Jeffrey Benjamin Brown
2016 June 15

The Reflective Set of Labeled Tuples: A simple, powerful new data structure

Introduction

Computer scientists spend a lot of time finding ways to represent problems using a simple, small vocabulary called Data Structures. They include:

- Nothing.
- Numbers.
- Arrays, also known as tables or grids.
- Tuples, meaning pairs (“2-tuples”), triples (“3-tuples”), quadruples, etc.
- Sets, also known as collections.
- Lists, which are like sets, but with their elements ordered first to last.
- Trees, also known as nested lists.
  - To nest data structures is to put one inside another.
- Maps, also known as dictionaries.
- Graphs, also known as networks.

That's about all of them!

This paper introduces another one, only slightly more complex than a graph. It is the Reflective Set of Labeled Tuples, or RSLT.[1]

Whereas other data structures are limited in what they can express, the RSLT can express literally any fact.

Natural languages like English or Spanish can express any fact, too. The RSLT relies on natural language. However, unlike ordinary text, the RSLT is quickly traversable, like a tree, along every connection a user could possibly want. And it is queriable in the infinitely flexible way of a graph[2].

I have implemented the RSLT, and open sourced it, on Github[3]. I used the Haskell programming language[4] and the Functional Graph Library[5][6], both of which deserve the highest praise. This paper, however, requires no familiarity with them, or indeed with computer programming at all.

---

1 I’ve been pronouncing it "RISS-ult", like “whistled”.
2 See Neo4j for an example of a beautiful language for asking questions about graphs..
3 https://github.com/JeffreyBenjaminBrown/digraphs-with-text
4 https://www.haskell.org/
5 https://hackage.haskell.org/package/fgl-5.5.2.3/docs/Data-Graph-Inductive-Graph.html
6 Special thanks to Elliot Cameron, who recognized that the RSLT could be implemented using an existing graph library.

After explaining the problems with existing data structures that the RSLT addresses, this paper describes what the RSLT is, from both a user's perspective and a coder's. Then it shows how a coder can implement the RSLT using a graph library. Last it describes some important future avenues of development.

Table of Contents

The problem: Why we need another data structure

The RSLT is a data structure -- that is, a vehicle for organizing, connecting, querying and viewing information. Unlike other data structures, the RSLT can encode literally any kind of information, using the same language, in the same format, in the same file.

The RSLT is a simple generalization of the graph. “Graph” is what mathematicians call a collection of dots connected with lines. See Figures 1 and 2 for examples.

In the RSLT there is only One Natural Encoding of every fact. This allows the RSLT to eliminate redundancy, eliminate the need for duplicate data. (Imagine – no more duplicate, redundant data!)[7] A fact can be available in every relevant context even though it is recorded only once. Edits made in any context "propagate" immediately "to" the other contexts, because there is no distance to cross, there are no copies to coordinate.

The RSLT eliminates the distinction between data (e.g. numbers in a table) and metadata (e.g. facts about the table that don’t themselves fit in the table). Data is data – be it names, numbers, implications, instructions, warnings, authorship, time of creation, time of intended completion, requirements, subgoals, arguments, relationships between arguments, privacy instructions for the data itself, etc. They are all encoded the same way, and all queriable (separately or jointly) the same way.

The RSLT is easy to learn. It uses ordinary natural language like English or Spanish. All it requires is an explicit understanding of the geometry[8] of how sentences divide into subexpressions. This is because the One Natural Encoding of a fact is exactly the sentence tree diagram for that fact.

Of course, there can be different ways to say the same thing in natural language. Reorderings can leave a statement’s meaning unchanged: "Sam I am" and "I am Sam" differ in presentation but not in their meaning. The RSLT gets around that problem by allowing a user to record order and meaning as separate information. Order and meaning might be entangled in ordinary text, but in a RSLT they do not need to be.

Because there is only One Natural Encoding of every fact, merging RSLT data is trivially easy.

The RSLT is low-cost. To access your existing data using a RSLT interface, the data does not need to change. It just needs to be hooked into the interface. Any space or speed optimizations in the data can be retained. See the section called “Hooking other, spacetime-optimized data structures into the RSLT.”

Metadata and the expressivity limits of data structures: A real-world example

Data scientists agree: Metadata is ubiquitous and dangerous. With the RSLT, no data is meta. Data is just data.

As an example, I once worked with medical data for economic research at Precision Health Economics, a consulting firm[9]. We dealt in facts like these:

(1) Patient X spent 30 days in the hospital.
(2) Institution Y provided the data for statement 1.

