Modeling of Tree Topology Using Coq

Benjamin Collet
M1 MoSIG, UGA
Grenoble, France
benjamin.collet@etu.univ-grenoble-alpes.fr

Supervised by: Karine Altisen, Pierre Corbineau and Stéphane Devismes.
Collaboration with Andrey Sosnin.

Abstract

During this internship, we have designed a tree topology model using the proof assistant Coq. This topology will be used inside a framework for certified proofs of self-stabilizing distributed algorithms. We first define a correct and complete model of the topology in Coq. Then we refine this model by adding definitions and proving trivial properties on these definitions. We also discuss about the relevance of our model.

1 Introduction

A proof assistant is a program that helps writing and verifying mathematical proofs by mechanically checking assertions. Coq [Coq, 1999; Bertot and Castéran, 2004] is the proof assistant developed at INRIA, with the participation of CNRS and other french research institutions. It has been used to prove some famous theorems, including the four color theorem [Gonthier, 2008]. Coq is based on the Calculus of Constructions, developed by Coquand and Huet in [1988], more precisely, on a variant that includes inductive types, the Calculus of Inductive Constructions [Paulin-Mohring, 1993]. The Calculus of Constructions can be seen as an extension of the Curry-Howard isomorphism which makes a strong connection between proofs and programs, and between theorems and types.

A self-stabilizing algorithm is a distributed algorithm that can support any finite number of transient faults, and go back to a correct behaviour after a finite amount of time. This notion was introduced by Dijkstra in [1974], since then, more and more complex algorithms have been proposed. As algorithms gain in complexity, proofs of their correctness and complexity are trickier to establish. These proofs, written by hand and based on informal reasoning, potentially contain errors due to arguments not being perfectly clear, as explained by Lamport in [2012].

It is in this context that the framework PADEC (*Preuves d'Algorithmes Distribués En Coq*) is developed [Altisen *et al.*, 2016]. PADEC is a software composed of specifications and proofs written in Coq. This framework helps building certified proofs of self-stabilizing algorithms that are based on the locally-shared memory model with composite atomicity [Dijkstra, 1974], the most commonly used model in the self-stabilizing area.

Usually, distributed algorithms are described for a specific shape of network, e.g. rings, trees, grids, etc. The subject of this internship was to design one topology. The ring is already implemented in the framework, I used it as example for designing trees. The main task was to define a specification; even if this was an original work, I was guided by Pierre. Difficulties of this task will be discussed in the next paragraphs.

The next step was to ensure that the specification was accurate and usable. This was achieved by adding definitions (e.g. ancestor, depth, leaf...) and proving trivial properties about these definitions. In addition to confirming the design, these definitions and properties will be useful when using the topology, as they can be part of more complex proofs. I was mostly autonomous in the choice of definition and property to add, except for a couple of target properties that are needed for some proofs.

The goal we want to achieve is to provide an abstract specification of trees. The specification, written in Coq, is a set of symbols, functions and predicates, with some formulas that must be satisfied. This specification leads to an infinite set of logical consequences that composes a theory of trees.

Our specification must be necessary and sufficient. Necessary means that all the necessary constraints are in the specification, that way, only trees can be accepted by the specification. Sufficient means that the specification can accept any tree. In practice, it consists in finding the right balance between a light specification that accepts all the trees but strong enough to reject everything else.

It is harder to provide all the necessary properties than ensure that they are sufficient. As we usually ponder about our specification with trees in mind, finding a model that represents a tree, but does not fit in the theory is easier than to think of something that is accepted by the theory, but is not a tree.

The rest of this paper is organized as follows. Section 2 analyzes the existing solutions, and derives constraints. Section 3 describes how we have modeled the tree topology and definitions that come with it. In Section 4, we discuss about some elements of the definition described in the previous section. We make concluding remarks in Section 5.

2 Preparatory work

2.1 Preparatory study: the Ring case

Prior to this internship, a directed ring topology was already defined. Because rings are simple, they where used as a proof of concept and as reference for other topologies, e.g. tree or grid. Intuitively, a ring is a graph composed of one cycle, in which every node has a unique successor. By applying the successor operation a certain finite amount of time (the size of the ring), we eventually get back to the initial node.

