A Mechanization of Phylogenetic Trees

Mamoun Filali
filali@irit.fr

IRIT CNRS
Université Paul Sabatier
118 Route de Narnonne
F-31062 Toulouse France

Abstract. We study the mechanization of phylogenetic trees in higher order logic. After characterizing trees within such a logic, we state how to reason and to compute about them. We introduce the so called generative partitions and relations whose purpose is to allow the reconstruction of a tree from its leaves. After introducing tree transformations, we define the graft operation. and consider sufficient conditions for the preservation of the generative partitions or relations after a graft. It follows that we can reconstruct a tree given its set of leaves and its generative relation which has been preserved along the growth of the tree. We apply this result to the reconstruction of a distributed computation.
keywords: HOL, tree structure, verification, ISAR.

1 Introduction

This paper gives a definitional formalization, in higher order logic (HOL), of phylogenetic trees. We also formalize how to reason and compute on such trees. We define the notion of a generative relation, that aims at characterizing information which enables to rebuild a tree. Finally, we propose a reconstruction algorithm based on the set of leaves and a generative relation. The correctness of the algorithm is established. We introduce an operation, the graft, that allows to represent the growth of a tree. a graft are stated. We illustrate such a reconstruction through the so-called leaf vectors and a concrete generative relation. It should be stressed that our study is not only concerned with the proposal of an original algorithm but also by the formal definitions and proofs within a logical framework.

The rest of this paper is organized as follows: Section 2 gives the representation and the basic operations. Section 3 introduces the graft operation and studies its reconstruction properties.Section 4 presents a concrete example where we apply the reconstruction algorithm. Section 5 contains the conclusions and related works.

2 A phylogenetic tree representation and basic operations

In this section, we introduce the formal representation of phylogenetic trees; For such a representation, we consider how to reason about it and how to compute on it. We rely mainly on basic set theory. However, rather than working with set theory only, we use type theoretic reasoning also. We have done the mechanization within the Isabelle logical framework [13]. Actually, we have used the Isabelle/Isar[1] [19] environment which goal is to assist in the development of human-readable proof documents composed by the user and checked by the machine.

2.1 Notations and basic definitions

In this section, we recall the basic set theory and order notions, we will use.We hope that the name of the definitions and their formal expression are self explanatory. We have used the definitions given in [6]. Moreover, we express them in the Isabelle syntax [13]. For each definition, first, we give its signature, then its formal expression. For instance, we have used the following definitions:

```
"Maximal ≜ λ S. {m ∈ S. ∀ m' ∈ S. m ⊆ m' ⇒ m = m'}"

"Down ≜  λ(S,e). {s ∈ S. s ⊆ e}"

"PDown ≜  λ(S,e). {s ∈ S. s ⊂ e}"

--{* proper partition*}
"PPartition ≜ λ (n,S).Partition(n,S) ∧ (∀ e ∈ S. e ⊂ n)"

"A//r ≜ ⋃x ∈ A. {r``{x}}"  -- {* set of equiv classes *}
```

In Isabelle, the reflexive transitive closure of relation `r`, denoted `r^⋆`, is introduced as an inductive data type [2]. Its introduction rules are `rtrancl_refl` which specifies that every couple `(a,a)` belongs to the transitive closure, and `rtrancl_into_rtrancl` which specifies that if `(a,b)` belongs to `r^⋆` and `(b,c)` belongs to `r`, then `(a,c)` belongs also to `r^⋆`.

```
inductive "r^*"
intros
  rtrancl_refl : "(a, a) ∈ r^*"
  rtrancl_into_rtrancl :
   "(a, b) ∈ r^* ⟹ (b, c) ∈ r ⟹ (a, c) ∈ r^*"
```

With respect to the proofs, we have used the Isabelle/Isar format. A proof is established by a sequence of intermediate results which has to be proved recursively or already established. Eventually, results are justified either as axioms of the logic or by rules of the logic. With respect to proofs, Isar promotes the

[1] "Isar" abbreviates "Intelligible semi-automated reasoning".

so called "declarative style" [18] which is closer to the usual mathematical reasoning than the procedural format. Let us mention that, basically, Isar supports natural deduction but also supports calculational reasoning [7].