---

7 This is a joke.

8 I say “geometric” because you don’t actually have to know the names of parts of speech. To encode a statement like “x uses y”, you don’t have to know that “uses” is the verb, or that “x” is the subject, or that “y” is the object. You only need to know that “uses” connects “x” to “y”.

9 http://www.precisionhealtheconomics.com/

(3) Previous patient-stay data from Institution Y have reported numbers greater than 30 as 30.
(4) Statements 2 and 3 imply that statement 1 might be inaccurate. The true figure might be greater than 30.

We would encode statements like (1) in a table. Tables are simple, uniform, and easy to process automatically. Statements like (2) through (4), by contrast, are "metadata" -- data about the table, which do not themselves fit into the table.

Economists and statisticians typically retain metadata in an ad-hoc manner. If things go well, it will be stored in a good place, and later read by the right people, so that the data can be treated as the metadata describes. But because it uses a different format (in our case, usually ordinary English), and because it exists separate from the data, the metadata is easily lost or misunderstood.

Moreover, unlike the table, the metadata is hard to process automatically. To do that, one has to write custom code for the particular metadata at hand – an expensive, error-prone undertaking.

Precision Health Economics brings medical doctors, economists, statisticians, and computer programmers together to create papers for publication in academic journals of medicine and economics, often top-ranked ones. Our metadata problems arose through no lack of expertise; they were inherent to the data structures available to us. It was one instance of a widespread, endemic representation problem – which the RSLT solves.

The expressivity limits of data structures

The next three subsections elaborate, respectively, these three facts:

(1) Information has order. Every statement is an order-*n* relationship, for some value of *n*.
(2) Information can be of any order. There is no natural upper limit to the possible value of *n*.
(3) Other data structures cannot naturally encode relationships of order greater than 2.

(1) Information has order

Information has order. That is not a statement about computer science; it applies to all information.

The "first-order" expressions possible in a natural language are the indivisible, atomic ones. They are usually single words, like *airport* or *mango*. (They could be longer phrases, like *Charlie Parker* or *Papua New Guinea*.)

Higher-order expressions are constructed by putting first-order expressions in relationships. An example of a second-order expression is *Bob uses aspirin*. It relates the two first-order expressions *Bob* and *aspirin* through the relationship "_ uses _" (pronounced “blank uses blank”). We say that *Bob* and *aspirin* are “members” of the relationship, and *uses* is the “label” for it.

(2) Information can be of any order

Every sentence is an expression of some order. An example of a third-order expression is *Bob uses aspirin because Bob gets migraines*. It consists of a "_ because _" relationship between the two second-order relationships *Bob uses aspirin* and *Bob gets migraines*.

In general, the order of a relationship is equal to one plus the greatest order of its subexpressions.

Every sentence, every statement, every utterable fact, is an *n*th-order relationship, for some value of *n*. Every (positive integer) order is valid; there is no natural limit to the potential depth of an expression.

(3) Other data structures max out at a low order

The popular data structures – sets, maps, lists, arrays – are each perfect for something. None of them, however, is as expressive as the graph.

So let's consider the graph – the current state of the art in general-purpose knowledge representation. Google uses a graph to generate search results. Facebook uses a graph to reason about things like users and advertisers. Graphs are powerful – but they cannot naturally[10] express third-order relationships.

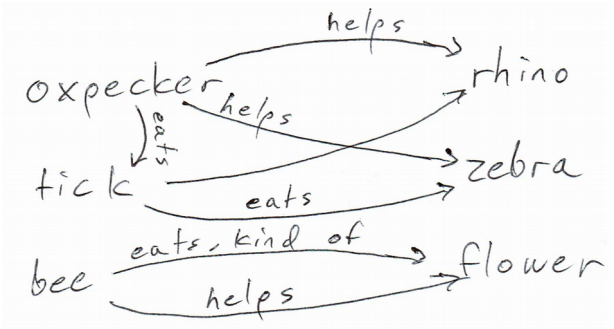

Figure 1: Mutualism among organisms

A graph is a set of "nodes" , which can be anything, and "edges", which are lines or arrows connecting the nodes. The nodes are things, and the edges are relationships between things. Edges can (and, in most data science contexts, should) have labels, like "_ needs _" or "_ knows _".[11]

Figure 1 is a graph. Its nodes include *rhino*, *zebra*, *oxpecker*, *tick*, *bee* and *flower*. Its two edge labels are *helps* and *eats*. Ticks help rhinos and zebras by eating ticks. Bees help flowers, by eating (a derivative of) the flowers.