The specification of the ring in PADEC is made of three functions. `Succ: A → A`, which for each node, associates its successor. `Pred: A → A`, which does the same for its predecessor. And finally, `Dist: A → A → nat`, which for each pair of nodes, gives back the distance between the two nodes, following successors direction.

Once these functions where defined, predicates that link `Pred`, `Succ` and `Dist` have been added. For instance, we have `Dist x (Succ y) = 1 + Dist x y`.

The distance, expressed as a natural number, is the key point of that specification, it makes easy induction schema. As a reminder, induction proofs are done in two steps, the base case, and the induction step. In a ring, an induction reasoning for some property `P` on nodes, will look like that:

- base case: for a given node `x` in the ring, (`P x`) is valid;
- induction step: for every node `y` in the ring, (`P y`) is valid implies that (`P (Succ y)`) is also valid.

If the base case and the induction step can be proved, then for every `z` in the ring, the property (`P z`) holds. Natural numbers help us during the induction step, as `Dist` from `x` give a unique result for each `y`. This allows to report a lot of the reasoning onto naturals, which are easy to manipulate.

Many usual properties about rings can also be derived from the definition, e.g. considering two nodes v and w, the distance from v to w is the same as the distance between their respective successors.

This specification succeeds at representing every ring and only rings. We can make this assertion with confidence as usual properties have been proved, and the specification has been instantiated with every ring based on $\mathbb{Z}/_{k\mathbb{Z}}$. Unfortunately, completeness cannot be certified inside Coq, as a consequence of Gödel's incompleteness theorems, that apply to every theory that is complex enough to describe basic arithmetic.

2.2 Functional constraints

Many definitions of trees exist in graph theory. One of the most common is: a tree is a graph $T = (V, E)$ such that the graph is connected and $|E| = |V| - 1$. This definition makes an undirected and unrooted tree. Because of the cardinalities, this definition also implicitly makes the tree finite. Finiteness of the tree is a property we want to keep, as we aim to describe network topology. However, most algorithms require a rooted tree, so we cannot use that definition.

Another definition would be: a tree is a graph $T = (V, E)$ such that each node has a unique parent, except a unique node called "root", with no parent. In order to avoid describing a tree plus disconnected rings, the graph also needs to be acyclic, or connected (one implying the other).

PADEC already contains a tree specification, that has been used to represent spanning tree in proofs. This specification uses the former definition (with the acyclic property), but appears to be difficult to use. Proofs on trees are usually done using structural induction. However, structural induction is not "built-in" in this specification, unlike in rings.

With the current specification, induction requires to prove well-foundedness of the tree, and to perform proofs on it. A binary relation on a set is called well-founded if all strictly decreasing sequences are finite. For trees, it means that if we follow a path from child to child, we will always encounter a leaf (a node without children) in a finite amount of step, regardless of which child we chose at each step. While Coq makes it possible to do induction based on well-founded schema, induction-based reasoning on naturals are easier.

In this work, we present another specification of trees that is also based on this definition. We keep the structural information: each node has a unique parent, except for one root, and we add a natural quantity to ease the use of proof by induction, like for the rings. This quantity will also make the graph connected, and thus, complete the definition.

2.3 Coq-related technicality

The replacement principle is widely use in interactive theorem proving. It consist of replacing an expression by another after having proved their equality. Equality is defined on a type by a function `eq (A:Type) (x:A): A → Prop`, and `eq A x x` should be true.

The default equality in Coq is the intentional equality, sometimes call Leibniz equality. This is the equality of the code in a λ-calculus context. For instance, $1 + 1 = 2$ can be written as $(\mathrm{S}\,0) + (\mathrm{S}\,0) = \mathrm{S}(\mathrm{S}\,0)$, which gives $\mathrm{S}((\mathrm{S}\,0) + 0) = \mathrm{S}(\mathrm{S}\,0)$ and finally $\mathrm{S}(\mathrm{S}\,0) = \mathrm{S}(\mathrm{S}\,0)$.

