Jeffrey Benjamin Brown
2016 June 15

The Reflective Set of Labeled Tuples: A simple, expressivity-maximal generalization of the graph[1]

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

- 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.

Graphs, which are just collections of things connected in pairs, have proven extremely useful. Facebook represents its users (and other things) in a graph[2]. Google represents websites (and other things) in a graph[3].

This paper introduces another one, only slightly more complex than a graph. It is the Reflective Set of Labeled Tuples, or RSLT.[4]

The RSLT can express anything, in a uniform, uncomplicated way. Whereas nodes in a graph can only be connected in pairs, a relationship in a RSLT can have any number of members. Whereas the edges in a graph cannot be connected to other edges, the members of a relationship in a RSLT can themselves be relationships.

Natural languages like English or Spanish can express anything, 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[5].

A graph can express anything, too! This paper shows how. Some facts, a graph expresses very naturally. More complicated kinds can be expressed as well, by using a different idiom for translating between real-world data and graph data. This paper will motivate, develop, and refine that idiom. It will then

---

1 Special thanks to Elliot Cameron, who helped me figure this stuff out.
2 https://developers.facebook.com/docs/graph-api
3 https://www.google.com/intl/en_us/insidesearch/features/search/knowledge.html
4 I’ve been pronouncing it "RISS-ult", like “whistled”.
5 See Neo4j for an example of a beautiful language for asking questions about graphs..

show how the interface can be separated from the implementation, so that a user need not know what a graph is at all. The interface that results is the RSLT..

Table of Contents

Graphs

Undirected graphs

Undirected graphs are commonly drawn as a set of dots with lines between them. The dots are called nodes, and (for reasons I cannot explain) the lines are called edges.

Graphs can also be described symbolically, without using pictures. Text Box 1 describes the same undirected graph as [[Figure 1, without using a picture. Computers represent graphs symbolically.

When computer scientists use graphs, they attach information called "node labels" and "edge labels". Figure 1 is an example, a graph where the nodes are labeled with peoples' names, and the edges are labeled "is married to" or "swims with regularly".

Multiple edges can connect the same two nodes in a graph. For instance, Alice and Bob are married and swim regularly.

Directed graphs

Marriage and swimming together are both examples of "undirected" or "symmetric" relationships. If you're married to someone, they are married to you. "Directed" relationships are not like that. If I disrupt your television, you might not disrupt mine. If I am colder than you, you are certainly not colder than me.

Directed graphs are like undirected ones, except the edges have arrowheads to indicate their direction.

[[Figure 2 is a (directed) graph. The nodes in it are labeled with the names of people., and the edges are labeled one of two things:

_ turned _ into a newt
_ gave a shrubbery to _

(Those "_" symbols are pronounced "blank".) Each label provides a template for a relationship: "_ turned _ into a newt", for instance, provides the template for the relationship "Aubrey turned Jim into a newt".

Directed relationships are hard to represent with an undirected graph. Undirected relationships, however, are easy to represent in a directed graph: just use an arrow and ignore its direction. When computer scientists talk about graphs, they generally mean directed graph. I will adhere to that convention.

Expressing more with a graph

The arity of a relationship

All the relationships we have seen before are "binary" ones. That is, they have exactly two members. Other relationship are possible. For instance, *"_ said _ to _"* relates three things. It is a "ternary" relationship.

“Arity” is the word[6] for the number of members in a relationship. Binary relationships have (or are) arity 2; ternary ones have arity 3. Any positive arity is possible; we could think of a 5-member relationship if we wanted to.

Arity 1 or "unary" relationships are also valid. Some important unary relationships include *"maybe _"*, *"not _"*, *"for every _"* and *"there exists _"*.

6 The word comes from the *ar* in the words *binary, ternary,* etc.

How to express relationships of any arity in a graph: The relationship-as-labeled-node idiom

Suppose we had to store this data in a graph:

> **Data Set 1**
> Bill likes the cheese.
> Ted likes the chair.
> Bill did spin the chair for fun.
> Ted did dance the air guitar for profit.

The first two statements are both *“_ likes _”* relationships. The last two could be parsed a few ways, but the simplest is as a a pair of arity-4 *"_ did _ the _ for _"* relationships.

A graph only has nodes and edges, and edges only connect pairs of nodes. How can we represent a relationship with four members, when every edge connects only two nodes?

We have to change how we read the graph.

In this new idiom (see Figure 3), the relationships in Data Set 1 are not represented using edges, but instead using "relationship nodes". The label on a relationship node indicates the template for the relationship: either *"_ likes _"* or *"_ did _ the _ for _"*.