As an example, the following statement which consists in assumptions[2] (**assumes**), a conclusion (**shows**) and a proof script (**proof**) establishes that the union of two hierarchies (see section 2.2) is also a hierarchy. A basic statement of the proof has the format:

from ⟨facts⟩ **have** label ′ :′ ⟨proposition⟩ **by** ⟨method⟩

which aim is to establish `proposition` from `facts` by applying `method`.

```
theorem Hierarchy_union:
 assumes h1:"H1 ∈ Hierarchy"
 assumes h2:"H2 ∈ Hierarchy"
 assumes s: "∀ n1 ∈ H1. ∀ n2 ∈ H2. SDS(n1,n2)"
 shows "(H1 ∪ H2) ∈ Hierarchy"
proof -
 from h1 h2 have e: "∅ ∉ H1 ∪ H2" by (unfold
    Hierarchy_def, blast)
 from s have "∀ n1 ∈ H1. ∀ n2 ∈ H2. SDS(n2,n1)"
   by (auto simp only: SDS_def)
 from this have "∀ n1 ∈ H2. ∀ n2 ∈ H1. SDS(n1,n2)" by
    auto
 from s this h1 h2 have sds: "∀ n1 ∈ H1 ∪ H2. ∀ n2 ∈ H1 ∪
     H2. SDS(n1,n2)"
   by (unfold Hierarchy_def, blast)
 from h1 h2 have f: "finite (H1 ∪ H2)" by (unfold
    Hierarchy_def, auto)
 from h1 h2 have "∀ n ∈ H1 ∪ H2. finite n" by(unfold
    Hierarchy_def, auto)
 from e sds f this show ?thesis by (unfold Hierarchy_def,
     blast)
qed
```

2.2 Hierarchies and trees

Our mechanization is based on the introduction of phylogenetic trees starting from the basic notions of set theory. For such a purpose, we first consider hierarchies [3] and then introduce trees as restricted hierarchies. Along with these hierarchies, we give some general definitions that will be used later.

[2] Sometimes assumptions are also called preconditions.

The basic idea of the following representations is to infer a structure from the relations between its elements; the structure is not encoded directly. Such a content based encoding is motivated by the fact that our basic concern is the reconstruction starting from *some* of the elements, namely the leaves, of the tree structure.

Hierarchies. We first introduce a generic `graph` as a set of nodes. A `node` is a set of generic elements.

```
types
  'e graph = "('e set) set" -- {* generic graph *}
  'e node = "('e set)" -- {* generic node *}
```

Hierarchies are finite graphs which elements are finite and non empty and obey to the `SDS`: “Subset Disjoint Subset” relation:

```
  "SDS ≜ λ(s1,s2). s1 ⊆ s2 ∨ s1 ∩ s2 = ∅ ∨ s2 ⊆ s1"

  "Hierarchy ≜ {H.    finite(H)
                    ∧ (∀ n ∈ H. finite(n))
                    ∧ ∅ ∉ H
                    ∧ (∀ n1 ∈ H. ∀ n2 ∈ H. SDS (n1,n2))
               }"
```

In the following, we give the formal definitions that will be used.

```
  "Leaves ≜ λ h. {l ∈ h. PDown(h,l) = ∅}"

  "ROOT ≜ λ h. ⋃ h"

  "Subtrees ≜ λ t. image (λ e. Down(t,e)) (Maximal(t))"

  -- {* proper subtrees *}
  "PSubtrees ≜ λ t. Subtrees (t - Maximal(t))"

  -- {* roots of proper subtrees, child nodes *}
  "R1 ≜ λ t. image ROOT (PSubtrees(t))"

 "Sigma ≜ λ S. {⋃ (⋃ S)} ∪ (⋃ S)"
```

Due to the lack of space, we do not state all the established results. We will give them on the fly when needed.

Trees and phylogenetic trees. Starting from hierarchies, we first define a tree as a hierarchy with its `ROOT` as the single maximal element :

```
"Tree ≜ {h ∈ Hierarchy. Maximal(h) = {ROOT(h)}}"
```

Then, we introduce phylogenetic trees as trees which nodes are either leaves or the union of all its subnodes:

```
"Phylo ≜
  {t ∈ Tree. ∀ n ∈ t. n ∈ Leaves(t) ∨ n = ⋃ PDown(t,n)}"
```

With respect to phylogenetic trees, we just mention the following equality that will allow us to say that the reconstruction can proceed starting from the leaves, while the statement of the reconstruction theorem is over the root. Actually, for a phylogenetic tree t, we have: $\mathrm{ROOT}(t) = \bigcup \mathrm{Leaves}(t)$. Moreover, we will rely on the following result about the union of phylogenetic trees:

```
lemma phylo_union:
 assumes t1: "t1 ∈ Phylo"
 assumes t2: "t2 ∈ Phylo"
 assumes u: "ROOT(t2) ∈ Leaves(t1)"
 shows "t1 ∪ t2 ∈ Phylo"
proof ... qed
```

Examples. The figure 1 illustrates the representation of phylogenetic trees. For instance, with respect to the previous definitions and the tree let `t2`, we have:

