Inductive representations of RDF Graphs

Jose Emilio Labra Gayo[a,*], Johan Jeuring[b,c], Jose María Álvarez Rodríguez[d]

[a] *University of Oviedo, C/Calvo Sotelo, S/N, 33007, Oviedo, Spain*
[b] *Utrecht University, The Netherlands*
[c] *Open University the Netherlands, The Netherlands*
[d] *South East European Research Center, (SEERC), 24 Proxenou Koromila Street, Thessaloniki, 54622, Greece*

Abstract

RDF forms the basis of the semantic web technology stack. It is based on a directed graph model where nodes and edges are identified by URIs. Occasionally, such graphs contain literals or blank nodes. The existential nature of blank nodes complicates the graph representation.

In this paper we propose a purely functional representation of RDF graphs using a special form of inductive graphs called inductive triple graphs. We employ logical variables to represent blank nodes. This approach can be implemented in any functional programming language such as Haskell and Scala.

Keywords: Functional Programming, RDF, Graph, Haskell, Scala, Inductive graphs

1. Introduction

RDF appears at the basis of the semantic web technologies layer cake as the *lingua franca* for knowledge representation and exchange. RDF is based on a simple graph model where nodes are predominantly resources, identified by URIs, and edges are properties identified by URIs. Although this apparently simple model has some intricacies, such as the use of blank nodes, RDF has been employed in numerous domains and has been part of the successful linked open data movement.

The main strengths of RDF are the use of global URIs to represent nodes and properties and the composable nature of RDF graphs, which makes it possible to automatically integrate RDF datasets generated by different agents.

Most of the current RDF libraries are based on an imperative model, where a graph is represented as an adjacency list with pointers or an incidence matrix. An algorithm traversing a graph usually maintains a state in which visited nodes are collected.

On the other hand, purely functional programming offers several advantages over imperative programming [13]. It is easier to reuse and compose functional programs, to

*Corresponding Author

Email addresses: labra@uniovi.es (Jose Emilio Labra Gayo), j.t.jeuring@uu.nl (Johan Jeuring), jmalvarez@seerc.org (Jose María Álvarez Rodríguez)

test properties of a program or prove that a program is correct, to transform a program, or to construct a program that can be executed on multi-core architectures.

In this paper, we present a purely functional representation of RDF Graphs and we introduce popular combinators such as fold and map for RDF graphs. Our approach is based on Martin Erwig's inductive functional graphs [10], which we have adapted to the intricacies of the RDF model.

The main contributions of this paper are:

- a simplified representation of inductive graphs
- a purely functional representation of RDF graphs

We expect that our approach will show its benefits in contexts where a formal treatment of RDF graphs is important, like RDF validation and manipulation, and to other contexts like concurrency settings where immutable data structures show its benefits.

This paper is structured as follows: Section 2 describes purely functional approaches to graphs. In particular, we present inductive graphs as introduced by Martin Erwig and we propose a new approach that we call triple graphs which are better suited to implement RDF graphs. Section 3 presents the RDF model. Section 4 describes how we can represent the RDF model in a functional programming setting and section 5 describes the Haskell implementation. Finally, Section 6 describes related work and Section 7 presents some conclusions and future work. Some parts of this paper have been presented in [16] where we also describe an implementation using Scala.

2. Inductive Graphs

2.1. General inductive graphs

In this section we review common graph concepts and the inductive definition of graphs introduced by Martin Erwig [10].

A directed graph can be defined as a pair $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is a set of vertices and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is a set of edges. A labeled directed graph is a directed graph in which vertices and edges are labeled. A vertex is a pair (v, l), where v is a node index and l is a label; an edge is a triple (v_1, v_2, l) where v_1 and v_2 are the source and target vertices and l is the label.

Example 2.1. Figure 1 depicts the labeled directed graph with $\mathcal{V} = \{(1, a), (2, b), (3, c)\}$, and $\mathcal{E} = \{(1, 2, p), (2, 1, q), (2, 3, r), (3, 1, s)\}$.