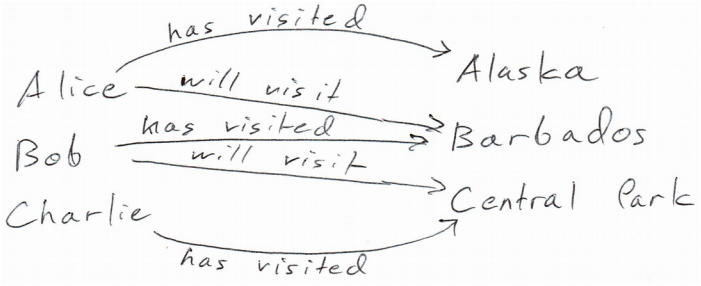

Figure 2: Travelers and destinations

Figure 2 is another graph. Its nodes represent travelers and destinations, and its edges represent two kinds of relationships: “_ has visited _” and “_ will visit _”.

In a graph (as they are commonly used) the nodes represent first-order expressions like "Bob" or "$60,000", and the edges represent second-order expressions like "Bob spent $60,000". The power of a graph comes from its ability to represent multiple kinds of first and second order expressions. Less technically, graphs are useful because one of them can represent many kinds of things *and* many kinds of relationships between pairs of things.

Edges in a graph connect nodes; they cannot connect other edges. Therefore third- and higher-order relationships cannot be naturally expressed in a graph. That is a severe limitation, because most statements are of an order greater than two, because most sentences are more complex than "puppies are fuzzy".

10 The word “naturally” is important here. Its meaning is elaborated shortly.

11 Sometimes, for readability, I will omit the blanks from a relationship. At least in binary relationships, such omission is unambiguous.

That is why we need the RSLT.

Is a graph *really* not expressive enough? Natural and unnatural representations.

There is a certain sense in which the RSLT is not necessary. In this sense, a graph provides more than enough capacity to hold all the data you could want. So does a list. In fact a *single number* can encode any finite collection of facts – the complete works of Shakespeare, for instance. Kurt Gödel proved this, on the way to proving his incompleteness theorem.

The problem is that the number is hard to read. It is *possible*, but it is not *natural*.

I have no explicit definition of naturality to offer. What I can say, though, is that nobody seems to have shown how to naturally encode an arbitrary collection of facts in a list, tree, graph, or any other data structure.

By contrast, this paper demonstrates how, with an RSLT, it is possible.

The RSLT: Definition and implementation

The RSLT is a simple data structure. To understand this paper, you do not need to know any programming language. However, it bears pointing out that the RSLT can be defined in just four lines of Haskell:

```
import Data.Graph.Inductive
type RSLT = Gr Expression RSLTEdge
data Expression = Word String | Relationship | Template [String]
data RSLTEdge = UsesTemplate | UsesMember Int
```

That code is available online[12], with explanatory comments.

Words, Templates and Relationships

The RSLT contains only three kinds of things: Words, Templates and Relationships. In this paper they are capitalized to distinguish them from their ordinary usage – that is, a "Relationship" is part of a RSLT, whereas a "relationship" is information outside of any digital context.

The Words are the atomic, first-order expressions of a RSLT. Examples include "cat" and "France".

The Templates define the kinds of Relationships possible. "_ is a _" is an example of a Template. Others include "_ produces _" and "_ said _ to _". By itself, a Template does not relate anything; rather, it defines a way that things can be related.

12 https://github.com/JeffreyBenjaminBrown/digraphs-with-text/blob/master/introduction/Minimal_Types.hs

A Relationship consists of a Template and some Members. Each blank in the Template is assigned a Member. For instance, by starting from the Template "_ need _", we can construct the Relationship "cats need water" by making "cats" the first Member and "water" the second.

At this point I can explain the acronym RSLT. It stands for Reflective Set of Labeled Tuples. The Templates provide the labels, and the Relationships are the tuples. A Member of a tuple might simply be a Word, but it might itself be another Relationship. The latter possibility is what makes the RSLT reflective[13].

Arity

The "arity" of a relationship is defined as the number of things it relates. An arity-2 ("binary") relationship relates two things; an arity-5 relationship relates 5 things.

The number of blanks in a Template determines its arity. Because most relationships are binary, most Templates have two blanks. However, a Template can as easily relate three (for instance, "_ gave _ to _") or more things. (This provides another sense in which the RSLT generalizes the graph: In a graph, every relationship is binary. However, as described in Appendix A, it is the RSLT's reflection, not its general arity, that makes it so expressive.)

A Template can also be unary; that is, it might only have one Member. Some important unary Templates include "not _", "maybe _", "some _", and "every _".

Relationships also have arity. The arity of a Relationship is the number of Members in it, which is equal to the arity of the Template that the Relationship uses.