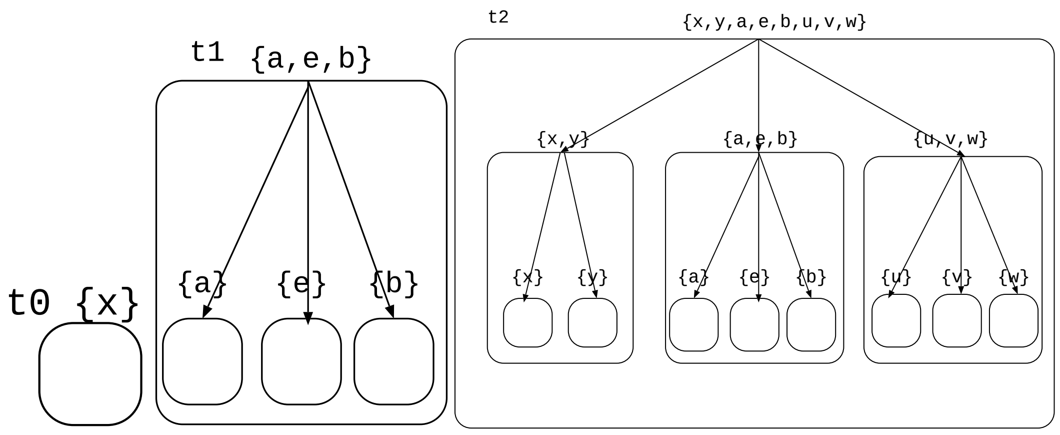

Fig. 1. phylogenetic trees

$$\begin{aligned}
t2 =& \{\{x,y,a,e,b,u,v,w\},\{x,y\},\{x\},\{y\}\\
&,\{a,e,b\},\{a\},\{e\},\{b\},\{u,v,w\},\{u\},\{v\},\{w\}\}\\
\text{Leaves}(t2) =& \{\{x\},\{y\},\{a\},\{e\},\{b\},\{u\},\{v\},\{w\}\}\\
\text{ROOT}(t2) =& \{x,y,a,e,b,u,v,w\}\\
\text{R1}(t2) =& \{\{x,y\},\{a,e,b\},\{u,v,w\}\}
\end{aligned}$$

The decomposition and induction theorems. In order to reason about phylogenetic trees, we first introduce a decomposition theorem: a tree is either a singleton containing its `ROOT`, or the sum (`Sigma`) of its proper subtrees.

```
theorem phylo_cases:
  assumes t: "t ∈ Phylo"
  shows "t = {ROOT(t)} ∨ t = Sigma (PSubtrees(t))"
proof ... qed
```

We state the induction theorem about phylogenetic trees as follows:

```
theorem phylo_induct:
  assumes b: "∀ e. P({e})"
  assumes r: "∀ T ∈ domSigma. (∀ t ∈ T. t ∈ Phylo ∧ P(t))
        ⇒ P(Sigma(T))"
  shows "∀ t ∈ Phylo. P(t)"
proof ... qed
```

where `domSigma` specifies the set of trees which can be "joined" to form a phylogenetic tree:

```
  "domSigma ≜
     {S. S ≠ ∅ ∧ finite(S) ∧ S ⊆ Tree
         ∧ (∀ t1 ∈ S. ∀ t2 ∈ S. t1 ≠ t2 ⇒⋃ t1 ∩ ⋃ t2 = ∅)
         ∧ PPartition(⋃ ⋃ S, image ROOT S)
     }"
```

2.3 Transformations

The basic property of the studied transformations is to preserve the underlying structure while transforming the nodes.

Hierarchy transformations and preservation theorem. First, we introduce general transformations which basic property is to preserve the cardinality of a set of nodes.

```
 "G_tr ≜ λ g. {tr. ∀ n1 ∈ g. ∀ n2 ∈ g.
                  (tr(n1) = tr(n2)) = (n1 = n2)}"
```

A hierarchy transformation is a general transformation which preserves the relations between the nodes of a hierarchy:

```
{* hierarchy transformations set*}
"H_tr ≜ λ h.
 {tr ∈ G_tr(h). (∀ n ∈ h. finite(n) ⇒ finite(tr(n)))
   ∧(∀ n ∈ h. n ≠ ∅ ⇒ tr(n) ≠ ∅)
   ∧(∀ n1 ∈ h. ∀ n2 ∈ h. n1 ⊆ n2 ⇒ tr(n1) ⊆ tr(n2))
   ∧(∀ n1 ∈ h. ∀ n2 ∈ h. n1 ∩ n2 =∅ ⇒ tr(n1) ∩ tr(n2) =∅)
 }"
```

A hierarchy is preserved by a hierarchy transformation:

```
theorem hierarchy_trans:
assumes t: "t ∈ Hierarchy"
assumes tr: "tr ∈ H_tr(t)"
shows "image tr t ∈ Hierarchy"
proof ... qed
```

A tree is also preserved by a hierarchy transformation.

