HoTT formalisation in Coq: Dependency Graphs & ML4PG

Jónathan Heras and Ekaterina Komendantskaya

March 11, 2014

1 Introduction

This note is a response to Bas Spitter's email of 28 February 2014 about ML4PG:

We (Jason actually) are adding dependency graphs to our HoTT library: https://github.com/HoTT/HoTT/wiki

I seem to recall that finding the dependency graph was a main obstacle preventing machine learning to be used in Coq. However, you seem to have made progress on this. What tool are you using? https://anne.pacalet.fr/dev/dpdgraph/ ?

Or another tool? Would it be easy to use your ML4PG on the HoTT library?

This note gives explanation of how ML4PG can be used in the HoTT library and how ML4PG relates to the two kinds of Dependency graphs available in Coq.

2 Dependency Graphs in Coq:

There are two kinds of dependency graphs in Coq:

DG-1 *Graphs showing dependency of a Coq theorem to all the auxiliary results that were used to prove that theorem.* These graphs can be generated using the *dpdgraph* tool available at [8]. This tool can be also used to generate the dependencies of all the theorems of a library (see http://hott.github.io/HoTT/file-dep-graphs/HoTT.types.Paths.svg).

Example 1 *In the* Paths *file of the Hott library [10], we can find the following two theorems.*

```
Lemma dpath_path_l {A : Type} {x1 x2 y : A}
  (p : x1 = x2) (q : x1 = y) (r : x2 = y) :
  q = p @ r <~> transport (fun x => x = y) p q = r.
Proof.
  destruct p; simpl.
```

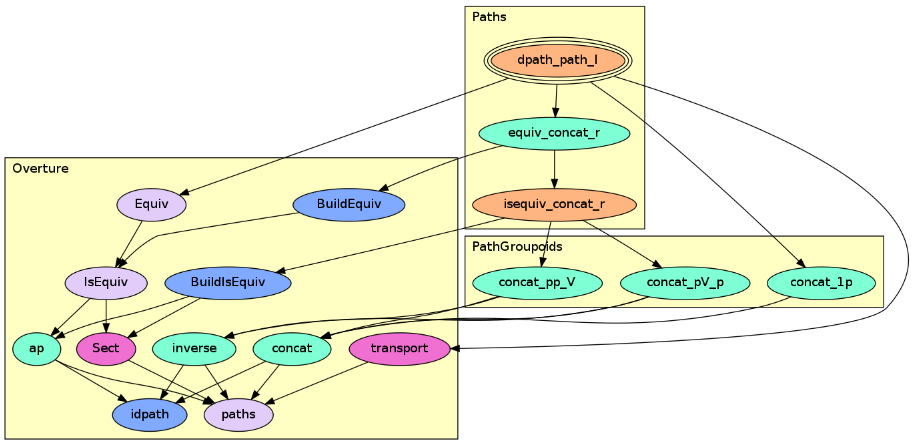


Figure 1: *Dependency graph for the theorem* `dpath_path_l` *included in the* `Paths` *library of HoTT.*

```
  exact (equiv_concat_r (concat_1p r) q).
Qed.

Lemma dpath_path_lr {A : Type} {x1 x2 : A}
  (p : x1 = x2) (q : x1 = x1) (r : x2 = x2) :
  q @ p = p @ r <~> transport (fun x => x = x) p q = r.
Proof.
  destruct p; simpl.
  refine (equiv_compose' (B := (q @ 1 = r)) _ _).
  exact (equiv_concat_l (concat_p1 q)^ r).
  exact (equiv_concat_r (concat_1p r) (q @ 1)).
Qed.
```

The dependency graphs, generated with the dpdgraph *tool, of these two lemmas are given in Figures 1 and 2, respectively.*

The graphs generated with the *dpdgraph* tool should be read as follows.