In the case of a function, it is the equality of the normal forms. Let's try to prove that `(fun x ⇒ x) = (fun x ⇒ 0 + x)`. `(fun x ⇒ x)` is already in a normal form. The normal form of `(fun x ⇒ 0 + x)` is written `(fun x ⇒ match 0 with 0 ⇒ x | ... end)`, which can be simplified in `(fun x ⇒ x)`. Both expressions have the same normal form, so they are equal.

Now same question with `(fun x ⇒ x + 0)`. This time, the normal form is `(fun x ⇒ match x with 0 ⇒ 0 | S x′ ⇒ S (x′ + 0) end)`, and cannot be reduced any further. While this function has the same images as `(fun x ⇒ x)`, we don't have an equality according to Coq. The equality on the images, that might be more suitable in this case, is called extensional equality.

For every type where intentional equality is not satisfactory, we can add an ad hoc equality. The resulting types, equipped with an equality relation, are called setoids.

For simple types, the equality relation is an equivalence relation (reflexive, symmetric and transitive). For more complex type, sometimes, the equality is not reflexive anymore. This is the case with the functions and the extensional equality (equality of their image). Let's consider two function from type `A` to type `B`, named f and g. On type `A`, we have the equality `eqA` and on type `B`, `eqB`. To say that f and g are equal, we need to prove `forall x y, eqA x y → eqB (f x) (g y)`. Even in the case where `eqA` and `eqB` are reflexive, we can have `eqA x y` and not `eqB (f x) (f y)`, e.g. f is the identity, `eqA` is the equality modulo 2, and `eqB` is the Leibniz equality.

As non-reflexive equality prevents the substitution of an expression by another, when the setoid relation is not reflexive (partial equality relation), we restrict our reasoning to elements for which we can prove the reflexivity. In the case of functions, these elements are called compatible functions.

3 Definition of a Tree Topology

3.1 Construction of the nodes

Our model is a class `Tree_Topology` that describes relations between nodes that compose this topology. Nodes are represented by a type `A` which is a setoid. Relations between these nodes are defined as functions. The first relation is the parent relation, represented by the function `Parent: A → option A`. This is a partial function since root does not have a parent.

Unfortunately, Coq only allows total functions, as it allows a strong static type safety, which is really useful in the context of mechanical proofs. In order to use partial functions, a useful type to know is the polymorphic type `option`. Used with the type `A`, an (`option A`) value can either hold `None`, which usually indicates a non-significant result of a function, or hold `Some a` where `a` is of type `A`. This way, we can have partial functions, while Coq maintains its strong static type safety.

The algorithms that the framework aim to prove, are suppose to represent the real-world, where networks are of finite size. To achieve finiteness, the solution retained was to add the list of all the nodes in the tree, i.e. there exists a list such that, for every node, this node is in that list. We discuss about that list in the last paragraph of Section 4

From the function `Parent` and the list of all the nodes, we can provide some definitions. By opposition to `Parent`, a node v is the child of a node w if w is `Parent` of v. Also, two nodes are called siblings if they share the same parent. The list of children of a node can be build from `Parent` and the list of all the nodes, and give the function `Children: A → list A`. If the list of children is empty, then the node is a called a leaf.

3.2 Relations between the nodes

The second function is the distance to the lowest common ancestor described by the function `Dist_LCA: A → A → nat`. The lowest common ancestor (LCA) of two nodes v and w is defined as the common ancestor of v and w that is the farthest from the root. In Figure 1, the LCA of the nodes v and w is the node 'b'. The `Dist_LCA` from v to w is the distance from v to 'b', so 1. We can note that this is not symmetric as `Dist_LCA w v = 2`.

Now that we have `Dist_LCA` in addition to `Parent` and the list of all the nodes, we can add some definitions. In order to properly define a finite tree, a root must exist and be unique (e.g. node 'a' in Figure 1). As the distance to the LCA is

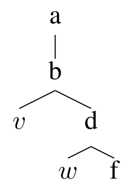


Figure 1: An example.

defined for all pairs of nodes, a common ancestor of all nodes must exists.

The depth of a node is defined as the distance to the root. The height of a node is defined as the longest distance to a descendant.