In software, a graph is often represented using imperative data structures describing how nodes are linked by means of edges. Such a data structure may be an adjacency list with pointers, or an incidence matrix. When a graph changes, the corresponding data structure is destructively updated. A graph algorithm that visits nodes one after the other uses an additional data structure to register what part of the graph has been visited, or adapts the graph representation to include additional fields to mark nodes and edges in the graph itself.

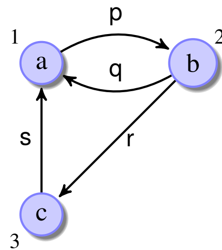

Figure 1: Simple labeled directed graph

Implementing graph algorithms in a functional programming language is challenging as one has to either pass an additional parameter representing the graph to all functions, or use monads to emulate the imperative style. This style complicates correctness proofs and program transformations.

Martin Erwig [9] introduced a functional representation of graphs, where a graph is defined by induction. He describes two implementations for persistent graphs [8], and a Haskell [10] implementation, which we summarize in this section.

A graph is defined inductively as an empty graph, or an extension of a graph with a node v together with its label and a list of successors and predecessors of v in the graph.

The type of the values used in an extension of a graph is given by the type `Context`
.

```
1  -- Context of a node in the graph
2  type Context a b =
3      (Adj b, Node, a, Adj b)
4
5  -- Adjacent labelled nodes
6  type Adj b = [(Node,b)]
7
8  -- Labelled nodes
9  type LNode a = (a,Node)
10
11 -- Index of nodes
12 type Node = Int
13
14 -- Labelled edges
15 type LEdge b = (Node,Node,b)
```

A context of a node is a value `(pred,node,label,succ)` where `pred` is the list of predecessors, `node` is the index of the node, `label` is the label of the node and `succ` is the list of successors. Labelled nodes are represented by a pair consisting of a label and a node, and labelled edges are represented by a source and a target node, together with a label.

Figure 2: Inductive graph representation using M. Erwig approach

```
class Graph gr where
 empty :: gr a b

 isEmpty :: gr a b -> Bool

 match :: Node -> gr a b -> (Context a b, gr a b)

 mkGraph :: [LNode a] -> [LEdge b] -> gr a b

 labNodes :: gr a b -> [LNode a]
```

Example 2.2. The context of node `b` in Figure 1 is:

```
([(1,'p')],2,'b',[(1,'q'),(3,'r')])
```

Although the graph type is implemented as an abstract type for efficiency reasons, it is convenient to think of the graph as an algebraic type with two constructors `Empty` and `:&`.

```
data Graph a b = Empty
             | Context a b :& Graph a b
```

Example 2.3. The graph from Figure 1 can be encoded as:

```
([(2,'q'),(3,'s')],1,'a',[(2,'p')]) :&
([],2,'b',[(3,'r')]) :&
([],3,'c',[]) :&
Empty
```

Notice that the same graph can be encoded in different ways. Another encoding is:

```
([(2,'r')],3,'c',[(1,'s')]) :&
([(1,'p')],2,'b',[(1,'q')]) :&
([],1,'a',[]) :&
Empty
```

The inductive graph approach has been implemented in Haskell in the so-called FGL library[1]. FGL defines a type class `Graph` representing the interface of graphs, together with some common graph operations, see Figure 2. Based on these basic operations we can define operations like *fold*, *map*, etc.

[1] `http://web.engr.oregonstate.edu/~erwig/fgl/haskell`

Example 2.4. The graph from example 2.3 is rerpesented in FGL as:

```
e :: Gr Char Char
e = mkGraph
  [('a',1),('b',2),('c',3)]
  [(1,2,'p')
  ,(2,1,'q')
  ,(2,3,'r')
  ,(3,1,'s')]
```

A problem with this interface is that it exposes the management of node/edge indexes to the user of the library. For example, it is possible to construct graphs with edges between non-existing nodes.

