Inductive Triple Graphs: A purely functional approach to represent RDF

Jose Emilio Labra Gayo

Johan Jeuring

Jose María Álvarez Rodríguez

Technical Report UU-CS-2013-009

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Inductive Triple Graphs: A purely functional approach to represent RDF

Jose Emilio Labra Gayo
University of Oviedo
Spain
labra@uniovi.es

Johan Jeuring
Utrecht University
Open University of the Netherlands
The Netherlands
j.t.jeuring@uu.nl

Jose María Álvarez Rodríguez
South East European Research Center
Greece
jmalvarez@seerc.org

Abstract

RDF is one of the cornerstones of the Semantic Web. It can be considered as a knowledge representation common language based on a graph model. In the functional programming community, inductive graphs have been proposed as a purely functional representation of graphs, which makes reasoning and concurrent programming simpler. In this paper, we propose a simplified representation of inductive graphs, called Inductive Triple Graphs, which can be used to represent RDF in a purely functional way. We show how to encode blank nodes using existential variables, and we describe two implementations of our approach in Haskell and Scala.

1 Introduction

RDF appears at the basis of the semantic web technologies stack as the common language for knowledge representation and exchange. It 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 implementations of 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.

Purely functional programming offers several advantages over imperative programming [13]. It is easier to reuse and compose functional programs, to 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. 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
- a description of Haskell and Scala implementations of an RDF library

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 called triple graphs, which is 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. Section 5 describes two implementations of our approach: one in Haskell and another in Scala. Section 6 describes related work and Section 7 concludes and describes future work.

2 Inductive Graphs

2.1 General inductive graphs

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

A directed graph is 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 an imperative data structure 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.

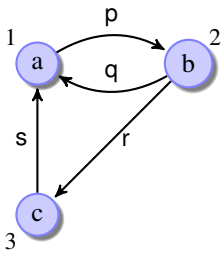

Figure 1: Simple labeled directed graph

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

Martin Erwig [9] introduces a functional representation of graphs where a graph is defined by induction. He describes two implementations that enable persistent graphs [8], and an implementation in Haskell [10], which we summarize in this section. He defines a graph as either 1) an empty graph or 2) an extension of a graph with a node v together with its label and a list of v's succesors and predecessors that are already in the graph.

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

```
-- Context of a node in the graph
type Context a b =
    (Adj b, Node, a, Adj b)

-- Adjacent labelled nodes
type Adj b = [(Node,b)]

-- Labelled nodes
type LNode a = (a,Node)

-- Index of nodes
type Node = Int

-- Labelled edges
type LEdge b = (Node,Node,b)
```

A context of a node is a value `(pred,node,lab,succ)`, where `pred` is the list of predecessors, `node` is the index of the node, `lab` 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.

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 type 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
```

Note that there may be different inductive representations for the same graph.

Example 2.4. Here is another representation of the graph in Figure 1:

```
([(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 FGL library[1]. FGL defines a type class `Graph` to represent the interface of graphs and some common operations. The essential operations are:

```
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]
```

Figure 2: Inductive graph representation using M. Erwig approach

A problem with this interface is that it exposes the management of node/edge indexes to the user of the library. It is for example 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

We propose a simplified representation of inductive graphs based on three 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

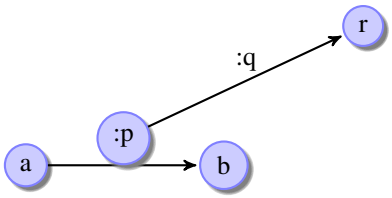

Figure 3: A triple graph with an edge acting also as a node

These three assumptions are motivated by the nature of RDF Graphs, which we will explain in the next section. As we will see in Section 2.3, our approach is general enough to represent any graph.

One advantage of this representation is that a user does not have to be aware of node indexes. Also, there is no need for two different types for nodes/edges simplifying the development of an algebra of graphs.

A graph of elements of type `a` is described by a set of triples where each triple has the type `(a,a,a)`. We will call these kind of graphs `TGraph` (triple based graphs).