An ancestor v of a node w is a node where the property `Dist_LCA v w = 0` holds. v is a LCA, and therefore an ancestor (see Figure 2a). Note that this definition includes w as an ancestor of itself. Conversely, a descendant v of a node w is a node where `Dist_LCA w v = 0` (Figure 2b).

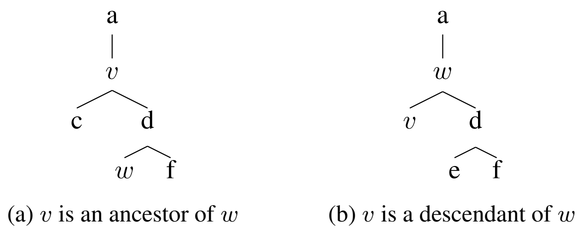


Figure 2: Ancestor and descendant with respect to `Dist_LCA`.

Now that we have defined the functions `Parent` and `Dist_LCA`, and used them is some definitions, we need to specify how they behave with respect to each other. Let's consider two nodes x and p such that `Parent x = Some p`, then `Dist_LCA x p = 1` and `Dist_LCA p x = 0`.

Let's consider two other nodes v and w. If we replace the first parameter, v, by its parent Pv, `Dist_LCA` will decrease of 1 (Figure 3a), except if already 0 (Figure 3b).

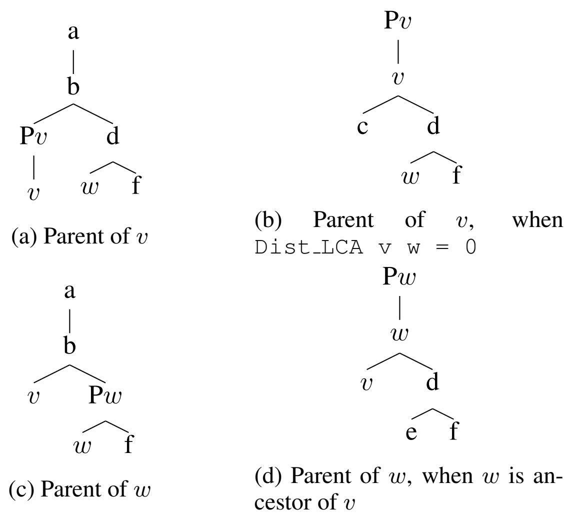
```
forall v w Pv, isParent Pv v ->
    Dist_LCA Pv w = pred (Dist_LCA v w).
```

If we replace the second parameter, w, by its parent, Pw, we face two cases. Either w is not an ancestor of v, so the LCA does not change, and `Dist_LCA` remains the same (Figure 3c).

```
forall v w Pw,
    isParent Pw w -> ~(isAncestor w v) ->
        Dist_LCA v w = Dist_LCA v Pw.
```

Either w is an ancestor of v, so the new `Dist_LCA` is the distance to Pw, and we need to add 1 (Figure 3d).

```
forall v w Pw,
    isParent Pw w -> isAncestor w v ->
        Dist_LCA v w = 1 + Dist_LCA v Pw.
```



(a) Parent of v

(b) Parent of v, when `Dist_LCA v w = 0`

(c) Parent of w

(d) Parent of w, when w is ancestor of v

Figure 3: How `Dist_LCA` behaves.

3.3 Validation of the theory

To acknowledge the soundness of our specification, we need to derive usual properties of the models we are describing. For instance:

- By definition, the depth of the root is 0 and the height of a leaf is 0.
- A leaf does not have children, it implies that is if v is a leaf and also an ancestor of w, then w is equivalent to v (n.b. our definition of ancestor includes the node itself).

```
forall v w,
    isLeaf v -> isAncestor v w ->
        v == w.
```

- The depth of a node is the depth of its parent plus 1, except for the root as mentioned earlier.

```
forall v w,
    isParent v w ->
        Depth w = 1 + Depth v.
```

- The height of a node v is greater or equal to the height of any of its children plus 1.

```
forall v w,
    isParent v w ->
        Height v >= 1 + (Height w).
```

From the distance to the LCA, we can define the usual distance between two nodes, v and w, as the sum: `Dist_LCA v w + Dist_LCA w v`. This distance is reflexive, symmetric and respects the triangle inequality.