Phylogenetic transformations and preservation theorem. Intuitively, when a phylogenetic transformation is applied to a non-leaf node, the decomposition into its descendant nodes is preserved. The characterizing property of a phylogenetic transformation is expressed as follows:

```
 "P_tr ≜ λ h. { tr ∈ H_tr(h). ∀ n ∈ h.
   n ∈ Leaves(h) ∨ tr(n) = ⋃ (image tr (R1(Down(h,n))))}"
```

Then, we state the preservation theorem:

```
theorem phylo_trans:
 assumes t: "t ∈ Phylo"
 assumes tr: "tr ∈ P_tr(t)"
 shows "image tr t ∈ Phylo"
proof ... qed
```

Example. A `Mutation` is a transformation that concerns the nodes up a graph node: `gp`, such “up” nodes contain `gp`, and a mutation is expressed as follows:

```
  "Mutation ≜ λ(gp,R).λ n. if gp ⊆ n then (n − gp) ∪ R else n"
```

We show that the `Mutation` transformation is a phylogenetic transformation:

```
theorem Mutation_P_tr:
  assumes h: "h ∈ Phylo"
  assumes g: "g ∈ Phylo"
  assumes pre: "PreGraft(h,gp,g)"
  shows "Mutation(gp,ROOT(g)) ∈ P_tr(h)"
proof ... qed
```

where PreGraft (We will use this predicate as the precondition of the Graft operation.) is defined as follows:

```
  "PreGraft ≜ λ(h,gp,g). h ∈ Hierarchy ∧ (((ROOT h) ∩ (ROOT g)) = ∅) ∧
                   gp ∈ Leaves(h) ∧ g ∈ Hierarchy ∧ g ≠ ∅"
```

2.4 Generative partitions and relations

One of our concerns is the reconstruction of a phylogenetic tree starting from the set of its leaves. The basic idea of such a reconstruction is to partition the leaves according to its direct proper subtrees and to apply recursively the reconstruction to each of the sets of the partition. These successive partitions define the sets which are generated by a generative partition.

Generative partitions. Given a phylogenetic tree h, GP is called a generative partition of h, if each node is either a leaf or partitioned according the direct sub-roots of n (Down(h,n) is the subtree of h which root is n).

```
"GenerativePartition≜ λ(h,GP). ∀ n ∈ h.
  GP(n) =(if n ∈ Leaves(h) then {n} else R1(Down(h,n)))"
```

Generative relations. Semantically, the generative relation is a symmetric relation of which the transitive closure is a generative partition. The motivation for introducing generative relations is to make local the reasoning about the of growth the tree and consequently easier than a global one. First, we define R2P which converts a relation to the partition function given by its reflexive and transitive closure: a node n is partitioned by the classes of the corresponding equivalence relation.

```
 "R2P(r) ≜ λ n. (n // ((r(n))^*))"

"GenerativeRelation ≜ λ (h,GR).
          (∀ n ∈ h. GR(n) ⊆ n × n ∧ sym(GR(n)))
      ∧ GenerativePartition(h, R2P(GR))"
```

2.5 The reconstruction function and theorem

We introduce the auxiliary function Reconstruct. Its definition is set up in order to be accepted as a well defined function by Isabelle: since it is a recursive function that is not primitive, we have to provide a measure that decreases at each call. The condition of the `if` expression ensures it. The `reconstruct` function is a curryfied version of `Reconstruct`.

```
  recdef Reconstruct "measure (λ (GP,s). card s)"
  "Reconstruct(GP,s) =
    (if (finite s) ∧ (∀ s' ∈ GP(s). s' ⊂ s) then
       Sigma (image (λ e. Reconstruct(GP,e)) (GP s))
     else {s})"
  (hints simp add: psubset_card_mono)

  "reconstruct(GP) ≜ λ s. Reconstruct(GP,s)"
```

The theorem characterizing the reconstruction is stated as follows:

```
theorem generative_partition_reconstruction:
shows
 "∀ GP. ∀ t ∈ Phylo. GenerativePartition(t,GP)
                         ⇒ (reconstruct(GP)(ROOT(t)) = t)"
proof ... qed
```

This theorem is established thanks to the induction theorem over phylogenetic trees (2.2).

