UNIVERSITÉ
DE GENÈVE

FACULTÉ DES SCIENCES
Département d'informatique

Master of Computer Science

Software modeling and verification
HomeWork 2

ADT : graphs

Author :
Mohammad Oday Darwich

Professors :
M. Didier Buchs
M. David Lawrence

29 octobre 2015

1 Introduction

1.1 Definition of Abstract Data Type (ADT) :

Abstract Data Type is a mathematical model for data, known as **ADT**, where a data type is defined by its behavior (semantics).
To understand ADT, lets take two steps back, If we take off **Abstract** and Data from ADT, now we have type, and a **type** would be **defined** as a **collection** of **type values**, e.g., integer type, 0, 1, 2 etc.
If we add the **Data** back in we would **define data type** as a **type** and the **set of operations** that will **manipulate the type**, e.g., addition, subtraction, and multiplication are operations that can be performed on the integer data type.
So, we can define an **abstract data type** (**ADT**) as a **data type**, that **is a type and the set of operations** 4 **that will manipulate the type**, e.g., for the integer data type, the operations might be delete an integer, add an integer, print an integer, and check to see if a certain integer exists.

In this TP, we will create ADT to define oriented graphs, i.e., directed graph is an ordered pair G = (V, A) with

- V a set whose elements are called vertices, nodes, or points ;
- A a set of ordered pairs of vertices, called arrows, directed edges (sometimes simply edges with the corresponding set named E instead of A), directed arcs, or directed lines.

To do this, we will use an eclipse plugin called **ALPiNA**, i.e., **Al**gebraic **P**etri **N**ets **A**nalyzer and is a model checker for Algebraic Petri Nets. And with the rewriting tool in ALPiNA we will check with our solution that all the theorems defined in graph.adt are validated. So, we will construct graph.adt file by replacing the *** with the adequate expressions.

1.2 File structure :

In this section i am describing each part enumerated in blue from 1 two 5 on left side, and each part i explain the code line by line (you see on the left the gray background the number of the code line on the **graph.adt** file).
And in the TP after completing my axioms i did tests to verify it (you can find code A lines and tests B on the annex).

1 **The head** of the **graph.adt** *.adt* file contains all file that we may use to construct our axioms.
In this TP we need to import boolean.adt.

```
2 import "boolean.adt"
```

in which we have added the axioms to define the behavior of the operators **not** and **and** as we have seen in the exercices. I also added the axioms of the operator **or** which i want to use it after in axioms like $canReach(,,)$ and $existsCycle(,)$.

```
18     or(true, $bv) = true;
19     or(false, false) = false;
20     or(false, true) = true;
```

And also in the **Head** we write the name of the *.adt* file that we are workin on.

```
4 Adt graph
```

2 Sorts : an ADT is a set of values, and the Sorts are names of Abstract data type.

3 Generators : Define Functions and The base cases.
As we see on the theorems in the bottom of the file graph.adt we have 4 nodes, so we complete defining the 4 nodes A, B, C, D by replacing the *** as you see below.

```
10  Generators
11
12    A : node;
13    B : node;
14    C : node;
15    D : node;
```

```
18    edge : node, node -> edge;
```

The method edge is to create an edge from the pairs of nodes.

```
20    empty : graph;
```

empty is type of graph and we define the graph inductivly (as **AT)**

```
21    noedge : edge;
```

I want to use this on the axiom getEdge 5

```
22    cons: edge, graph -> graph;
```

We want to define inductivly a graph so the result of cons should be graph.
and as we on the **(L49)** 5 $cons(\$e1, \$g)$ take $\$e1$, so edge as first parameter and graph $\$g$ as second parameter.

4 Operations 1.1 : As we talked above, the set of operations is to manipulate the type, for the edge data type, the operations might be delete an edge, add an edge etc...

```
24  Operations
```

4.

```
26      source : edge -> node;
```

The source of the edge is a node, from where the edge is starting, throughout his direction.

```
28      target : edge -> node;
```

The target of an edge is a node so this functions return the node, to where the edge is pointing and it take an edge as parameter.

```
30      remove : graph, edge -> graph;
```

This function remove an edge from the graph so it take as parameter a graph and an edge.

```
32      isTerminal : graph, node -> bool;
```

This function as we see at the first sight in **Line 53** return false so it return bool and also the second parameter is node to it take $\$x$ as second parameter. and that mean that if the graph is empty, then no terminal for the graph.

```
34      existsEdge : graph, node -> bool;
```