Example 2.5. The following code compiles but produces a runtime error because there is no node with index `42`:

```
gErr :: Gr Char Char
gErr = mkGraph
  [('a',1)]
  [(1,42,'p')]
```

2.2. Inductive triple graphs

In this section, we introduce a simplified representation of inductive graphs based on the following assumptions:

- each node and each edge have a label
- labels are unique
- the label of an edge can also be the label of a node

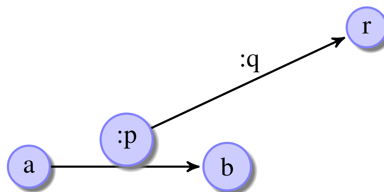

Figure 3: A triple graph with an edge that is also a node

These two assumptions are motivated by the nature of RDF Graphs, which we will explain in the next section.

Our approach is general enough to convert the previous representation to this one and vice versa.

An advantage of this representation is that a user does not have to be aware of node indexes. Furthermore, there is no need for two different types for nodes and edges in this representation, simplifying the development of a graph algebra.

A graph of elements of type `a` is generated by a set of triples where each triple has the type `(a,a,a)`. We will call this type of graphs TGraph (triple based graphs). We can consider triple graphs to be defined by the following datatype (in practice, the internal implementation could be different):

```
data TGraph a = Empty
            | TContext a :& Graph a
```

where `TContext a` is defined as:

```
type TContext a =
  (a, [(a,a)], [(a,a)], [(a,a)])
```

A `TContext` of a node is a value `(node,pred,succ,rels)` where `node` is the node itself, `pred` is the list of predecessors, `succ` is the list of successors, and `rels` is the list of pairs of nodes related by this node in its role as edge.

Example 2.6. The graph from Figure 1 can be defined as:

```
('a',[('c','s'),('b','q')],[('p','b')],[]) :&
('b',[],[('r','c')],[]) :&
('c',[],[],[]) :&
('p',[],[],[]) :&
('q',[],[],[]) :&
('r',[],[],[]) :&
('s',[],[],[]) :&
Empty
```

With this representation it is easy to model graphs in which nodes are used as edges.

Example 2.7. The graph from Figure 3 can be defined by:

```
('a',[],[('p','b')],[]) :&
('b',[],[],[]) :&
('p',[],[('q','r')],[]) :&
('q',[],[],[]) :&
('r',[],[],[]) :&
Empty
```

As in Erwig's approach, it is possible to have different representations for the same graph.

Example 2.8. The previous graph could also be defined as follows when we reverse the nodes:

```
('r',[],[('p','q')],[]) :&
('q',[],[],[]) :&
('p',[],[],[('a','b')]) :&
('b',[],[],[]) :&
('a',[],[],[]) :&
Empty
```

In Haskell `TGraph` is implemented as a type class with at least the following methods:

```
class TGraph gr where
 -- empty graph
 empty :: gr a

 -- decompose a graph
 match :: a -> gr a -> (TContext a, gr a)

 -- make graph from triples
 mkGraph :: [(a,a,a)] -> gr a

 -- nodes of a graph
 nodes :: gr a -> [a]

 -- extend a graph
 extend :: TContext a -> gr a -> gr a
```

Figure 4: TGraph representation

Note that with this simplified interface, it is impossible to construct graphs with edges between non-existing nodes.