2.6 Discussion

In this section, we discuss the definition of phylogenetic trees that has been elaborated. With respect to the structural point of view: a phylogenetic tree can be defined as either a singleton node or as the *Sigma* of its subtrees. Such a set based construction is not admitted by most of the type theory based logical frameworks [9, 1, 4, 13]. In fact, in such frameworks a tree is usually recursively defined through the *list* of its subtrees, or through a map of its subtrees from a given index type. We have tried to work with each of these representations. Their main drawback is to break the underlying natural confluence. For instance, with such representations, inserting a subtree after actually removing it, does not yield the original tree. Such a confluence is fundamental for establishing naturally our reconstruction result. Otherwise, we would have to introduce modulo relations in order to not distinguish between trees of which subtrees are identical but not in the same order.

3 The graft operation

In our setting, the graft operation models the growth of a tree. As its name suggests, the graft operation consists in grafting a tree at a given node. In this study, we consider a restricted version: grafting occurs at singleton nodes only.

3.1 Graft decomposition

Let h be a graph, gp a node of h where the graft should occur and g the graph to graft. We express the graft through two basic operations: first, h is transformed through a Mutation, second, g is grafted through the union (∪) operation. Such a decomposition is illustrated by figure 2. The Graft is expressed as follows:

```
"Graft ≜ λ(h,gp,g).(image (Mutation (gp,ROOT(g))) h) ∪ g"
```

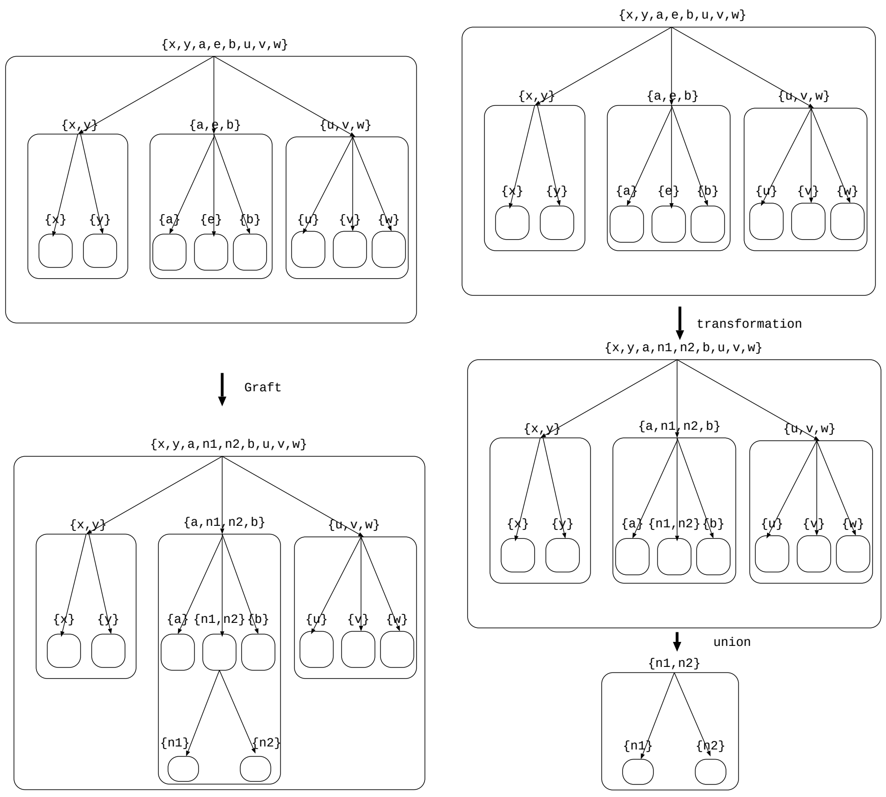

Fig. 2. Decomposition of a graft

We show that the grafted tree is also a phylogenetic tree. The proof is established thanks to the decomposition of the graft operation; we first establish that Mutation is a phylogenetic transformation, then thanks to the union theorem, g being phylogenetic, it follows that the grafted tree is phylogenetic.

```
theorem graft_phylo:
 assumes h: "h ∈ Phylo"
 assumes g: "g ∈ Phylo"
 assumes pre: "PreGraft(h,gp,g)"
 shows "Graft(gp,g)(h) ∈ Phylo"
proof ... qed
```

3.2 Reconstructing a graft through a generative partition

This section states a general result about the preservation of a generative partition GP. In fact, we have a precondition about the partitioning of the mutated nodes.

```
theorem generative_partition_graft_phylo:
  assumes h: "h ∈ Phylo"
  assumes g: "g ∈ Phylo"
  assumes pre: "PreGraft(h,gp,g)"
  assumes gp_h: "GenerativePartition(h,GP)"
  assumes gp_g: "GenerativePartition(g,GP)"
  assumes gp_tr:
    "∀ n ∈ h. GP(Mutation(gp,ROOT(g))(n)) =
       (if n ∈ Leaves(h) then {Mutation(gp,ROOT(g))(n)}
        else image (Mutation(gp,ROOT(g))) (GP(n)))"
  shows "GenerativePartition(Graft(h,gp,g),GP)"
proof ... qed
```

3.3 Reconstructing a graft through a generative relation

In the same way, a generative relation can be preserved while extending a tree through a graft. Thanks to this preservation: a tree, growing through graft operations, will always be reconstructible from its leaves through its invariant generative relation.