Implementing the RSLT using a graph

The following example will clarify the definition that follows it.

Only a coder would need to know about the graph behind a RSLT. A user would need to understand Words, Templates and Relationships, but not nodes or edges.

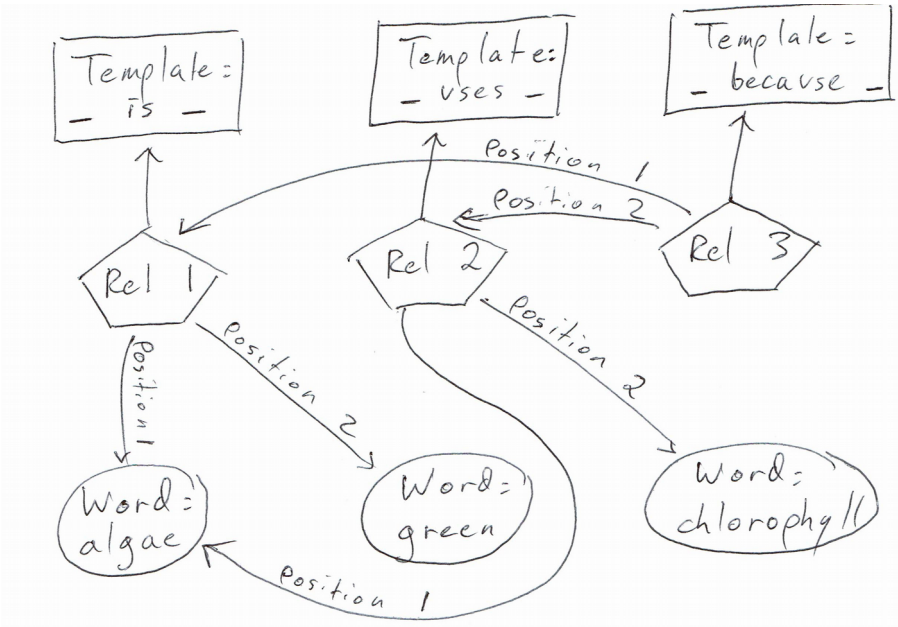

Figure 3: Why Algae is green: A RSLT as a graph

13 In computer science, reflection generally is the capacity for a program to view or modify itself. In the special context of a data structure, it is the capacity for one item in it to refer to another.

Example: Why algae is green

Figure 3 depicts a RSLT implemented using a graph. The Words (ovoids), Templates (rectangles) and Relationships (pentagons) are all nodes in the graph.

Relationship 1 encodes the statement “algae is green”. It uses the “_ is _” Template. Relationship 1 assigns “algae” to the first blank of its Template, and “green” to the second.

Relationship 2 encodes the statement “algae uses chlorophyll”. It uses the “_ uses _” Template, assigning “algae” to the first position and “chlorophyll” to the second.

Relationship 3 encodes the statement “algae is green because algae uses chlorophyll”. It uses the “_ because _” Template, assigning Relationship 1 to the first position and Relationship 2 to the second.

Relationships 1 and 2 are second order expressions, because their Members are all Words, which are first-order expressions. Relationship 3 is a third-order expression, because its Members are second-order expressions.

The general specification

The nodes in a RSLT graph are expressions. Each expression is either a Word, a Template, or a Relationship. Word and Template nodes carry text. Relationship nodes carry no information; it is the edges emitted from a Relationship that give it meaning.

Every edge in the RSLT graph points from a Relationship to something the Relationship uses. There are two kinds of edges. One kind runs from a Relationship to its Template, and the other runs from a Relationship to one of its Members (which might be a Word or a Relationship). That second kind of edge comes with a number indicating which position in the Template the Member belongs to. Every arity-*k* Relationship emits *k* edges, one to edge to each of its *k* Members. Each Relationship emits one additional edge to its Template, which must also have arity *k*.

Edges do not point from Words or Templates, only to them.

Encoding high-order facts in a RSLT

The fast way: Nested # notation

Suppose we wanted to encode the following facts in a RSLT:

Data Set 1

Mildred has stable angina.
Mildred is a person.
Stable angina is a coronary artery disease.
Every person with coronary artery disease needs monitoring.

All we need is for the computer to understand how those facts divide as sentence trees. Nested parentheticals offer a simple text-based input method for making that division explicit:

Data Set 1.1

(Mildred) has (stable angina).
(Mildred) is a (person).
(Stable angina) is a (coronary artery disease).
(Every ((person) with (coronary artery disease))) needs (monitoring).

If the transformation from Data Set 1 to Data Set 1.1 looks tricky, fear not – there exist good (not perfect, but very good) automatic sentence parsers already[14].