2.3. *Algebra of graphs*

We define some common general operations over graphs. One of the most versatile combinators is `foldGraph`:

```
foldTGraph :: TGraph gr =>
          b -> (TContext a -> b -> b) -> gr a -> b
foldTGraph e f g = case nodes g of
 [] -> e
 (n:_) -> let (ctx,g') = match n g
         in f ctx (foldTGraph e f g')
```

These operations satisfy common laws like the fusion law, which says that given:

```
h e = e'
f a b = f' a (h b)
```

then:

```
h . foldTGraph f e = foldTGraph f' e'
```

fold is the basic recursive operator on datatypes: any recursive function on a datatype can be expressed as a *fold*. For example, we can define some common functions in terms of `foldTGraph` like `lengthTGraph` to calculate the number of nodes in a `TGraph` and `sumTGraph` which adds the elements of a `TGraph`

```
lengthTGraph :: TGraph gr => gr a -> Int
lengthTGraph = foldTGraph 0 (\x y -> 1 + y)

sumTGraph :: TGraph gr => gr Int -> Int
sumTGraph = foldTGraph 0(\c r -> node c + r)
```

We can define `mapTGraph` in terms of `foldTGraph`.

```
mapTGraph :: TGraph gr =>
            (TContext a -> TContext b) -> gr a -> gr b
mapTGraph f =
 foldTGraph empty
   (\ctx g ->
      extend (mapCtx f ctx) g)
 where
  mapCtx f (n,pred,succ,rels) =
   (f n,
    mapPairs f pred,
    mapPairs f succ,
    mapPairs f rels)
  mapPairs f = map
    (\(x,y) -> (f x, f y))
```

fold and *map* are very popular combinators that capture the most popular operations on datatypes [18].

An interesting property of `mapTGraph` is that it maintains the graph structure whenever the function f is injective. If f is not injective, the graph structure may be completely modified.

Example 2.9. Applying the function `mapTGraph (_ -> 0)` returns a graph with a single node.

We can use `mapTGraph` to define some other common operations, such as an operation that reverses the order of the edges, over graphs.

Example 2.10. The following function reverses the edges in a graph.

```
rev :: (TGraph gr) => gr a -> gr a
rev = mapTGraph swapCtx
 where
```

```
swapCtx (n,pred,succ,rels) =
  (n,succ,pred,map swap rels)
```

We have implemented several other functions over graphs, such as depth-first search, topological sorting, strongly connected components, etc, the implementations are available at `https://github.com/labra/haws`.

3. The RDF Model

The RDF Model was accepted as a recommendation in 2004 [1]. The 2004 recommendation is being updated to RDF 1.1, and the current version [5] is the one we use for the main graph model in this paper. Resources are globally denoted IRIs (internationalized resource identifiers [7])[2]. Notice that the IRIs in the RDF Model are global identifiers for nodes (subjects or objects of triples) and for edges (predicates). Therefore, an IRI can be both a node and an edge. Qualified names are employed to shorten IRIs. For example, if we replace `http://example.org` by the prefix `ex:`, `ex:a` refers `http://example.org/a`. Throughout the paper we will employ Turtle notation [6]. Turtle supports defining triples by declaring prefix aliases for IRIs and introducing some simplifications.

Example 3.1. The following Turtle code represents the graph in Figure 1.

```
@prefix : <http://example.org/>

:a :p :b .
:b :q :a .
:b :r :c .
:c :s :a .
```

An *RDF triple* is a three-tuple $\langle s, p, o \rangle \in (\mathcal{I} \cup \mathcal{B}) \times \mathcal{I} \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L})$, where $\mathcal{I}$ is a set of IRIs, $\mathcal{B}$ a set of blank nodes, and $\mathcal{L}$ a set of literals. The components s, p, o are called, the subject, the predicate, and the object of the triple, respectively. An *RDF graph* $\mathcal{G}$ is a set of RDF triples.

Example 3.2. The following Turtle code represents the graph in Figure 3.

```
:a :p :b .
:p :q :r .
```

Blank nodes in RDF are used to describe elements whose IRI is not known or does not exist. The Turtle syntax for blank nodes is `_:id` where `id` represents a local identifier for the blank node.

Example 3.3. The following set of triples can be depicted by the graph in Figure 5.