A simplified mutation: the basic update upd. For the purpose of our application, we consider the following node transformation:

```
"upd ≜ λ (l,N). λ S. if l ∈ S then S − {l} ∪ N else S"
```

Since we have: $\text{upd}(l, N) = \text{Mutation}(\{l\}, N)$, upd inherits the property of Mutation; then it is a phylogenetic transformation.

Moreover, in order to simplify the proof obligations for establishing that a generative relation is preserved after a graft, we have elaborated sufficient conditions that should be established by the update function. Due to the lack of space, we do not detail them.

We have established the preservation of the generative relation for the graft of a so called canonical tree which consists of a root and a set of leaves:

```
"Canonic ≜ λ N. {N} ∪ (⋃ e ∈ N. {{e}})"
```

We have the following invariant theorem establishing the preservation of a generative relation when grafting a canonic tree:

```
lemma generative_relation_graft_phylo:
 assumes t: "t ∈ Phylo"
 assumes up: "{l} ∈ t"
 assumes gp1: "GenerativeRelation(t,GR)"
 assumes gr_m: "∀ n. GR(n) ⊆ n × n ∧ sym(GR(n))"
 assumes terminal: "Terminal(t)"
 assumes N: "∀ n ∈ t. N ∩ n = ∅"
 assumes N_e: "N ≠ ∅ ∧ finite(N)"
 assumes gp2: "GenerativeRelation(Canonic(N),GR)"
 assumes gr_tr:
  "∀ n ∈ t. l ∈ n ⇒ r_upd(l,N)(n)(GR(n),GR(upd(l,N)(n)))"
  shows "GenerativeRelation(Graft({l},Canonic(N))(t),GR)"
proof ... qed
```

4 Application

As an application of phylogenetic trees, we consider distributed diffusing computations. In the initial state, one site (or process) multi-casts a message to a subset of other nodes. Then, all nodes share the same behavior: when a message is received, the receiver performs a computation step and, possibly, multi-casts a message to a subset of other nodes.

We are interested in the following problem: how to reconstruct the global history of such a computation, after its termination[3], from information gathered during the computation. For such a diffusing computation, the control flow is a tree in which the nodes are the computation steps and the edges are the message communications. Our algorithm consists in collecting an encoded representation of these leaves. From this leaves set, we apply the reconstruction algorithm based upon a generative relation defined on the computation as a phylogenetic tree.

4.1 Control tree encoding

We define an encoding for the control tree. The nodes generated during the computation (temporary leaves) are encoded as vectors. At each site, a local counter is incremented by $p-1$ each time a computation step multi-casts p messages. Thus, one[4] plus the sum of the local counters represents the number of the control tree leaves. Moreover, the value of the counter of site s is the maximum of the vectors component at the index s.

[3] Such a reconstruction is usually used for debugging purposes.

[4] We have to take into account the initial states where the counters are all null and the tree consists of one leaf node.

