red 2:True
5F0F24666973685965E12372656462E2F1
['fish', {1: 'red', 2: True}]
fish 1 | red
      2 | True
```

Just like raw UTF8, Infra’s encoding is easy to make human friendly in editors, and usable by user-interface widgets

and libraries everywhere. All while rivaling the machine-readability of binary metaformats such as ASN.1 or Protocol Buffers.

We explore Infra further in other videos. Thank you.

B Transcript for Video 2 - Quantities

Unlike text, Infra is sophisticated in how it deals with numbers. As before, we’ve rigged up an encoding view on the right side. Let’s slowly type negative-one-thousand-point-five.

```
- 212D
```

At this point, this is a one character string.

```
-1 31FF
```

Now, infra recognizes "dash one" as a number and encodes it efficiently.

Its header byte signifies that the following byte be interpreted as an integer.

```
-10 31F6
```

By simply pressing zero, I’ve turned negative one into negative ten, while remaining binary encoded.

```
-100 319C
```

Now it’s negative 100. Now typing a third zero...

```
-1,000 32FC18
```

Negative one thousand. Because this is explicitly a number in the encoding, the editor can show digit grouping based on the user’s localization settings. Text formats often cannot tolerate commas in numbers because they use them as delimiters to separate list items.

```
-1000. 262D313030302E
```

Because the string representation of -1000 does not contain a decimal point, infra must revert to encoding this as a string. In general, numbers are tested for round-trip stability before being converted to a binary encoded number.

```
-1,000.5 44C47A2000
```

Now that the character 5 has been added, converting the string to a float and back to a string again results in the original string. Thus Infra can safely encode -1,000.5 as a floating point number.

The widely-used Grisu3 and Dragon4 [21] shortest-decimal-representation algorithms are used to reverse decimal-to-binary rounding loss when performing the round-trip test.

We explore Infra further in other videos. Thank you.

C Transcript for Video 3 - Patch Intro

This is a demonstration of the powerful Patch mechanism in Infra. Hello world!