[2] Although the 2004 RDF recommendation employs URIs, the current working draft uses IRIs

```
:a :p _:b1 .
:a :p _:b2 .
_:b1 :q :b .
_:b2 :r :b .
```

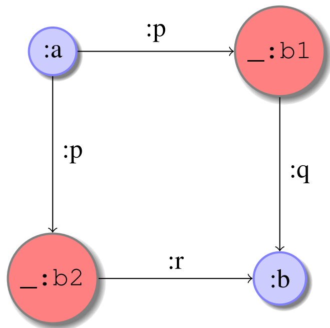

Figure 5: Example with two blank nodes

Blank node identifiers are local to an RDF document and can be described by means of existential variables [17]. Intuitively, a triple $\langle b_1, p, o\rangle$ where $b_1 \in \mathcal{B}$ can be read as $\exists b_1 \langle b_1, p, o\rangle$. This predicate holds if there exists a resource s such that $\langle s, p, o\rangle$ holds.

When interpreting an RDF document with blank nodes, arbitrary resources can be used to replace the blank nodes, replacing the same blank node by the same resource.

Example 3.4. Example 3.3 can be represented by:

$$\exists b_1, b_2 \begin{bmatrix} \langle \text{:a}, \text{:p}, b_1 \rangle \\ \langle \text{:a}, \text{:p}, b_2 \rangle \\ \langle b_1, \text{:q}, \text{:b} \rangle \\ \langle b_2, \text{:r}, \text{:b} \rangle \end{bmatrix}$$

Currently, the RDF model only allows blank nodes to appear as subjects or objects, and not as predicates. This restriction may be removed in future versions of RDF so we do not impose it in our graph representation model. Literals are used to denote values such as strings, numbers, dates, etc. There are two types of literals: datatype literals and language literals. A datatype literal is a pair (val, t) where $\text{val} \in \mathcal{L}$ is a lexical form representing its value and $t \in \mathcal{T}$ is a datatype URI. In Turtle, datatype literals are represented as `val^^t`. A language literal is a pair $(s, lang)$ where $s \in \mathcal{L}$ is a string value and $lang$ is a string that identifies the language of the literal. The values of $lang$ should follow BCP47 [21]. In Turtle, language literals are represented as `s@lang`.

Example 3.5. The following set of triples contains two literals with two different languages and a datatype literal.

```
:a rdfs:label "Hello"@en .
```

```
:a rdfs:label "Hola"@es .
:a :p "1"^^<xsd:integer> .
```

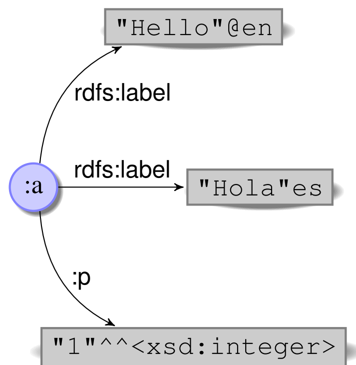

Figure 6: Example with literals

In the RDF data model, literals are constants. Two literals are equal if their lexical form, datatype and language are equal. The different lexical forms of literals can be considered unique values. Although the current RDF graph model restricts literals to appear only as objects, we do not impose that restriction in our model. For simplicity, we only use lexical forms of literals in the rest of the paper.

4. Functional representation of RDF Graphs

The RDF model is a labeled directed graph where the nodes are resources. A resource can be modeled as an algebraic datatype:

```
data Resource = IRI String
              | Literal String
              | BNode BNodeId

type BNodeId = Int
```

The RDF graph model has three special aspects that we need to take into account:

- edges can also be nodes at the same time (subjects or objects)
- nodes are uniquely identified. There are three types of nodes: resource nodes, blank nodes and literals
- the identifier of a blank node is local to the graph, and has no meaning outside the scope of the graph. Consequently, a blank node has an existential nature [17]

To address the first two aspects we employ the triple inductive graphs introduced in Section 2.2, which support defining graphs in which edges can also appear as nodes, and both nodes and edges are uniquely identified. The existential nature of blank nodes can be modeled by logical variables [23].