The nodes in the graph that do not represent relationships, we will call "word nodes". The word nodes include *Bill*, *Ted*, *cheese*, *air guitar*, *chair*, *fun*, and *profit*. (Notice that while word nodes usually represent single words, they can represent multi-word phrases like "air guitar" too. The appendix describes an implementation that recognizes the composition relationship between words and phrases.)

Each blank in the node label of a relationship node is assigned a member using an edge label. Each arity-2 relationship must emit an edge labeled *"member 1"* to its first member, and another edge labeled *"member 2"* to its second member. Similarly, an arity-4 relationship must emit 4 edges.

Higher-order relationships

The relationships we have seen so far are all first-order, in the sense that they are not nested. In none of those relationship was a member another relationship. But consider Statement 4:

> **Statement 4**
> Intel needs silicon because Intel makes computer chips.

Statement 4 is a *"_ because _"* relationship, connecting a *"_ needs _"* to a *"_ makes _"* relationship. Its members are both first-order relationships, so it is a second-order relationship. It can be usefully thought of as comprising two “levels”, with the *because* relationship at the top level and the other two below.

Intuitively, the order of a relationship is the number of levels in it. Formally, an order-*n* relationship is one in which its members all have order *n-1* or less and at least one of them has exactly order *n-1*. *n* corresponds to the number of levels in the statement.

Can we represent higher-order relationships? With the relationship-as-labeled-node idiom, we can. [[Figure 4 encodes Statement 4 in a graph.

Template redundancy: A drawback

Consider the following pair of binary relationships:

> **Data Set 5**
> Donald has ten dollars.
> Donald has a boat.

We can use the relationship-as-labeled-node idiom to represent Data Set 5 with a graph. Figure 5 does that.

Notice that the two relationship nodes in Figure 5 carry redundant information. We have written the relationship template *"_ has _"* on both of them. Among data scientists, this is a red flag. To modify the template, you would have to modify every instance of it. To attach data to the template[7], you would have to attach that data to every instance. Redundant data wastes space and time.

Abstracting the common template away: The relationship-as-unlabeled-node idiom

A slight modification, the relationship-as-unlabeled-node idiom, solves the template redundancy problem. In this new idiom, a relationship node contains no information at all. It is meaningful only because of the edges it emits.

In addition to the *k* edges that go to its *k* members, a relationship node emits one more edge, labeled *template*, toward a new kind of node, the “template node”. A template node’s label provides the template for a relationship: *"_ likes _"*, *"_ said _ to _"*, etc.

Figure 5.1 encodes the information in Data Set 5 using the relationship-as-unlabeled-node idiom. Rather than duplicating the label, each relationship node emits an edge to the same relationship template node. This way, someone can to the template attach data about it – a disambiguation, a warning, the date of creation, the author's name, etc. – much more quickly. Specifically, if *n* is the number of relationships using the template, whereas it would take *O(n)* operations to attach data using the relationship-as-labeled-node idiom, it takes only *O(1)* using the relationship-as-unlabeled-node idiom.

The Reflective Set of Labeled Tuples, or RSLT

The definition

We’ve already got it! The relationship-as-unlabeled-node idiom implements, using a graph, the RSLT interface.

To recap, a RSLT (or, equivalently, a graph using the relationship-as-labeled-node idiom) contains three kinds of node: Words, Templates and Relationships. The labels on Word and Template nodes contain

7 [[If you find yourself asking why someone would attach data to a relationship template, read on!

text; Words look like *"Alice"* or *"batmobile"* and Templates look like *"_ has _"* or *"_ uses _ to mean _"*.

Relationships are encoded using an unlabeled node and some edges emitted from it. The *k+1* edges from an arity-*k* Relationship node attach two kinds of information: its Template and its *k* members. An edge to a Template carries the label *template*, and an edge to the *i*-th member carries the label *member i*. Members of a Relationship can be Words or other Relationships.