Many programmers find heavily nested parentheticals hard to read. Here is an equivalent text-based representation of those facts, “nested # notation”, with a lower punctuation burden:

Data Set 1.2

Mildred #has stable angina.
Mildred #(is a) person.
Stable angina #(is a) coronary artery disease.
##Every person #with coronary artery disease ###needs monitoring.

The # symbol adjacent to an expression indicates that the expression is a Relationship label. Every other expression is a Relationship Member. In the last statement, the varying number of # marks indicates the precedence of Relationship labels: #with operates first, ##every operates second, and ###needs operates last.

Nested # notation works equally well with higher-arity Relationships[15].

Nested parentheticals and nested # notation both allow the computer to generate an expression’s subexpressions automatically, or find them and use them if they already exist in the RSLT.

The slow way: Building expressions from the bottom up

In principle, all that is needed to enter Data Set 1 is to type the four sentences in Data Sets 1.1 or 1.2. I have not yet written that input method. (I have implemented nested # notation, but so far only as an output method – see below.)

The existing interface for entering data is slower to use, because it requires building each subexpression before creating a superexpression that uses them.

Because it is awkward and temporary, the purpose of this section is not to explain the existing interface. I just want to demonstrate that the data can be represented.

(You can find the following code online[16], and with a Haskell interpreter like GHCI, run it yourself.)

First, let's create an empty RSLT named "g":

```
> let g = empty :: RSLT
```

14 One excellent example can be tested online at http://www.link.cs.cmu.edu/link/submit-sentence-4.html
15 For example, “Che ###used markers #and paper ##from China ###to write #about China.”
16 https://github.com/JeffreyBenjaminBrown/digraphs-with-text/blob/master/introduction/demo.hs

Next, let's insert into g the five Templates we will need:

```
> g <- pure $ foldl (flip insTplt) g
    ["every _"
 ,"_ is a _"
 , "_ has _"
 ,"_ with _"
 , "_ needs _"]
```

Now let's look at g:

```
> view g $ nodes g
(0,"#every _")
(1,"_ #(is a) _")
(2,"_ #has _")
(3,"_ #with _")
(4,"_ #needs _")
```

Each expression appears alongside its address. For instance, the unary "every _" Template resides at address 0, and the binary "_ needs _" Template is at address 4.

Next, lets put our first order expressions into the RSLT: the Words "person", "Mildred", "stable angina", "coronary artery disease", and "monitoring". (For these purposes "coronary artery disease" and "stable angina" are single Words.)

```
> g <- pure $ foldl (flip insStr) g
     ["person",  "Mildred",  "stable angina"
  ,  "coronary artery disease",  "monitoring"]
```

The new Words are inserted after the Templates:

```
> view g $ nodes g
(0,"#every _")
(1,"_ #(is a) _")
(2,"_ #has _")
(3,"_ #with _")
(4,"_ #needs _")
(5,"person")
(6,"Mildred")
(7,"stable angina")
(8,"coronary artery disease")
(9,"monitoring")
```

Now that we have both Templates and Words, we can start adding Relationships. First let's encode that Mildred is a person and stable angina is a coronary artery disease:

```
> g <- pure $ fromRight $ insRel 1 [6,5] g
  -- creates node 10, "Mildred is a person"
> g <- pure $ fromRight $ insRel 1 [7,8] g
  -- creates node 11, "stable angina is a coronary artery
disease"
```

The first of those two commands says "insert into g a Relationship that uses the Template at address 1 ("_ is a _"), where the Members are the expressions at address 6 ("Mildred") and 5 ("person"). The second command is similar.

Rather than showing all of g, let's just look at the Relationships in it:

```
    > view g $ nodes $ labfilter isRel g
    (10,"Mildred ##(is a) person")
    (11,"stable angina ##(is a) coronary artery disease")
```

Expressions 10 and 11 are the two new Relationships. In both of them, the ## symbol identifies the Relationship label "is a", on either side of which appear the two Members.