The type of RDF graphs is defined as:

```
data RDFGraph = Ground (Graph Resource)
              | Exists (BNodeId -> RDFGraph)
```

Example 4.1. The graph from Figure 5 is defined as:

```
Exists (\b1 ->
Exists (\b2 ->
Ground (
  ('a',[],[('p',b1),('p',b2)],[]) :&
  ('b',[(b1,'q'),(b2,'r')],[],[]) :&
  (b1, [], [], []) :&
  (b2, [], [], []) :&
  (p, [], [], []) :&
  (q, [], [], []) :&
  (r, [], [], []) :&
 Empty)))
```

One of the benefits of the `RDFGraph` encoding is that it makes it easy to construct some common functions on RDF graphs.

For example, merging two RDF graphs can easily be accomplished by means of function composition and folds over triple graphs.

```
mergeRDF :: RDFGraph -> RDFGraph -> RDFGraph
mergeRDF g (Exists f) = Exists (\x -> mergeRDF g (f x))
mergeRDF g (Ground g') = foldTGraph g compRDF g'
 where
  compRDF ctx (Exists f) =
       Exists (\x -> compRDF ctx (f x))
  compRDF ctx (Ground g) =
       Ground (comp ctx g)
```

Given their functional representation, blank nodes are automatically handled by the functional language implementation. So the programmer does not have to take care about renaming blank nodes and the system ensures its correct behaviour.

It is possible to define maps over RDFGraphs as:

```
mapRDFGraph::(Resource -> Resource) ->
               RDFGraph -> RDFGraph
mapRDFGraph h (Basic g) =
     Basic (mapTGraph (mapCtx h) g)
mapRDFGraph h (Exists f) =
     Exists (\x -> mapRDFGraph h (f x))
```

Finally, in order to define `foldRDFGraph`, it is necessary to have a seed generator that assigns different values to each blank node. In the following definition, we inject integer numbers starting from 0.

```
foldRDFGraph ::
  a -> (Context Resource -> a -> a) -> RDFGraph -> a
foldRDFGraph e h =
  foldRDFGraph' e h 0
 where
  foldRDFGraph' e h seed (Ground g) =
      foldTGraph e h g
  foldRDFGraph' e h seed (Exists f) =
      foldRDFGraph' e h (seed + 1) (f seed)
```

Notice that the use of integer seeds to generate unique identifiers for existential variables could be encapsulated in a monad which would lead to monadic fusion laws [19].

5. Implementation

We have developed two implementations in Haskell: one using higher-order functions and other based on the FGL library[3].

We have also developed a Scala implementation[4] using the *Graph for Scala* library. This implementation is described in more detail in [16]. The Scala implementation contains a full RDF 1.1 parser which passes all the W3c tests so our approach can handle every RDF construct.

Our first implementation uses a functional representation of graphs. A graph is defined by a set of nodes and a function from nodes to contexts.

```
data FunTGraph a =
 FunTGraph (a -> Maybe (Context a, FunTGraph a))
          (Set a)
```

This implementation was inspired by the functional graph representation that appears in [22]. It offers some theoretical insight, but is not intended to be used for practical proposes.

The second implementation in Haskell is based on the FGL library. Here, a `TGraph a` is represented by a `Graph a` and a map from nodes to the edges that they relate.

```
data FGLTGraph a = FGLTGraph {
  graph :: Graph a a,
  nodeMap :: Map a (ValueGraph a)
}

data ValueGraph a = Value {
```

[3]The Haskell implementations are available at `https://github.com/labra/haws`
[4]The Scala implementation is available at `https://github.com/labra/wesin`

```
7 grNode :: Node,
8 edges :: Set (a,a)
9 }
```

The `nodeMap` keeps track of the index of each node in the graph and the set of (subject,object) nodes that the node relates if it acts as a predicate.

In this way, we see that any inductive triple graph can be converted to an inductive graph using Martin Erwig's approach.