```
Hello world
```

We have entered a list of two strings. I'm going to do a Copy, [pause] paste.

Hello world Hello world

We now have a patched shallow copy of the first list to the left. We see two views of the same Hello-world instance. Shift-ctrl-v would have resulted in a deep copy.

Hello world left 1

Here is what actually got pasted. It is a Patch program that reference the element to its left, like a spreadsheet formula.

This creates a live copy. A change to the original is reflected in both.

Here I'll add an exclamation point.

Hello world! Hello world!

On the other hand, when a Patch's result is edited, patch code is automatically generated to produce the equivalent change procedurally. The red recording light indicates this to the user.

Hello world! Hello world!

I'll add the word "structured".

Hello world! Hello structured world!

Note that the original is unchanged. Let's take a look at the Patch's code now.

Hello world! left 1
down 1 insert structured

This code references the left list and inserts the word 'structured' as the second item in the list.

Now let's change the original list again.

Goodbye world! Goodbye structured world!

Not only was the Patch output updated, but it retained its procedural edits as well. The patch mechanism provides not only spreadsheet-like recombination and modification of source data, but allows those modifications to be synthesized through direct manipulation. The combination of these two features can give end-users a simple way to take advantage of programming for productivity purposes.

This kind of language-independent data refactoring is virtually impossible in textual formats. In more elaborate cases, an Infra editor has everything it needs to provide ever more elaborate degrees of programming by demonstration, and programming by example.

This scratches the surface of what the Patch mechanism brings to Infra as an end-user medium and integrated programming environment.

We explore Infra further in other videos. Thank you.

Acknowledgments

This work was partially supported by U.S. ARO MURI grant No. W911NF-09-1-0553, as well as ONR grant N00014-14-1-0133.

References

- [1] Umut A Acar, Guy E Blelloch, and Robert Harper. 2002. *Adaptive functional programming*. Vol. 37. ACM.
- [2] Tim Berners-Lee, James Hendler, Ora Lassila, and others. 2001. The semantic web. *Scientific american* 284, 5 (2001), 28–37.
- [3] Patrick Borrás, Dominique Clément, Th Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and Valérie Pascual. 1989. *Centaur: the system*. Vol. 13. ACM.
- [4] Thomas Boutell. 1997. PNG (Portable Network Graphics) Specification Version 1.0. (1997).
- [5] Tim Bray. 2014. The javascript object notation (json) data interchange format. (2014).
- [6] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. 1998. Extensible markup language (XML). *World Wide Web Consortium Recommendation REC-xml-19980210*. <http://www.w3.org/TR/1998/REC-xml-19980210> 16 (1998).
- [7] cc. 2008. Binary JSON. (2008).
- [8] Jeffrey Dean and Sanjay Ghemawat. 2010. MapReduce: a flexible data processing tool. *Commun. ACM* 53, 1 (2010), 72–77.
- [9] Andrea A. diSessa and Harold Abelson. 1986. Boxer: a reconstructible computational medium. *Commun. ACM* 29, 9 (1986), 859–868.
- [10] Jonathan Edwards. 2005. Subtext: uncovering the simplicity of programming. In *ACM SIGPLAN Notices*, Vol. 40. ACM, 505–518.
- [11] Douglas C. Engelbart and William K. English. 1968. A Research Center for Augmenting Human Intellect. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I (AFIPS '68 (Fall, part I))*. ACM, New York, NY, USA, 395–410. DOI: <http://dx.doi.org/10.1145/1476589.1476645>
- [12] Sadayuki Furuhashi. 2014. MessagePack: ItãŽs like JSON. but fast and small, 2014. (2014).
- [13] A. Nico Habermann and David Notkin. 1986. Gandalf: Software development environments. *Software Engineering, IEEE Transactions on* 12 (1986), 1117–1127.
- [14] Wilfred James Hansen. 1971. Creation of hierarchic text with a computer display. (1971).
- [15] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the future: the story of Squeak, a practical Smalltalk written in itself. In *ACM SIGPLAN Notices*, Vol. 32. ACM, 318–326.
- [16] ISO ITU-T. 2002. IEC: Abstract Syntax Notation One (ASN. 1) Specification of Basic Notation. *Report no. ITU-T Rec. X 680 (2002)*, 8824–1.
- [17] J. M. Dlugosz. 2003. Dlugosz' Variable-Length Integer Encoding - Revision 2. (2003). <http://www.dlugosz.com/ZIP2/VLI.html>.
- [18] MPS JetBrains. 2014. Meta Programming System. (2014).
- [19] K. Beck. 2015. Prune: A Code Editor that is Not a Text Editor. (2015). <https://www.facebook.com/notes/kent-beck/prune-a-code-editor-that-is-not-a-text-editor/1012061842160013>.
- [20] Andrew J Ko, Htet Htet Aung, and Brad A Myers. 2005. Design requirements for more flexible structured editors from a study of programmers' text editing. In *CHI'05 extended abstracts on human factors in computing systems*. ACM, 1557–1560.
- [21] Florian Loitsch. 2010. Printing floating-point numbers quickly and accurately with integers. *ACM Sigplan Notices* 45, 6 (2010), 233–243.

- [22] John McCarthy. 1960. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM* 3, 4 (1960), 184–195.
- [23] Martin Nilsson. 2004. Extensible Binary Markup Language. *Draft specification, Matroska* (2004).
- [24] Seymour Papert. 1980. *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc.
- [25] Yakov Shafranovich. 2005. Common format and MIME type for Comma-Separated Values (CSV) files. (2005).
- [26] Charles Simonyi. 1995. The death of computer languages, the birth of intentional programming. In *NATO Science Committee Conference*. 17–18.
- [27] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. 2007. Thrift: Scalable cross-language services implementation. *Facebook White Paper* 5, 8 (2007).
- [28] Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, and Manuel Fahndrich. 2011. TouchDevelop: programming cloud-connected mobile devices via touchscreen. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*. ACM, 49–60.