Last, let us encode the idea that every person with coronary artery disease needs monitoring:

```
    > g <- pure $ fromRight $ insRel 3 [5,8] g
      -- creates node 12: "person with coronary artery disease"
    > g <- pure $ fromRight $ insRel 0 [12] g
      -- creates node 13: "every person with coronary artery disease"
    > g <- pure $ fromRight $ insRel 4 [13,9] g
      -- creates node 14: "every person with coronary artery disease
needs monitoring"
```

Here is the final RSLT:

```
    > view g $ nodes g
    (0,"#every _")
    (1,"_ #(is a) _")
    (2,"_ #has _")
    (3,"_ #with _")
    (4,"_ #needs _")
    (5,"person")
    (6,"Mildred")
    (7,"stable angina")
    (8,"coronary artery disease")
    (9,"monitoring")
    (10,"Mildred ##(is a) person")
    (11,"stable angina ##(is a) coronary artery disease")
    (12,"person ##with coronary artery disease")
    (13,"####every person ##with coronary artery disease")
    (14,"####every person ##with coronary artery disease
########needs monitoring")
```

Recall that the number of # symbols on a Relationship label indicates the order of that Relationship. Thus in expression 14, ##with binds first, then ####every, and last ########needs. I chose to make the number of # symbols start at 2 and increase exponentially (2, 4, 8, ...) so that visually parsing them would be easy: You can see which label has more # symbols without explicitly counting them.

Complexities and possible extensions

Subgraph Reflection: Directions, Stars and Branches

Direction

Some relationships, like "x equals y", "x is like y", or "x contradicts y" are undirected, symmetric. They treat x and y the same. A symmetric relationship can be reversed without changing the meaning.

Other relationships are directed or asymmetric. Examples include "x needs y", "x clarifies y", or "x is an instance of y". For each, you could usefully declare one member position Up and the other member position Down.

I implemented the Direction, and called it RelSpec. It is reified; that is, RelSpecs are another kind of node in the RSLT graph.

Stars

A star in a graph consists of a central node and a set of other nodes connected to it. The other nodes in the star are called leaves, because every star is a tree.

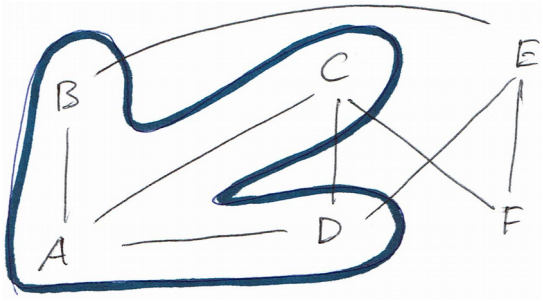

Figure 4: The star centered at A has leaves B, C and D

Figure 4 depicts an abstract graph. The thick blue line indicates a star that is a subset of the graph. The star is centered at A, and includes leaves B, C and D. E and F are not part of the star, because they are not connected to A.

Figure 5 depicts another graph. Every edge from X to Y indicates that X needs Y. A thick blue line is drawn around the star of things that need air. The robot and the Barbie doll, since they do not need air, are not part of the star.

In a RSLT, once you can assign Directions to Templates, you can ask to see a Star. A Star in a RSLT consists of all Expressions bearing some relation to a central Expression. Examples of Star queries include "every town with a power plant," “every project Jerry is a member of,” or "every rejection letter that demonstrates kindness".

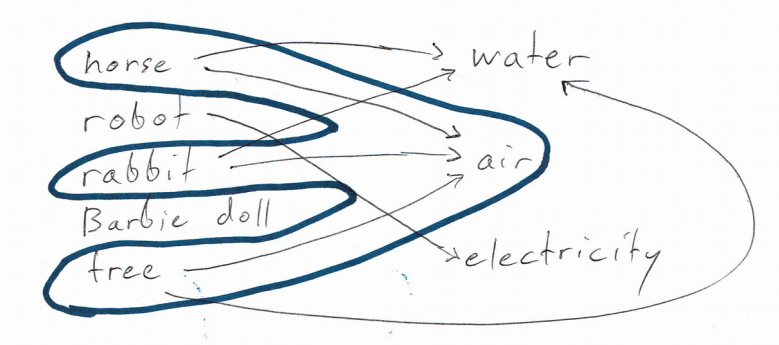

Figure 5: What needs what

I implemented some Star-creating functions. They are the ones with “fork” in the name. Stars are not yet reified as a node variety.