We assume triple graphs are defined by the following datatype. Practical implementations may use a different representation.

```
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 when it is an 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 edges are also nodes.

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

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

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

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

Example 2.8. The previous graph can also be defined as follows, where we reverse the order of the nodes:

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

In Haskell, we implement `TGraph` as a type class with the following essential operations:

```
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 (similar to :&)
 extend :: TContext a -> gr a -> gr a
```

Figure 4: TGraph representation

Using this simplified interface, it is impossible to create graphs with edges between non-existing nodes.

2.3 Representing Graphs at triple Graphs

We can represent general inductive graphs [10] using inductive triple graphs. The main difference between general inductive graphs and inductive triple graphs is that in general inductive graphs, labels of nodes and edges have an index (an `Int`), which does not need to be different. We represent a general inductive graph using a record with a triple graph that stores either the index of the node or the index of the edge, and two maps, one from indexes to node labels and another from indexes to edge labels.

```
data GValue a b = Node a | Edge b

data Graph a b = Graph {
 graph :: TGraph (GValue Int Int),
 nodes :: Map Int a
 edges :: Map Int b
```

Example 2.9. The graph from example 2.4 can be represented as:

```
Graph {
 graph =
   (Node 1,[(Node 3,Edge 4),
            (Node 2,Edge 2)],
           [(Edge 1,Node 2)],
           [] :&
   (Node 2,[],
           [(Edge 3,Node 3)],
           []) :&
   (Node 3,[],[],[]) :&
   (Edge 1,[],[],[]) :&
   (Edge 2,[],[],[]) :&
   (Edge 3,[],[],[]) :&
   (Edge 4,[],[],[]) :&
   Empty,
 nodes = Map.fromList
     [(1,'a'),(2,'b'),(3,'c')],
 edges = Map.fromList
     [(1,'p'),(2,'q'),(3,'r'),(4,'s')]
}
```

The conversion between both representations is straightforward and is available at `https://github.com/labra/haws`.

Conversely, we can also represent inductive triple graphs using general inductive graphs. As we describe in Section 5, our Haskell implementation is defined in terms of Martin Erwig's FGL library.

2.4 Algebra of graphs

Two basic operators on datatypes are the *fold* and the *map* [17] . The fold is the basic recursive operator on datatypes: any recursive function on a datatype can be expressed as a fold. Using the representation introduced above, we can define `foldGraph`:

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

The map operator applies an argument function to all values in a value of a datatype, preserving the structure. It is the basic functorial operation on a datatype. On `TGraph`'s, it takes a function that maps `a`-values in the context to `b`-values, and preserves the structure of the argument graph. We define `mapGraph` in terms of `foldGraph`.

```
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))
```

An interesting property of `mapTGraph` is that it maintains the graph structure whenever the function f is injective. Otherwise, the graph structure can be completely modified.

Example 2.10. Applying the function `mapTGraph (\_ -> 0)` to a graph returns a graph with a single node.

Using `mapGraph`, we define some common operations over graphs.

Example 2.11. 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 defined other graph functions implementing depth-first search, topological sorting, strongly connected components, etc. [2]

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 in RDF are globally denoted IRIs (internationalized resource identifiers [7]).[3] 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 .
```

[2] The definitions can be found on `https://github.com/labra/haws`.

[3] The 2004 RDF recommendation employs URIs, but the current working draft uses IRIs.

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.

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

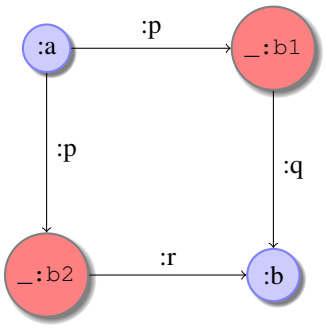

Figure 5: Example with two blank nodes

Blank node identifiers are local to an RDF document and can be described by means of existential variables [16]. 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.

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.

