Jeffrey Benjamin Brown
2016 June 15

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

Introduction

Are you human?

Congratulations! You understand data structures!

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, which you know as tables or grids.
- Tuples, meaning pairs, triples, 4-tuples, etc.
- Sets, which you might know as collections.
- Lists, which are like sets but ordered first to last.
- Trees, which are the same thing as nested lists.
  - To nest a data structure is to put it inside of another data structure.
- Maps, which you know as dictionaries.
- Graphs, which you know as networks.

That's about all of them!

I am introducing another one, only slightly more complex than a graph. It is the Reflexive Set of Labeled Tuples, or RSLT. (I've been pronouncing it "RISS-ult", so that it sounds like “whistled”.)

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 you could possibly want. And it is queriable in the infinitely flexible way of a graph. (See Neo4j for an example of a beautiful, powerful, simple language for graph queries. It can answer the strangest questions – for example, “Show me every city I have lived where someone gave an alligator to a phone company executive.”)

I have implemented the RSLT, open sourced on Github. [[prepare link and downstream reading; add link]]

I used the Functional Graph Library. If you like graphs and functional code, FGL is truly beautiful.

And, *if ye dare*, you can find my mindmaps, with both the basis for this paper and much more information, as a Freeplane document here. Freeplane is a wonderful, simple[1], popular, decade-old Java app for building and exploring trees of text.

Table of Contents

Introduction....1
- Congratulations! You understand data structures!....1

Motivation....2
- A few short promises....2
- Expressivity limits and the metadata problem: A real-world example....3
- Expressivity limits in data structures....4
  - Information has order....4
  - Information can be of any order....4
  - Other data structures max out at a low order....5

The RSLT: Definition and implementation....6
- Words, Templates and Relationships....6
- Implementing the RSLT using a graph....7
- Encoding high-order facts in a RSLT....7
  - The fast way: Nested # notation....7
  - The slow way: Building expressions from the bottom up....8

Complications....10
- Subgraph reflexivity: Directions, Stars and Branches....10
- Subexpression reflexivity....11
- Types....12
- Incomplete subexpressions....12
- Collections....13
- Hooking other, spacetime-optimized data structures into the RSLT....13
  - Example: Hooking a table into a RSLT....13

Motivation

A few short promises

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, which is what mathematicians call a collection of dots connected with lines. You've probably drawn graphs yourself, where the dots were phrases like "to do" or "go to store" or "reasons to get married".

1 Seriously. To traverse a file in Freeplane requires only the arrow keys and the spacebar.

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

The RSLT eliminates the distinction between data and metadata. Data is data -- be it numbers, implications, instructions, warnings, authorship, time of creation, time of intended completion, requirements, subgoals, arguments, relationships between arguments, privacy instructions for the data itself ... I could go on. 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 how information in the words we use branch into sentence trees. This is because the One Natural Encoding of a fact is exactly the sentence tree diagram for that fact.

At this point you might suspect I am over-promising somewhat. 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. How does the RSLT solve that? Simple – by separating the order from the meaning! Those two kinds of information might be entangled in ordinary text, but in a RSLT they can be recorded separately.

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

The RSLT is low-cost. (If this paragraph is meaningless to you, don't worry; it's not critical to the ideas in this paper.) To encode your existing data in (behind) 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 your existing non-RSLT data can be retained. I elaborate in the section called “Hooking other, spacetime-optimized data structures into the RSLT.”

Some people find those few short promises sufficient motivation to learn about the RSLT. If you are like that, you can skip the rest of this motivation section.

Expressivity limits and the metadata problem: A real-world example

I once had a job working with medical data for economic research. We dealt with facts like these:

(1) Patient X spent 30 days in the hospital.
(2) Institution Y provided the data for statement 1.
(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), however, are "metadata" -- data about the table, which do not themselves fit into the table.

Metadata is typically retained in an ad-hoc manner, if at all. If things go well, the metadata will be stored in a good place, and then 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.