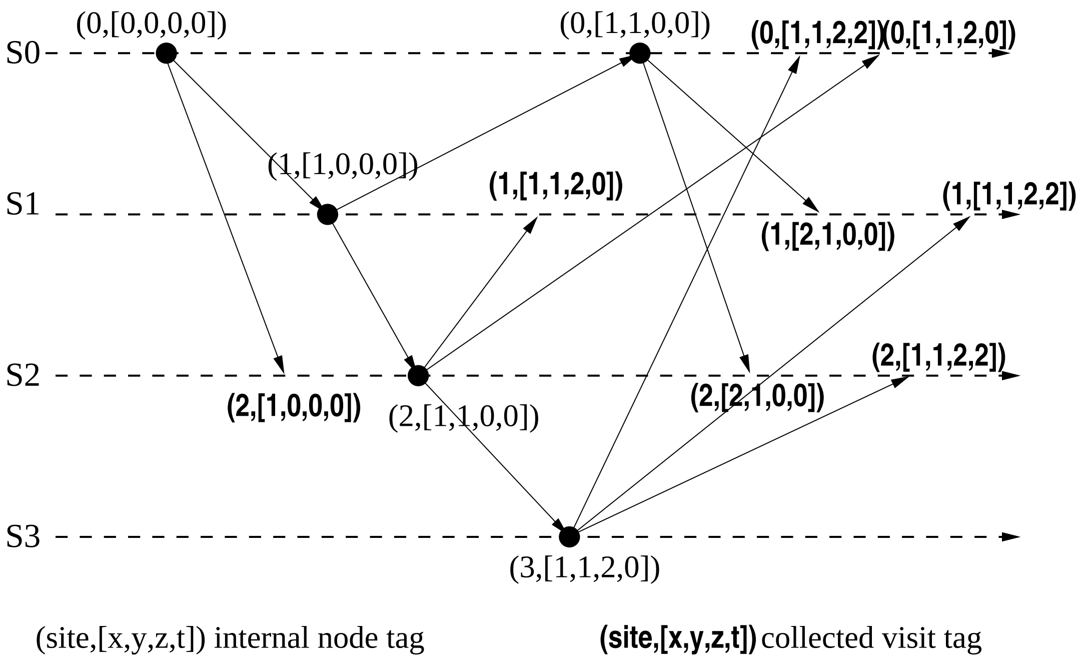

Fig. 3. Visit tags encoding

We associate a "visit tag" to each node. This tag is composed of the site of the node and a natural integers vector. This vector V has a size N, corresponding to the number of sites, and is assigned the local counter values of the sites it has visited. Figure 3 shows the tagging of the nodes of a diffusing computation with this encoding.

The state space of all the application is modeled by a global type `State`. It contains the fields related to the network, the local computations and the collector. The computation is concerned by the following fields:

- The field `lcounter` implements the local counter of each site;
- The field `collected` records the visit tags of the computation leaves.

Two transitions are considered and each of them is launched when a message is received:

- `ReceiveAndEnd` describes a computing step without further message multicast. In this case, the visit tag contained in the received message is sent to the collector;
- `ReceiveAndSplit` describes a computing step terminated by a message multicast. In this case, a new tag is created: it holds the destination site (d) and a vector which is identical to the tag vector of the splitting node ($m.V$), except for the splitting site (`self`) component, which gets the new local counter value $lcounter[self]$ assigned by this computation step. No message is sent to the collector.

With respect to phylogenetic trees, the diffusing computation is seen as a tree. A `ReceiveAndEnd` assigns to a node the definitive leaf status. While a `ReceiveAndSplit` extends a tree with new leaves. We interpret it as a `Graft` operation. Then, for validation purposes, we have an auxiliary variable `auxTree` for recording the growth of such a "superposed" tree: we prove that at termination, this auxiliary tree and the reconstructed tree are the same.

4.2 Termination detection and reconstruction

We introduce a collector process to gather vectors: a vector is sent to the collector when a process performs a computation step without multi-casting a new message. In such a case, this step generates a leaf with respect to the control flow of the computation. Then, with respect to phylogenetic trees, the collected tagged messages are in fact leaves of the phylogenetic tree superposed to the diffusing computation (and recorded in the auxiliary variable `auxTree`).

The reconstruction of the control tree can only start when the global computation is terminated. Several distributed algorithms can solve the termination problem, especially, thanks to a collector process[12]. However, the encoding itself provides a simple criterion for termination detection [8]: a computation is terminated when the number of collected leaves is equal to one plus the sum of the elements in the maximum of the collected visit vectors[5]:

$$\mid \text{collected} \mid = 1 + \sum_{s \in \text{Site}} \max_{v \in \mathit{collected}} v.V[s]$$

The generative relation for the diffusing computation as a phylogenetic tree is defined as follows:

```
"gr ≜ λ n. {(v1,v2). v1 ∈ n ∧ v2 ∈ n ∧
              (if V(v1) = min_on(n) ∨ V(v2) = min_on(n)
                 then (v1 = v2)
               else (∃ s. V(v1)(s) = V(v2)(s) ∧ s ≠ w(v1) ∧
                     s ≠ w(v2) ∧ V(v1)(s) ≠ min_on(n)(s)))
           }"
```

We derive the correctness of the reconstruction through the following invariant:

```
"ReconstructionInvariant ≜ λ st. auxTree(st) =
   reconstruct(R2P(gr))(collected(st) ∪ network(st))"
```

Then, when termination is reached, the network is empty, and the reconstruction applied to the collected messages gives the computation tree.

[5] |_| denotes the cardinality of _ .

5 Conclusion

In this paper, we have proposed a mechanization of phylogenetic trees. Starting from basic set theory, we have introduced phylogenetic trees through hierarchies and trees. Then, we have defined generic transformations. We note that sets based representations, although already suggested in the literature[10], are not widely used in computer science. To the best of our knowledge, the representation of a tree through the set of its leaves together with a generative partition or relation, as well as the study of dedicated transformations, are original. We have given a concrete example, where such notions have been applied to tree reconstruction and shown how such a reconstruction could be validated. It is interesting to remark that thanks to theorem proving techniques, such a validation was possible; actually we have considered an unknown number of nodes and unbounded natural vectors. Usual model checking techniques cannot handle such problems.