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

An RDF document in 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. It follows that a blank node behaves as an existential variable [16]

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 [19].

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)))
```

This `RDFGraph` encoding makes it easy to construct a number of common functions on RDF graphs. For example, two `RDFGraph`'s can easily be merged 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)
```

We define the map function over `RDFGraphs` by:

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

Finally, to define `foldRDFGraph`, we need a seed generator that assigns different values to blank nodes. In the following definition, we use 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)
```

5 Implementation

We have developed two implementations of inductive triple graphs in Haskell[4]: one using higher-order functions and another based on the FGL library. We have also developed a Scala implementation[5] using the *Graph for Scala* library.

5.1 Implementation in Haskell

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 offers a theoretical insight but is not intended for practical proposes.

[4]Haskell implementations are available at `https://github.com/labra/haws`.

[5]Scala implementation is available at `https://github.com/labra/wesin`.

The second Haskell implementation is based on the FGL library. In this implementation, 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 {
 grNode :: Node,
 edges :: Set (a,a)
}
```

`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. Any inductive triple graph can be converted to an inductive graph using Martin Erwig's approach.

5.2 Implementation in Scala

In Scala, we define a `Graph` trait with the following interface:

```
trait TGraph[A] {
  def empty : TGraph[A]

  def mkTGraph
      (triples : Set((A,A,A))): TGraph[A]

  def nodes : Set[A]

  def decomp
      (node : A): (Context[A],TGraph[A])

  def extend
      (ctx : Context[A]): TGraph[A]
```

The Scala implementation is based on the *Graph for Scala* library developed by Peter Empen. This library provides an in-memory Graph library with a default implementation using adjacency lists and Scala inner classes. It is important to notice that although the base library can employ an underlying non-purely functional approach, the API itself is purely functional.

The library contains a generic trait `Graph[N, E]` to define graphs with nodes of type `N` and edges of kind `E`. There are four edge categories: hyperedge, directed hyperedge, undirected and directed edge. Each of these categories has predefined edge classes representing any combination of non-weighted, weighted, key-weighted, labeled and key-labeled. In our case, we will employ 3-uniform directed hypergraphs given that an edge relates three elements (origin, property and destiny). The library offers both a mutable and immutable implementation of graphs.

The functions from the *Graph for Scala* library used in this paper are given in Table 1.

Our implementation defines a case class `TGraphImpl` which takes a `Graph[A,Triple]` as a parameter.

Table 1: Functions employed from the *Graph for Scala* library

`empty`	Returns an empty Graph
`nodes`	List of nodes of a graph
`edges`	List of edges of a graph. For each edge `e`, we can obtain its 3 components using `e._1`, `e._2` and `e.last`
`isEmpty`	Checks if graph is empty
`+`	Adds an edge to a graph returning a new graph. A 3-edge between `a`, `b` and `c` is expressed as `a~>b~>c`

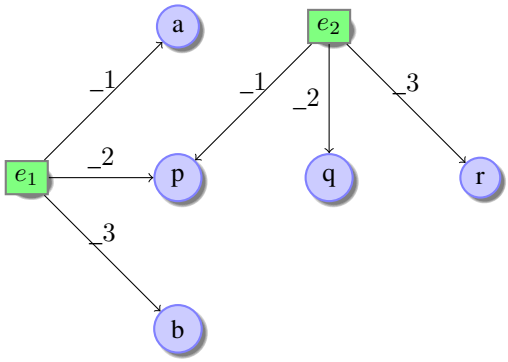

Figure 6: RDF graph as an hypergraph. e_i are 3-hypergedges

`Triple` is defined as an instance of `DiHyperEdge` restricted to hyperedges of rank 3 (triples). Figure 6 depicts the graph from Figure 3 using 3-ranked hyperedges. e_1 and e_2 are the hyperedges that relate the 2 triples.