This function test if there are an edge (true) from a node or not (false) so it take graph and node as parameter and it return boolean.

```
36      getEdge : graph, node -> edge;
```

Given a node it return the first edge so the second parameter is node.

```
38      canReach : node, node, graph -> bool;
```

Given the theorem below of line 77

```
93      canReach(A, D, cons(edge(A,C), cons(edge(A,B), cons(edge(B,C), cons(edge(D,
        C), cons(edge(C,A), empty))))) = false;
```

we see that it return boolean because it will test if a node can be reached from itself 5 or from another given node(true) or not. and it takes as parameter node, node, graph (we define it respecting the given order)

```
40      existsCycle : node, graph -> bool;
```

Given the theorem below of line 96

```
96      existsCycle(A, cons(edge(A,C), cons(edge(A,B), cons(edge(B,C), cons(edge(D,
        C), cons(edge(C,A), empty))))) = true;
```

we see that it return boolean and it takes as parameter node, node, graph (we define parameters respecting the given order), so it Test if a node belongs to a cycle (true) or not(false).

5 Axioms : Defines relations between operations, a self-evident truth, or a proposition whose truth is so evident as first sight.

```
44    source(edge($x, $y)) = $x;
```

the source of an edge is x, so it return $\$x$.

```
45    target(edge($x, $y)) = $y;
```

The target of an edge is the second parameter, so It return the node in which is the direction of this edge 4, so it return $\$y$.

```
49    if $e = $e1 then remove(cons($e1,$g), $e) = $g; //3
```

if $e = e1$ It remove the edge e and it return a graph. as we see in the line below

```
48    remove(empty, $e) = empty; //2
```

so there is no change for the graph even if we delete an edge, so it return $\$g$

```
50    if $e != $e1 then remove(cons($e1,$g),$e) = cons($e1,remove($g,$e));
```

if e different then $e1$ so adding $e1$ to g then remove e, is the same as remove e from g and then add $e1$.

```
53    isTerminal(empty, $x) = false;

54    if $g != empty & existsEdge($g,$x) = false then isTerminal($g,$x) = true;
55    if $g != empty & existsEdge($g,$x) = true then isTerminal($g,$x) = false;
```

The given case, line 53, we see that if no graph then retrun false (node is not terminal). So here there is an essential condition should be considered, i.e., the graph is not empty otherwise the test $isTerminal(\$g, \$x)$ will give us false all times. And by drawing some graph, we conclude that a node can be terminal *if and only if* it's not a source for any edge, that's why i added the second condition (existsEdge($g,$x) =true) and for the 2^{nd} case, if node is a source for one of edges then the node is not terminal. Also, we can achieve this axioms in another way using $getEdge(\$g, \$x)$.

```
59    getEdge(empty, $x) = noedge;
60    if source($e) = $x then getEdge(cons($e,$g),$x) = $e;
```

L59, It retrun *noedge* 3 if the edge do not exist.
L60, If $\$x$ is the source of e then it will return the edge which has this node x as a source

```
61    if source($e) != $x then getEdge(cons($e,$g),$x) = getEdge($g,$x);
```

If $\$x$ is not the source of e then it will go recursively throughout the graph but { *if we can say* the *new graph* or *reduced graph* } do not contains the edge e, which e has as source x, so it's like going throughout the rest of the reduced graph from the edge e.

```
67    existsEdge(empty, $x) = false;
```

If the no graph then no existing edge then return false.

```
68    if $x  = source($e) then existsEdge(cons($e, $g),$x) = true;
```

If $\$x$ is the source of e then the existsEdge function will return true because the graph contain the edge e which has this node x as a source.

```
69    if $x != source($e) then existsEdge(cons($e, $g),$x)  = existsEdge($g,$x);
```

If $x is not the source of e then the existsEdge function will go recursively throughout the graph g until it find the edge otherwise it will return false if there are no edge corresponding to this edge e. The mechanism is similar to getEdge.

```
75    if $x = $y then canReach($x, $y, $g) = true;
```