The RSLT graph types, as so far described, 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
```

The first line imports the Functional Graph Library. The second says the RSLT is a graph in which the nodes are Expressions and the edges are RSLTEdges. The third says that an Expression is either a Word (in which case it contains one string), a Relationship (in which case it contains no information), or a Template (in which case it contains the strings that go between the blanks in the Template – for instance, the Template *"_ gave _ to _"* would contain the strings "gave" and "to). The last line says that a RSLTEdge comes in two varieties: UsesTemplate (which contains no additional information) and UsesMember (which includes an integer indicating which position the member occupies.)

Notice, however, that the type declarations do not encode all the information about what a RSLT can be. Specifically, they omit the rules about the number and kind of edges that a Relationship must emit.

How the RSLT gets its name

A *k*-tuple is an ordered set of *k* things. For instance, *(cat, mouse)* is a 2-tuple. A labeled *k*-tuple is just a tuple with a label. *("_ glares at _", (cat, mouse))* is a labeled 2-tuple, encoding the idea that the cat glares at the mouse.

The Relationships in a RSLT can be thought of as labeled tuples: the members form a tuple, and the Template provides a label. Since most of the nodes in a RSLT graph are Relationships, it seems fair to call the RSLT a set of labeled tuples.[8]

(At this point we have seen three different senses of the word "label". A node label attaches information to a node, an edge label attaches information to an edge, and a Template serves as the label for a Relationship. That potential confusion is why the term Template seemed better than Label.)

For a computer program, "reflection" is the capacity to view or modify itself. For a data structure, "reflection" is the capacity to refer to itself. Since one Relationship in a RSLT can be a member of another, the RSLT is reflexive.

8 This is particularly true given that Words could be implemented as arity-0 Relationships. (Thanks to Devon Stewart for that observation.) Reflective Set of Labels and Labeled Tuples would be more accurate, but that name sounds complicated, and might impede adoption.

For a user, it's even easier!
Separating interface from implementation.

The Words, Templates and Relationships in the RSLT canbe implemented using the nodes and edges in a graph. A RSLT user, however, does not need to know about the graph!

It is only in this sense in which the RSLT is a new data structure. In principle, the RSLT offers nothing a graph does not offer, because it is a graph. And yet, to a user who only deals with the interface, it is a generalization of the graph. The nodes in a graph can only be connected in pairs, but a relationship in a RSLT can have any number of members. The edges in a graph cannot be connected to other edges, but the members of a relationship in a RSLT can themselves be relationships.

From ordinary English to RSLT: A short trip

All that is required to input a natural language sentence into a RSLT is to make explicit how it divides as a sentence tree into nested relationships. There exist very good automatic sentence parsers already[9]. Someday I expect to be able to feed natural language data into a RSLT with no human oversight. Nonetheless it is worth examining the transformation required.

Consider Data Set 7:

Data Set 7

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

Upon hearing the statements in Data Set 7, an English speaker is able – somehow, with no conscious effort – to parse out the tree illustrated by [[Figure 7.1:

The human listener knows, for instance, that the fourth statement is not about every person, just every person with coronary artery disease. (Recall that *every* is a unary operator.)

It would be hard to feed a picture like Figure 7.1 into the computer; we need a textual representation. Data Set 7.1 provides one possibility. It uses "nested parentheticals" to make the hierarchy of relationships explicit:

Data Set 7.1

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

Feeding statements like those in Data Set 7.1 into a RSLT is straightforward and automable. (For how, see the appendix titled "Building expressions from the bottom up".)

9 One compelling example can be tested in your browser at http://www.link.cs.cmu.edu/link/submit-sentence-4.html.

Heavily nested parentheticals are hard to read. “Nested # notation” provides an equivalent text-based representation to make sentence structure explicit, with a much lower punctuation burden:

Data Set 7.2

Mildred Funnyweather #has stable angina.
Mildred Funnyweather #(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. The varying number of # marks is used to indicate the precedence of Relationship labels.

For instance, in the last sentence, *“##Every person #with coronary artery disease ###needs monitoring,”* a computer can see without even understanding English that the top relationship is *needs*, because *needs* carries the most # marks. So something *needs* something. What is needed? The right-hand member of the *needs* relationship is the single Word *monitoring*. What is doing the needing? The left-hand member of the *needs* relationship is *“##Every person #with coronary artery disease.”* Again, without knowing English, the computer can see that *every* is the top relationship in that expression, because it carries the most # marks. *Every* is a unary expression, so it only has a right-hand member, *“person #with coronary artery disease”*. Etc.

Nested # notation expresses higher-arity Relationships equally well[10].

An invitation to coders

The RSLT as described so far, I have implemented, in Haskell[11], using the Functional Graph Library[12]. It is open-sourced on Github[13], because I want it to develop as quickly as possible.

I need your help! Most of the rest of the ideas in this paper have not been implemented thoroughly.

Complexities and extensions

Types

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