Following is a sketch of the `TGraphImpl` code:

```
case class TGraphImpl[A]
    (graph: Graph[A,Triple])
    extends TGraph[A] {

 def empty: TGraph[A] =
    TGraphImpl(graph.empty)

 def nodes : Set[A] =
    graph.nodes.map(_.value)

def extend
   (ctx : Context[A]): TGraph[A] = {
 TGraphImpl(
  ((((graph + ctx.node)
  /: ctx.succ) {(g,p) => g +
       Triple(ctx.node,p._1,p._2)}
  /: ctx.pred) {(g,p) => g +
    Triple(p._1,p._2,ctx.node)}
     /: ctx.rels) {(g,p) => g +
    Triple(p._1,ctx.node,p._2)})}

def decomp:
    Option[(Context[A],TGraph[A])]={
 if (graph.isEmpty) None
 else {
  val node = nodes.head
  for {
   pred <- pred(node)
   succ <- succ(node)
   rels <- rels(node)
  } yield(Context(node,pred,succ,rels),
        TGraphImpl(graph - node))
 }
}
```

Notice that we employ the operator + to add elements and edges to a graph returning a new graph. The *Graph for Scala* library provides an implementation with mutable graphs, and another with immutable graphs. Since we work in a purely functional setting, we prefer to work with immutable data structures. Further work remains to be done to compare the efficiency of the implementations, or to further optimise an implementation.

A context of a node in a graph is defined with the following case class:

```
case class Context[A](
      node : A,
      pred: Set[(A,A)],
      succ: Set[(A,A)],
      rels: Set[(A,A)])
```

Following the encoding presented in previous section, we define RDF graphs as:

```
abstract class RDFGraph
case class Ground
     (graph : TGraph[RDFNode])
  extends RDFGraph
case class Exists
     (fn: BNode => RDFGraph)
  extends RDFGraph
```

where RDF nodes are defined by the `RDFNode` class.

```
abstract class RDFNode
case class IRI(iri: IRI)
  extends RDFNode
case class Literal(lit: Literal)
  extends RDFNode
case class BNode(id: String)
  extends RDFNode
```

Now, it is possible to define `mapRDFGraph` as:

```
def mapRDFGraph
   (fn : RDFNode => RDFNode,
       graph : RDFGraph
   ) : RDFGraph = {
 graph match {
   case Ground(g) =>
         Ground(g.mapTGraph(fn))
   case Exists(f) =>
         Exists ((x : BNode) => f(x))
 }
}
```

In the same way, we have defined other common functions like `foldRDFGraph`.

6 Related Work

There are a number of RDF libraries for imperative languages like Jena[6], Sesame[7] (Java), dotNetRDF[8] (C#), Redland[9] (C), RDFLib[10] (Python), RDF.rb[11] (Ruby), etc.

For dynamic languages, most of the RDF libraries are binders to some underlying imperative implementation.

banana-RDF[12] 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 fuctional implementations of RDF libraries. Most of these employ mutable data structures. For example, scaRDF[13] 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. RDF4h[14] is a complete RDF library implemented using adjacency maps, and Swish[15] 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.

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[16] to check integrity constraints of RDF [14], and propose a programming language for the semantic web [15].

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

Mallea et al [16] describe the existential nature of blank nodes in RDF. Our use of existential variables was inspired by Seres and Spivey [19] 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 these kind of datatypes. Their paper contains several examples and one of them is a representation of 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. [18] 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

In this paper, we have presented a simplified representation for inductive graphs that we called Inductive Triple Graphs and that can be applied to represent RDF graphs using existential variables. This representation can be implemented using immutable data structures in purely functional programming languages. A functional programming implementation makes it easier to develop basic recursion operators such as folds and maps for graphs, to obtain programs that run on multiple cores, and to prove properties about functions. We developed two different implementations: one in Haskell and another in Scala. The implementations use only standard libraries as a proof-of-concept without taking possible optimizations into account. In the future, we would like to offer a complete RDF library and to check its availability and scalability in real-time scenarios.

8 Acknowledgments

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

References

[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. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, 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] 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.

[17] 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.

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

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