6. Related Work

There area quite a few RDF libraries using imperative languages, such as Jena[5], Sesame[6] (Java), dotNetRDF[7] (C#), Redland[8] (C), RDFLib[9] (Python), RDF.rb[10] (Ruby), etc.

For dynamic languages, most of the RDF libraries are binders to some underlying imperative implementation. For example, banana-RDF[11] is an RDF library implementation in Scala. Although the library emphasizes type safety and immutability, the underlying implementations are Jena and Sesame.

There are some functional implementations of RDF libraries. Most of these employ mutable data structures. For example, scaRDF[12] started as a facade of Jena and evolved to implement the whole RDF graph machinery in Scala, employing mutable adjacency maps.

There have been several attempts to define RDF libraries in Haskell. Swish[13] provides an RDF toolkit with support for RDF inference using a Horn-style rule system. It implements some common tasks like graph merging, isomorphism and partitioning representing an RDf graph as a set of arcs. RDF4h[14] is another complete RDF library which defines a type class `RDF` and an implementation using adjacency maps. We consider that our approach could be added as another implementation.

Martin Erwig introduced the definition of inductive graphs [9]. He gives two possible implementations [8], one using version trees of functional arrays, and the other using balanced binary search trees. Both are implemented in SML. Later, Erwig implemented the second approach in Haskell which has become the FGL library.

Jeffrey and Patel-Schneider employ Agda[15] to check integrity constraints of RDF [14], and propose a programming language for the semantic web [15].

[5] http://jena.apache.org/
[6] http://www.openrdf.org/
[7] http://www.dotnetrdf.org/
[8] http://librdf.org/
[9] http://www.rdflib.net/
[10] http://rdf.rubyforge.org/
[11] https://github.com/w3c/banana-rdf
[12] https://code.google.com/p/scardf/
[13] https://bitbucket.org/doug_burke/swish
[14] http://protempore.net/rdf4h/
[15] https://github.com/agda/agda-web-semantic

Mallea et al [17] describe the existential nature of blank nodes in RDF. Our use of existential variables was inspired by Seres and Spivey [23] and Claessen [3]. The representation is known in logic programming as ‘the completion process of predicates’, first described and used by Clark in 1978 [4] to deal with the semantics of negation in definite programs.

Our representation of existential variables in RDFGraphs uses a datatype with an embedded function. Fegaras and Sheard [11] describe different approaches to implement folds (also known as catamorphisms) over this kind of datatypes, and show as an example how to represent graphs using a recursive datatype with embedded functions.

The representation of RDF graphs using hypergraphs, and transformations between hypergraphs and bipartite graphs, have been studied by Hayes and Gutiérrez [12].

Recently, Oliveira et al. [20] define structured graphs in which sharing and cycles are represented using recursive binders, and an encoding inspired by parametric higher-order abstract syntax [2]. They apply their work to grammar analysis and transformation. It is future work to check if their approach can also be applied to represent RDF graphs.

7. Conclusions

This paper introduces a purely functional representation of RDF graphs. Our approach is based on a variation of inductive graphs, which we dub inductive triple graphs. The main advantage of this approach is that it enables the development of an algebra of RDF graphs with common operations like mapping, folding, or merging.

We have implemented our representation of RDF graphs using the functional programming languages Haskell and Scala.

One of the benefits of an immutable data structure such as our representation for graphs is its potential for concurrent programming. In the future we want to release a complete RDF library, and check its suitability and scalability in some real-world scenarios.

8. Acknowledgments

This work has been partially funded by the Spanish project MICINN-12-TIN2011-27871 ROCAS (Reasoning about the Cloud by Applying Semantics) and by the International Excellence Campus grant of the University of Oviedo which allowed the first author to spend a research visit at Utrecht University.

[1] J. J. Carroll and G. Klyne. Resource description framework (RDF): Concepts and abstract syntax. W3C recommendation, W3C, Feb. 2004. `http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/`.