Untyped, a Template such as *"_ is a _"* can be misused. A user could, for instance, encode the statement "Georgia is a pink", when *pink* is not the kind of word that makes sense in the second position of an *#(is a)* relationship. Typing would let you prevent such misuse, by specifying that the first Member should be a noun and the second a category.

10 For example, “Che ###used markers #and paper ##from China ###to write #about China.”
11 https://www.haskell.org/
12 https://hackage.haskell.org/package/fgl-5.5.2.3/docs/Data-Graph-Inductive-Graph.html
13 https://github.com/JeffreyBenjaminBrown/digraphs-with-text

The next section describes how to represent Branches. Once we can do that, typing is as easy as assigning a Branch to every blank in a Template. For an arity-*k* Template, this could be done by drawing k edges from the Template to *k* Branches and labeling each edge something like *type i*, where *i* is the position in the Template of the blank that the edge describes.

Figure 6 illustrates; it shows how the Template *"_ believes _"* could be typed to *"[person] believes [belief]"*.

Subgraph Reflection: Directions, Stars and Branches

Branches are collections of stars. Stars are defined using directions.

Directions

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 its 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[14].

Once you can give a name like Up to a direction, you can start at a node and say, *"show me (this node and) all the nodes that are Up from here"*. To do that is to request a Star.

Stars

A Star in a RSLT consists of some central expression and all the expressions related to it in a particular way.

Figure 5 depicts the graph *"What Needs What"*. 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.

Further examples of star queries include *"(show me) every town with a power plant," "everything Harriet Tubman said about cooperation," or "every rejection letter that demonstrates kindness"*.

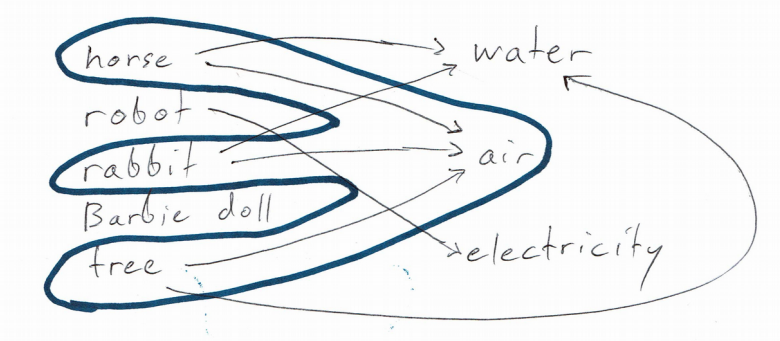


Figure 1: What needs what

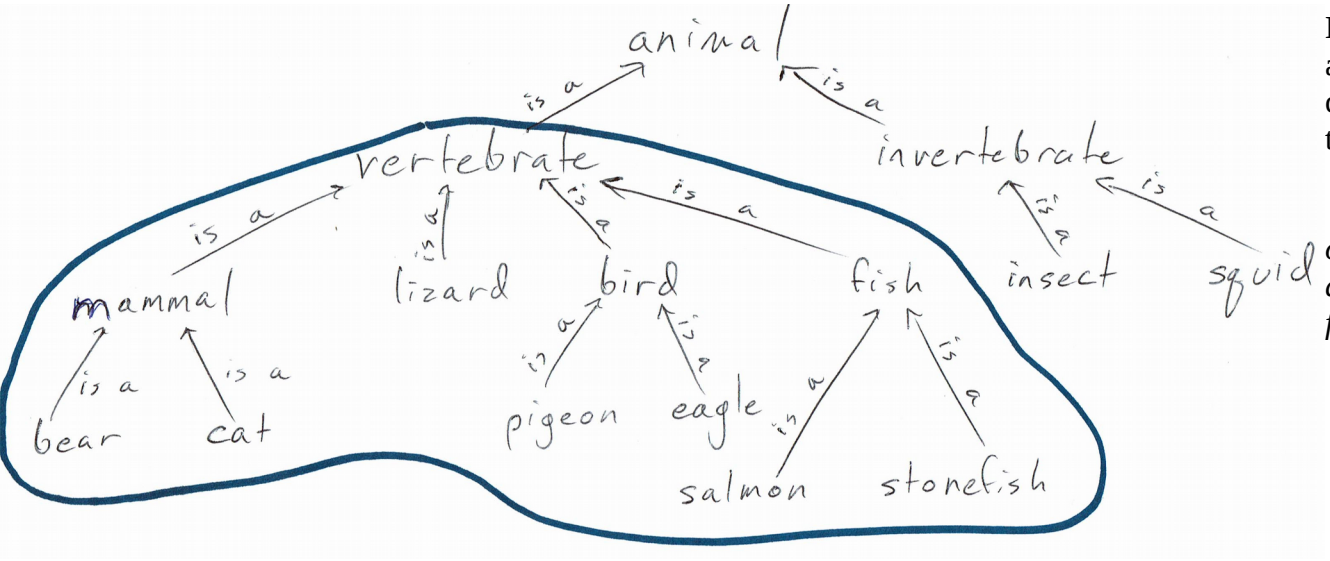
14 You could call them True and False, too, or Blue and Seven.

Branches

A branch in a graph is everything descended from a central node in a certain way, the simplest way being along a single Direction. The central node is called the “root” of the branch.

Figure 6 provides an example: The vertebrates are a branch of the animals. Mammals are vertebrates, so they are in the vertebrates branch. Bears are mammals, so they are in the vertebrates branch, despite not being directly connected to vertebrates.

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.



Branches let a user make queries like these:

"everybody descended from Shakespeare"

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

"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, ..."

Figure 2: The vertebrates are a branch of the animals

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.”*

The branch could be another type of RSLT node. A representation that included some control on the number of generations explored could represent stars as a special case, in which the number of generations is one.

Subexpression Reflection

There is a kind of statement that the RSLT as so far defined cannot encode. Here is an example, written presented in ordinary English, in the nested parentheticals idiom, and in the nested # notation idiom:

> **Statement 8**
> **As ordinary English:** The volcano will fume until a cat stretches its legs.
> **As nested parentheticals:** ((The volcano) will (fume)) until ((a cat) stretches (its legs)).
> **As nested # notation:** The volcano ##will fume ####until a cat ##stretches its legs.

The word *its* in Statement 8 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. *it* cannot equivalently be replaced by another instance of *a cat*, because in that case they might refer to different cats. (It is hard to imagine how a cat would stretch another's legs, given their lack of thumbs, but the computer doesn't know that.)

The RSLT, in order to represent such statements, needs another kind of expression, which we will call the Subexpression Reference. In the graph implementation of a RSLT, a Subexpression Reference is a fourth kind of node. An edge labeled *SubexRefRefs*, points from the Subexpression Reference toward the subexpression it references. Another edge, labeled *SubexRefClarifies*, points toward the superexpression it clarifies. The Subexpression Reference node itself provides instructions for how to traverse from the top of the superexpression to the relevant subexpression.

To finish the example of Statement 8, the Subexpression Reference at *its* would emit an edge toward the expression *a cat* and another toward Statement 8[15]. The Subexpression Reference node would indicate that the subexpression can be found by taking a right at #until and then a left at #stretches.

Relationship algebra

Some relationships adhere to certain transformation rules. For instance, "_ *supports* _" is a "transitive" relationship: If *a supports b* and *b supports c*, then *a supports c*. As another example, "_ *resembles* _" is a "reflexive" relationship: If *a resembles b*, then *b resembles a*.

When transformation rules are modeled in software, automatic reasoning becomes possible. Prolog is an elegant and powerful example.

A RSLT could represent transformation rules as another kind of node in the graph. I don't know where to begin doing that, but it would clearly be good.

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

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

---

15 Note that this violates the usual rule that edges point exclusively from superexpressions to subexpressions.

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 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.

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

Demonstration: Building expressions in a RSLT from the bottom up

In principle, all that is needed to enter Data Set 7 is to type the four sentences in Data Sets 7.1 or 7.2. 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
```