As we have seen above, a node can reach itself 4 so considering the condition (same node) it return true.

```
76    if $x != $y then canReach($x, $y, empty) = false;
```

if there are no edge then x cannot reach y, and in this case there is no graph $canReach(\$x, \$y, empty)$ then no edges then the result will be false.

```
77    if $x != $y & $g != empty & existsEdge($g, $x) = false then canReach($x, $y
      , $g) = false;
```

if $x is different then $y and the graph is not empty and there are no existing edge for $x, *in other words* x is not the source for no one edge, then it cannot reach the node from x and the result of test is false. and in another way to do it, if x is terminal also cannot be reached from another node (not itself).

```
78    if $x != $y & $g !=empty & existsEdge($g, $x) = true then canReach($x, $y,
      $g) = or(canReach(target(getEdge($g,$x)), $y, remove($g, getEdge($g, $x))),
      canReach($x,$y, remove($g, getEdge($g, $x))));
```

if $x diffrent then $y and $g not empty and $x is a source of an edge then we do the test recursivly because we will be in front of many cases.
At the first time if we get the first edge $canReach(target($getEdge($g, $x)), .., ..)$ and we find that we cannot reach this node $x from $y then the result will be false ! but no, here appear the second case that with from the node $x which is source of the edge e(first edge) we cannot reach it and maybe there are another way to reach it given another node $y .
so what i am doing next is to remove the edge the i passed (just before), reducing my graph (obtaining graph without the passed edge)
and do the test again and again so the test will will pass by the axioms of canReach L-74, 75, 76 and it will return the corresponding result. true if the first case was founded and false in the rest.

```
85    if existsEdge($g, $x) = false  then  existsCycle($x,$g)  = false; // si
      dans le graphe pas d'edge qui part de $x
```

If there is no edge given a node $x result is false, *(no edge mean graph empty so no cycle because no edge).*

```
86    if existsEdge($g, $x) = true  then existsCycle($x,$g) =  or(canReach(
      target(getEdge($g,$x)), $x, remove($g,getEdge($g,$x))), existsCycle($x,
      remove($g,getEdge($g,$x))));
```

In this axioms we tread all cases.
If there is an edge starting from the node $x then i am testing if i's reachable from the target of the edge of $x, $x is source
And in $canReach(,\$x,)$ the second parameter is x so it means that x is also the destination of the edge then there are a cycle.
But in case if $x is not target of the first edge, so here the third case appear and the test go recursively (same to can reach) , $->$ removing the passed edge, $->$reducing my graph, and $->$ do the test of existsCyle again until it find the source is the same as the destination in the new graphs, otherwise the result will be false.

6 Theorems : A theorem is a statement that can be demonstrated to be true by accepted mathematical operations and arguments.

7 Variables : It's a generic element of sorts.

2 Conclusion

I find an interesting thing in ADT that we can do the recursivity and it's so efficient especially in computer science. We can also construct different axioms independently from each other, But i see that it's difficult to construct an axiom too independent from another axiom which treat another case (like if i change the first axiom of exist cycle, in a second case for this axioms most of time they are dependent and it's too difficult to separate) because if i introduce an axioms in another one and i want to change one of them that may cause a fault result which is not good and also we cannot know if we don't know what result we are attending after doing the tests.
After this TP i can say that i understand very well how ADT works, and i also learn to do not give up in a TP !.

A Code

B Some Test

```
1
2  \\
3  target(edge(A,B))
4
5  \\
6  existsEdge(cons(edge(B,C), cons(edge(D,C), cons(edge(C,A), empty))),C)
7
8  \\Test to is Terminal
9  isTerminal(cons(edge(B,C),C))
10 isTerminal(cons(edge(B,C), cons(edge(D,C), cons(edge(C,A), empty))),A)
11 isTerminal(cons(edge(A,C), cons(edge(A,B), cons(edge(B,C), cons(edge(D,C), cons(edge
       (C,A), empty))))),A)
12 isTerminal(cons(edge(A,B),empty),B) > if ture
13 isTerminal(cons(edge(A,B),empty),A) > if false
14 isTerminal(empty,A)
15
16 \\ test de remove :
17 remove(cons(edge(D,C), cons(edge(C,A), empty)),edge(C,A))
18 remove(cons(edge(D,C), cons(edge(C,A), empty)),edge(C,A))
19       cons(edge(D, C), empty)
20
21 //test of Get edge
22
23 getEdge(cons(edge(D,C), cons(edge(C,A), empty)),D)  --> for the first case|| the
       edge of D
24 getEdge(cons(edge(D,C), cons(edge(C,A), empty)),D)  --> the second case || there are
       no edge D
25 getEdge(cons(edge(D,C), cons(edge(C,A), empty)),C)  --> if the a node is not the
       first go recusivly
26
27 \\ existsCycle
28 existsCycle(D,cons(edge(D,C), cons(edge(C,A), empty))
```