The firm, Precision Health Economics, brought 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. Ours was one instance of a widespread, endemic representation problem.

Data scientists agree*:* Metadata is ubiquitous and dangerous.

With a RSLT, no data needs to be meta!

Expressivity limits in data structures

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

(1) Every statement is an order-n relationship, for some n.
(2) There is no natural upper limit on the possible value of n.
(3) Other data structures cannot encode relationships of order greater than 2.

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 relationship is "Bob uses aspirin." It relates the two first-order expressions "Bob" and "aspirin" through the relationship "_ uses _" (pronounced “blank uses blank”).

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".

Every sentence, every statement, every utterable fact, can be parsed as an *n*th-order relationship, for some value of *n*. Every order is valid; there is no natural limit on the potential depth of an expression.

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 a 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 wonderful.

Before (maybe twice) I described a graph as a collection of dots with some lines connecting the dots, and maybe words or other information written on the lines and the dots. That is accurate but nonstandard terminology. I should introduce the standard terminology.

A graph is a set of "nodes", which can be anything, and "edges", which are lines or arrows connecting the nodes. Edges represent relationships between nodes. Edges can (and typically should) have names, like "_ needs _" or "_ saw _". Nodes in a graph represent first-order expressions like "Bob" or "$60,000", and edges in a graph represent second-order expressions like "Bob spent $60,000".

Here are some examples of graphs:

[[Food chain: eats, helps]]
[[People traveling: Will go, went]]

Graphs, for all their splendor, cannot express third-order relationships! The nodes are things (first order expressions), and the edges are relationships (second order expressions). An edge cannot connect two edges, so a third-order relationship is impossible[2].

That expressivity limit is a problem. For instance, imagine a graph with these nodes:

"Greece"
"$400 billion"
"bad credit risk"

and these edges:

(a) "Greece owes $400 billion."
(b) "Greece is a bad credit risk."

It would be natural to want to encode that (b) is a consequence of (a). That would be a relationship between two relationships, that is, an edge between two edges. It cannot be encoded – not naturally, not without changing the interpretation of some of the graph.

And that is to say nothing of the fourth-order statement, "According to Citibank's definition, Greece is a bad credit risk, because Greece owes $400 billion." Or the fifth-order statement, "If Citibank raised

2 "Impossible" is a slight overstatement. You can add third-order expressions to a graph by introducing a kind of node designed to represent second-order expressions. The result, however, is awkward: it forces you to treat some nodes one way and others another. (And that difference in treatment constitutes a kind of metadata that does not itself fit into the graph!) By contrast, the RSLT lets you treat every kind of data, of any order, in exactly the same way.

to 300% of GDP the debt ratio it uses to define a bad credit risk, then Greece would not be a bad credit risk, because Greece owes $400 billion and its GDP is $200 billion."

Most statements are of an order greater than two, because most sentences are more complex than "puppies are fuzzy".

The RSLT is the only data structure that accommodates them all equally.

The RSLT: Definition and implementation

It 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
```

You can find that code, with explanatory comments, here.

Words, Templates and Relationships

The RSLT contains only three kinds of things: Words, Templates and Relationships. I will capitalize those words to distinguish them from their ordinary usage -- that is, a "Relationship" is part of a RSLT, whereas a "relationship" is pure information, apart of any digital context.

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

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

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 Reflexive 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 reflexive.

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, an edge must connect exactly two nodes.)

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

One can implement a RSLT using a graph, an insight I owe to Elliot Cameron.

The nodes in a RSLT graph are expressions. Each expression is either a Word, a Template, or a Relationship.

The graph has 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. That second kind of edge comes with a number indicating which position in the Template the Member belongs to. Every arity-k Relationship emits one edge to its Template (which must also have arity k) and another edge to each of its k Members, where the Membership edges carry integers indicating Member's position in the Relationship.

Edges never point from Words or Templates, only to them.

[[plants are green because plants use chlorophyll: picture, explanation]]

Encoding high-order facts in a RSLT

The fast way: Nested # notation

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

<u>Data Set 1</u>

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

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

<u>Data Set 1.1</u>

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

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[3].

Inputting an expression using nested # notation allows the computer to generate the substatements in it automatically, or find them and use them if they already exist.

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 Set 1.1. I have not yet written that input method. (I have implemented nested # notation, but only so far 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.

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:

```
> 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.