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

16 https://github.com/JeffreyBenjaminBrown/digraphs-with-text/blob/master/introduction/demo.hs

```
> 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 Funnyweather ", "stable angina", "coronary artery disease", and "monitoring". (With respect to English, three of those are mult-word phrases, but with respect to the RSLT they are all 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.

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[17] RSLT is therefore equivalent to a reflective triplestore.

17 "Minimal" meaning with Words, Templates and Relationships, but no Directions, Stars, Branches, or SubexRefs.

However, existing triplestore libraries are flat – a triple's members can be strings, or numbers, etc., but they cannot themselves be other triples. To my astonishment, I have found no existing implementation, nor even mention, of reflection in a triplestore.

Unifying Words and Phrases

Rather than representing a multi-word phrase like "Father's Day" as a single word, it would be more natural to model the inclusion relationship between words and phrases that use the Words. Words in a phrase would be connected by the ternary *"[word] precedes [word] in [phrase]"* relationship. For display purposes, everything but the first two members would be suppressed, including the # marks.

"[word] precedes [word] in [phrase]" is associative in its first two members. 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[18] label is *gave* and the second is *to*.

It would be better if those labels were themselves arbitrary RSLT expressions. The result would be a Template node which , like a Relationship node, carries no information in itself, instead relying entirely on its edges for meaning.

18 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.