Most of our results have been proved formally within Isabelle. In fact, our trees are "unordered" trees. Such a data type could be considered as an inductive data type where `Sigma` would play the role of a constructor; however, due to the negative occurrence [6] most of the logical frameworks (HOL [9], Isabelle [13], PVS [4], Coq [1]) do not support such a definition schema. Vos and Swiestra [17] have studied restrictions for accepting inductive data types with negative occurrences; since our trees are finite, we could have reused their work. This work is not known to be available within the Isabelle framework. An alternative way would have been to introduce "unordered" trees through an equivalence relation [14] over ordered trees where subtrees are constructed with a list. It would be interesting to compare the subsequent developments of phylogenetic trees, generative relations and partitions.

With respect to the formalization of trees and biology related results, numerous works have been published. Among the more recent, we can cite [16] who consider the problem of tree inclusion in a categorical setting. [11] reviews basic network models for reasoning about biology; he notices that applications to biology of existing tools from algebra is just beginning. To the best of our knowledge, the mechanization of these works has not been considered yet. We think that our work could be reused as a starting point for establishing algorithms correctness but also for the correctness of their proposed proofs[7].

[6] The negative occurrence is due to the fact that the parameter of `Sigma`, considered as a constructor, is a set of trees.

[7] It is interesting to remark that the analysis of the algorithm of [5] is reported to be incorrect in [15].

References

1. B. Barras, S. Boutin, C. Cornes, J. Courant, J. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, August 1997. http://coq.inria.fr.
2. S. Berghofer and M. Wenzel. Inductive datatypes in HOL - lessons learned in formal-logic engineering. In Springer-Verlag, editor, *Theorem Proving in Higher Order Logics*, volume 1690, pages 19–36, 1999.
3. S. Bocker and A. W. Dress. A note on maximal hierarchies. *Advances in Mathematics*, (151):270–282, 2000.
4. S. Crow, S. Owre, J. Rushby, N. Shankar, and S. Mandayam. A Tutorial Introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton*, http://www.csl.sri.com/pvs, April 1995.
5. J. Culberson and P. Rudnicki. A fast algorithm for constructing trees from distance matrices. *Information Processing Letters*, 30(4):215–220, may 1989.
6. B. Davey and H. Priestley. *Introduction to Lattices and Order.* Cambridge Mathematical Textbooks. Cambridge University Press, 1990.
7. E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, 1989.
8. M. Filali, P. Mauran, G. Padiou, P. Quéinnec, and X. Thirioux. Refinement based validation of a distributed termination detection algorithm . In *FMPPTA'2000 , Cancun*, volume 1800 of *Lecture Notes in Computer Science*, pages 1027–1036. Springer-Verlag, may 2000.
9. M. Gordon and T. Melham. *Introduction to HOL.* Cambridge University Press, 1994.
10. D. E. Knuth. *The art of computer programming Fundamental algorithms*, volume 1. Addison-Wesley, 1969.
11. R. Laubenbacher. Algebraic models in Systems biology. In H. Anai and K. Horimoto, editors, *Algebraic Biology 2005 - Computer Algebra in Biology*, pages 33–40. Universal Academic press, Tokyo, Japan, 2005.
12. F. Mattern. Global quiescence detection based on credit distribution and re covery. *Information Processing Letters*, 30(4):195–200, Feb. 1989.
13. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic.* Number 2283 in Lecture Notes in Computer Science. Springer-Verlag, 2002.
14. L. C. Paulson. Defining functions on equivalence classes. *ACM Transactions on Computational Logic*, 7(4):658–675, 2006.
15. L. Reyzin and N. Srivastava. On the longest path algorithm for reconstructing trees from distance matrices. *Information Processing Letters*, 101(1):98–100, january 2007.
16. F. Rosello and G. Valiente. An algebraic view of the relation between largest common subtrees and smallest common supertrees. *Theoretical Computer Science*, (362):33–53, 2006.
17. T. E. Vos and S. D. Swierstra. Inductive data types with negative occurrences in HOL. In *Workshop on Thirty Five years of Automath, Edinburgh, UK*, 2002.
18. M. Wenzel and F. Wiedijk. A comparison of the mathematical proof languages Mizar and Isar. *Journal of Automated Reasoning*, 29:389–411, 2002.
19. M. M. Wenzel. *Isar – a generic interpretative approach to readable proof documents.* Number 1690 in Lecture Notes in Computer Science. Springer-Verlag, 1999.