Branches

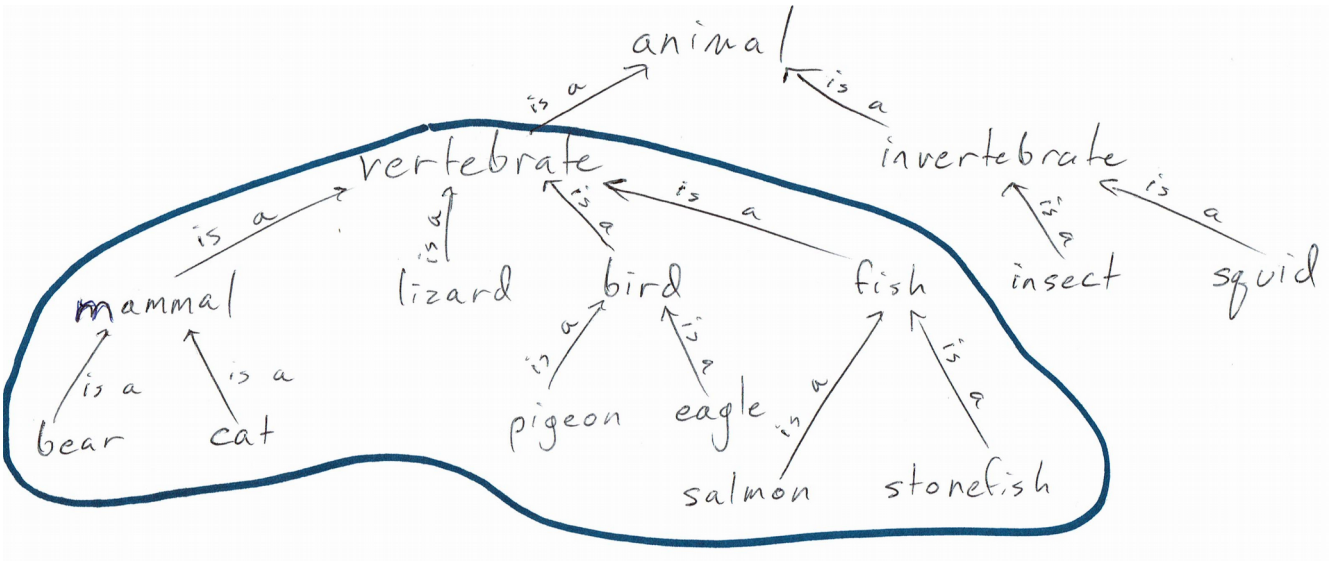

Figure 6: The vertebrates are a branch of the animals

A branch in a graph is everything descended from a given node along a given direction. Figure 6 provides an example: The vertebrates are a branch of the animals. Bears are mammals, which are vertebrates, so “bear” is part of the branch. So is “mammal”, despite not being a leaf. Whereas in a star, everything but the center is a leaf, a branch can include intermediate things that are neither the center nor a leaf.

A branch can be thought of as a set of successive generations of stars. To draw the *vertebrate* branch of *animal*, for example, you would start by drawing a star around *vertebrate*. That first star reaches *mammal*, *lizard*, *bird* and *fish*. (It does not reach *animal*, because the edge from *vertebrate* to *animal* points in the other direction.) Next you might draw the star around *mammal*, which would reach *bear* and *cat*. Etc.

Further examples of branch queries include these:

"Kate the vice-president of engineering, and everybody reporting to Kate, and everybody reporting to someone who reports to Kate …"

"everybody descended from Shakespeare"

"everybody I danced with today, and everybody they (the first generation of leaves) danced with today, and everybody they (the second generation of leaves) danced with today, ..."

I implemented some Branch-creating functions; they are the ones with "dfs" (depth-first search) or "bfs" (breadth-first) in the name. Branches are not yet reified as a variety of node.

A general Branch could use any collection of Directions, in any order – for instance, "show me everybody with a prior conviction for methamphetamine production who spoke to somebody who bought ephidrine in Orlando last year.”

Subexpression Reflection

There is a kind of statement that the RSLT as so far defined cannot encode. Here is an example:

> **Statement 1:** “The volcano ##will fume ####until a cat ##stretches its legs."

The word "its" in that sentence refers to the subexpression "a cat". The pronoun "it" is meaningful to a human reader, but a computer would not know what it referred to. "its" cannot equivalently be replaced by another instance of "a cat", because in that case they might refer to different cats.

The RSLT, in order to represent such statements, needs another kind of expression: the Subexpression Reference, or SubexRef. In the graph implementation of a RSLT, a SubexRef is a fourth kind of node. An edge labeled SubexRefRefersTo points from the Subexpression Reference toward the subexpression it references. Another edge, labeled SubexRefClarifies, points toward the superexpression it clarifies. The SubexRef node itself indicates the position in the superexpression that the subexpression occupies.

For instance, in Statement 1, the Subexpression Reference at "its" would emit an edge toward the expression "a cat" and another toward Statement 1. The Subexpression Reference node would indicate that the subexpression can be found by taking a right at #until and then a left at #stretches.

I have not yet implemented subexpression reflection.

Types

Types makes languages safer. One types a relationship by declaring what kinds of things can be assigned to its blanks.

Untyped, a Template such as "_ is a _" can be misused. A user could encode the statement "Georgia is a pink". The word "pink" is not the kind of word that makes sense in the second position of an #(is a) relationship.