As explain in Subsection 2.1, having `Dist_LCA` expressed as a natural number eases the use of proof by induction. Induction is useful to prove properties that are valid in a whole part of the tree. We can define both downward and upward induction.

Downward induction: Let `P` be a property that propagates from any node to all of its children, then, if it holds for a node v, by induction, it propagates to all descendants of v, i.e. the subtree rooted in v. As a corollary, if `P` holds for the root, then it holds for the whole tree.

Upward induction: Let `Q` be a property that propagates from any node to its parent. If `Q` holds for a node w, it hold for all ancestors of w, i.e. the path from w to the root. As a corollary, if `Q` holds for all leafs, it holds for the whole tree.

To gain confidence in the expressive power of our specification, we also need to verify that it can represent any trees. The usual method for this is to build a model that verifies our specification from a model of another specification. As we are basing our specification on an existing specification of trees, we should also prove that they are equivalent. However, both methods are complex and time consuming, and are not in the scope of a two months internship.

4 Discussion about the model

The `Parent` operation induces a direction between nodes, as a consequence, the resulting tree is a rooted tree. Also, `Parent` is uniquely defined for any given node, so the model cannot describe a polytree or any kind of directed acyclic graph. In addition, only a unique path from a node to any of its ancestor exists.

`Dist_LCA` is defined for any pair of nodes, in other words, there is a path between any pair of nodes. This prevents description of a forest as the graph described must be connected. Also, `Dist_LCA` yields a unique value, thus preventing cycles. Indeed, for a node v to be an ancestor of one of its ancestor w, both `Dist_LCA v w` and `Dist_LCA w v` need to be equal to 0, and this defines the equivalence between v and w.

Together, the existence of a common ancestor and the unicity of the path to it, ensures that it exists a unique path from a node to another. This property itself is sufficient to properly describe a tree.

The topology described in this paper will be used to help the proof of distributed algorithms. These algorithms are to be used in the context of finite networks. The solution adopted was to add the list of all the nodes. Lists are finite in Coq, which make the size of the topology finite too. This solution is already used in the framework: the `Network` class, also contains a list of all the nodes.

5 Conclusion

We proposed a specification of trees, in Coq, that can be use inside the PADEC framework, as a network topology to prove self-stabilizing distributed algorithms. This specification needed to be accurate and usable. The specification is made of operators that create nodes that populate the models, and properties that describe relations between these nodes. Finally, properties where added to ensure the specification accepts all trees and only trees.

At the time of the writing, the specification has been written in Coq. Some of the usual properties of trees have been proved. The others, with more complex proofs, are defined in Coq, but admitted. Next step would have been to express the different induction schemas. Overall, we are confident about the results we have obtained during this internship and they will be integrated inside PADEC.

References

[Altisen and Corbineau, 2016] Karine Altisen and Pierre Corbineau. A framework for certified self-stabilization. `http://www-verimag.imag.fr/~altisen/PADEC/`.

[Altisen *et al.*, 2016] Karine Altisen, Pierre Corbineau, and Stéphane Devismes. A framework for certified self-stabilization. In *Formal Techniques for Distributed Objects, Components, and Systems*, pages 36–51, Cham, 2016. Springer International Publishing.

[Bertot and Castéran, 2004] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Springer-Verlag Berlin Heidelberg, 2004.

[Coq, 1999] The Coq Proof Assistant. Reference manual. `https://coq.inria.fr/refman/`.

[Coquand and Huet, 1988] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2):95–120, 1988.

[Dijkstra, 1974] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[Gonthier, 2008] Georges Gonthier. Formal proof—the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, Dec 2008.

[Lamport, 2012] Leslie Lamport. How to write a 21st century proof. *Journal of Fixed Point Theory and Applications*, 11(1):43–63, Mar 2012.

[Paulin-Mohring, 1993] Christine Paulin-Mohring. Inductive definitions in the system coq rules and properties. In *Typed Lambda Calculi and Applications*, pages 328–345, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.