[2] A. J. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In J. Hook and P. Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 143–156. ACM, 2008.

[3] K. Claessen and P. Ljunglöf. Typed logical variables in haskell. In *Proceedings of Haskell Workshop*, Montreal, Canada, 2000. University of Nottingham, Technical Report.

[4] K. L. Clark. *Logic and Databases*, chapter Negation as failure, pages 293–322. Eds. Plenum Press, 1978.

[5] R. Cyganiak and D. Wood. Resource description framework (RDF): Concepts and abstract syntax. W3C working draft, W3C, Jan. 2013. `http://www.w3.org/TR/rdf11-concepts/`.

[6] E. P. Dave Becket, Tim Berners-Lee and G. Carothers. Turtle, terse rdf triple language. World Wide Web Consortium, Working Draft, WD-Turtle, July 2012.

[7] M. Dürst and M. Suignard. Internationalized resource identifiers. Technical Report 3987, IETF, 2005.

[8] M. Erwig. Fully persistent graphs - which one to choose? In *9th Int. Workshop on Implementation of Functional Languages*, number 1467 in LNCS, pages 123–140. Springer Verlag, 1997.

[9] M. Erwig. Functional programming with graphs. *SIGPLAN Not.*, 32(8):52–65, Aug. 1997.

[10] M. Erwig. Inductive graphs and functional graph algorithms. *J. Funct. Program.*, 11(5):467–492, Sept. 2001.

[11] L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 284–294, New York, NY, USA, 1996. ACM.

[12] J. Hayes and C. Gutiérrez. Bipartite graphs as intermediate model for rdf. In *Third International Semantic Web Conference (ISWC2004)*, volume 3298 of *Lecture Notes in Computer Science*, pages 47 – 61. Springer-Verlag, 2004.

[13] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[14] A. S. A. Jeffrey and P. F. Patel-Schneider. Integrity constraints for linked data. In *Proc. Int. Workshop Description Logics*, 2011.

[15] A. S. A. Jeffrey and P. F. Patel-Schneider. As xduce is to xml so ? is to rdf: Programming languages for the semantic web. In *Proc. Off The Beaten Track: Workshop on Underrepresented Problems for Programming Language Researchers*, 2012.

[16] J. E. Labra-Gayo, J. Jeuring, and J. M. Álvarez Rodríguez. Inductive triple graphs: A purely functional approach to represent RDF. In M. Croitoru, editor, *3rd International Workshop on Graph Structures for Knowledge Representation and Reasoning*, Beijing, China, August 2013. LNAI Series, Springer-Verlag.

[17] A. Mallea, M. Arenas, A. Hogan, and A. Polleres. On blank nodes. In L. Aroyo, C. Welty, H. Alani, J. Taylor, A. Bernstein, L. Kagal, N. F. Noy, and E. Blomqvist, editors, *International Semantic Web Conference (1)*, volume 7031 of *Lecture Notes in Computer Science*, pages 421–437. Springer, 2011.

[18] E. Meijer, M. Fokkinga, R. Paterson, and J. Hughes. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. *FPCA 1991: Proceedings 5th ACM Conference on Functional Programming Languages and Computer Architecture*, 523:124–144, 1991.

[19] E. Meijer and J. Jeuring. Merging monads and folds for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 228–266. Springer, 1995.

[20] B. C. Oliveira and W. R. Cook. Functional programming with structured graphs. *SIGPLAN Not.*, 47(9):77–88, Sept. 2012.

[21] A. Phillips and M. Davis. Tags for Identifying Languages. Technical Report 47, Internet Engineering Task Force, September 2009.

[22] C. Reade. *Elements of Functional Programming*. International Computer Science. Addison-Wesley, 1989.

[23] S. Seres and J. M. Spivey. Embedding Prolog into Haskell. In *Proceedings of HASKELL'99*. Department of Computer Science, University of Utrecht, 1999.