Typing lets you specify that the first Member should be a noun and the second a category.

RSLT Templates could be typed by including a Branch at each blank.

Incomplete subexpressions

In the Demonstration section, we encoded the statement "every person with coronary artery disease needs monitoring". Before we could, we had to first encode the subexpression "every person with coronary artery disease" – and before that, we had to encode "person with coronary artery disease".

I described how the data entry task could be shortened, by using parentheticals or nested #-notation. Rather than create those three subexpressions, one could simply enter "##every person #with coronary artery disease ###needs monitoring", and the subexpressions would be created automatically, or found and used if they already exist.

But that still leaves the problem that the incomplete subexpressions are not worth human attention. Regarding Data Set 1, if I asked to see all the encoded facts about the concept of person, I would want to see these:

> Mildred is a person.
> Every person with coronary artery disease needs monitoring.

I would not, however, be interested in these:

> person with coronary artery disease
> every person with coronary artery disease

That is because they are not meaningful as standalone expressions; they only exist to serve as subexpressions in something else. If the RSLT knew that about them, it could refrain from reporting them as if they were complete thoughts in themselves.

A simple solution would be to divide the Relationship constructor[17] into two, one of them still called Relationship, the other SubRelationship. (The Template and Word constructors would remain unchanged.)

External, spacetime-optimized data structures can be hooked in the RSLT interface without sacrificing its optimizations

The RSLT has good space properties, because it eliminates the need for redundant data. It also has good speed properties, for the same reason trees and graphs do. What it is optimized for, however, is expressivity.

There are other data structures optimized for a particular kind of problem, which can outperform the RSLT on a speed or space basis. Using a RSLT, however, does not mean sacrificing those optimizations!

Example: Hooking a table into a RSLT

A table (“array”) is a very space-efficient data structure. Imagine a table that reports how many kites were produced by which country in which year. It describes ten countries and ten years, so there are 100 numbers in the table. As an array, those 100 numbers are stored in sequence, packed as tightly as possible, with no other information between them.

If they were to be stored in the RSLT, each number would be a Member of an expression of this form:

> **Form A:** "[country] produced [number] kites in [year]."

That would use more space.

17 Here I am borrowing the term “constructor” from Haskell. In this context it just means “kind of node”.

However, the table can be "hooked" into the RSLT "interface" without changing how the table is stored. Doing so lets you query the data as if you had encoded 100 facts of Form A in the normal RSLT manner.

Such a hook must provide the following information: "There is an external data set which encodes, for some years and some countries, how many kites were produced. The countries it covers are these ... The years it covers are these ... If someone asks about one of the available (country, year) pairs, here is how to find the corresponding number of kites."

Appendix A: The RSLT is equivalent to the reflective triplestore

Any *n*-ary relationship can be represented as a tree of binary relationships. “_ did _ to _”, for instance, could be represented as “_ did (_ to _)”. The minimal[18] RSLT is therefore equivalent to a reflective triplestore. To my astonishment, I have found no existing implementation or even mention of reflection in a triplestore. Existing triplestore libraries are flat – a triple’s members can be strings, or numbers, etc., but they cannot themselves be other triples.

Appendix B: Three simpler problems

Words and Phrases

Rather than representing a multi-word phrase like “Father’s Day” as a single Word, it would be more natural to introduce the concept of a Phrase as a string of individual Words. Words in a phrase would be connected by the ternary “[word] precedes [word] in phrase [phrase]” relationship, which is associative in its first two members. For display purposes, everything but the first two members would be suppressed, including the # marks.

The “precedes in phrase” relationship is associative. For instance, the phrase “mother in law” could equally well be constructed by appending in to mother and then appending law, or by appending law to in and then appending that to mother.

Phrases in Templates

As currently specified, a Template contains a list of strings to present as labels between the blanks. For instance, in “_ gave _ to _”, the first visible[19] label is “gave” and the second is “to”. It would be better if those labels were themselves arbitrary RSLT expressions.

18 “Minimal” meaning with Words, Templates and Relationships, but no Directions, Stars, Branches, or SubexRefs.

19 In point of fact, there are four labels in a ternary relationship, but in this one only the second and third are visible. The first and fourth are the empty string, situated before the first and after the last blank, respectively.

Collections

Consider the expression "I need frogs or I need trees or I need waterfalls." You could save space by writing "I need [frogs or trees or waterfalls]." You could make that even shorter by removing the second "or" (and putting the first one at the front of the list), as in "I need [or: frogs, trees, waterfalls]."

I called this idea Coll, and reified it.