3 For example, “Che ###used markers #and paper ##from China ###to write #about China.”

Next, lets put our first order expressions into the RSLT: the Words "person", "Mildred", "stable angina", "coronary artery disease", and "mustard". [[footnote]](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",  "mustard"]
```

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,"mustard")
```

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.

Let's view the RSLT again:

```
    > 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,"mustard")
    (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 mustard:

```
    > 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 mustard"
```

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,"mustard")
    (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 mustard")
```

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.

Complications

Subgraph reflexivity: Directions, Stars and Branches

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.

Graph theorists talk about stars. A star consists of a central node and a collection of other nodes connected to the central one. The other nodes are called leaves.

Once you can assign Direction to nodes in a RSLT graph, you can ask to see a Star, that is the set of all Expressions bearing some relation to a central Expression. Examples of such Star queries include "(show me) everything that needs electricity" or "every story that demonstrates kindness".

[[pictures]]

A Branch in a graph is everything descended from a given node along a given direction. Here are some examples:

  "everybody descended from Shakespeare",
  "the boss, and everybody reporting to the boss, and everybody reporting to someone who reports to the boss ...",
  "everybody I danced with today, and everybody they danced with today, and everybody ..."

[[pictures]]

Starting from a Star, you can create a Branch by drawing another Star around each leaf in the first one, and then drawing another Star around each of the new leaves, etc. until you've reached all the data you can reach that way.

[[pictures]]

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

I implemented some Star and Branch-creating functions; they are the ones with "fork" or "dfs" or "bfs" in the name. I have not yet reified Stars or Branches.

A general Branch could use any collection of Directions, in any order. An example of such a query would be "show me everybody who gave an apple from someone who threw a blueberry at someone who made a silly face on a boat".

Subexpression reflexivity

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

  "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 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 a fourth kind of object: the Subexpression Reference. In the graph implementation of a RSLT, a Subexpression Reference is a fourth kind of node, an edge from which points to the subexpression being referenced. The Subexpression Reference node indicates the position in the superexpression that the subexpression occupies. (The superexpression itself is already encoded as an edge from the superexpression to the Subexpression Reference.)

For instance, in the vapor-cat example, the Subexpression Reference at "its" would emit an edge toward the expression "a cat", and 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 implemented subexpression reflexivity.

Types

Types makes languages safer. One types a relationship by declaring what kinds of things can go in which position of it.

Untyped, a Template such as "_ is a _" can be misused. A user could encode the statement "Idea 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, I encoded the statement "every person with coronary artery disease needs mustard". Before I could, I had to first encode the subexpression "every person with coronary artery disease" – and before that, I had to encode "person with coronary artery disease".

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

But that still leaves the problem that those incomplete subexpressions are not worth human attention. In the demo, 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 mustard.

I would not, however, be interested in these:

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

They are not meaningful as standalone expressions; they only exist to serve as subexpressions in something else. The graph should know that about them, so that it can refrain from showing them as if they were complete thoughts in themselves.

A simple solution would be to divide the Relationship constructor[4] into two, one of them still called Relationship, the other SubRelationship. (The Template and Word constructors would remain unchanged.)

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 clarifying by putting the first one at the front of the list): "I need [or: frogs, trees, waterfalls]."

I called this idea Coll, and reified it, but have not written many functions for it.

Hooking other, spacetime-optimized data structures into the RSLT interface

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, that 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 contains the following information: "There is an external data set which encodes, for some years and some countries, how many kites were produced in those countries in those years. 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."

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