- A box represents a library file. E.g. Figure 1 shows that the proof of `dpath_path_l` involves results from the `Paths`, `PathGroupoids` and `Overture` libraries.
- An edge $n1 \rightarrow n2$ means that $n1$ uses $n2$. If $n1$ uses $n2$ and $n3$, and $n2$ uses $n3$; then, the only edges that appear in the graph are $n1 \rightarrow n2$ and $n2 \rightarrow n3$.
- A multi-circled node n indicates that n is not used (no predecessor in the graph) – if we generate the dependencies for a theorem t, the node for t will be the only multi-circled node.

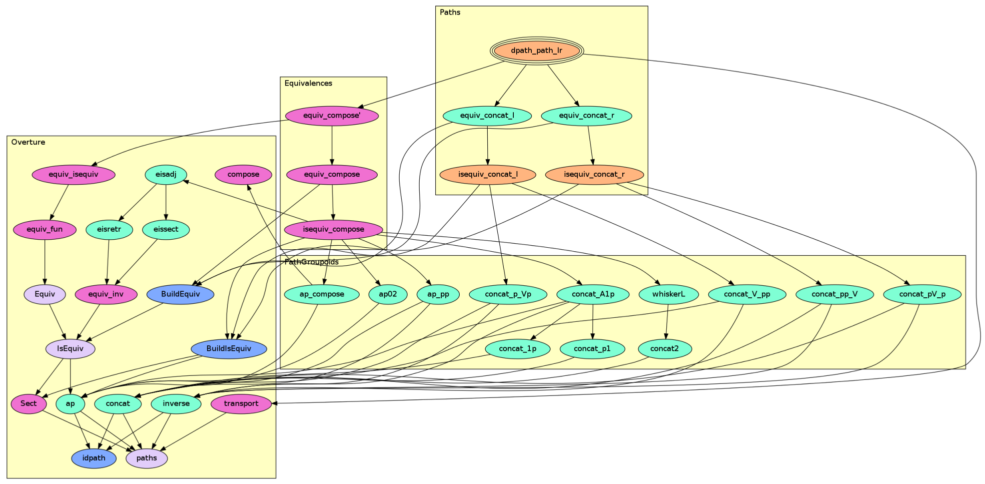


Figure 2: *Dependency graph for the theorem* ***`dpath_path_lr`*** *included in the* ***`Paths`*** *library of HoTT.*

- Orange nodes are theorems
- Green nodes are definitions.
- Light pink nodes are classes, or Inductive types.
- Blue nodes are constructors of the classes or Inductive types.
- Dark pink nodes are constructors inside the constructors of classes.

DG-2 *Graphs showing relations between libraries.* In these graphs, nodes represent library files, and an edge $l_1 \to l_2$ means that l_2 was imported in l_1. The graph is shown after a transitive reduction process. This kind of graphs can be generated using scripts from different projects like the MathComponents library [2] or the MathClasses library [9]. Some examples of this kind of dependency graphs are available at:

- `http://hott.github.io/HoTT/dependencies/HoTTCore.svg`.
- `http://ssr.msr-inria.inria.fr/~jenkins/current/index.html`.
- `http://www.unirioja.es/cu/cedomin/crship/BPL_general/toc.html`.

In addition to the two above mentioned methods to generate dependency graphs, a method to extract dependencies from Coq libraries was presented in [1]; however, to the best of our knowledge, this method does not generate any graphical representation for the captured dependencies.

3 ML4PG: how it works on the example of the HoTT library

ML4PG [4, 5, 7] is a machine-learning extension to the Proof General interface for Coq. It works in the background of Proof General capturing statistical features from Coq definitions and theorems. On user's demand, the statistical features are sent to a machine-learning tool to find families of similar definitions, theorems or proofs using clustering algorithms (a family of unsupervised machine-learning algorithms that groups together families of similar objects in *clusters*).

ML4PG implements two different methods to extract features from Coq terms and Coq proofs respectively, see [4] for a detailed explanation of the two feature-extraction algorithms.

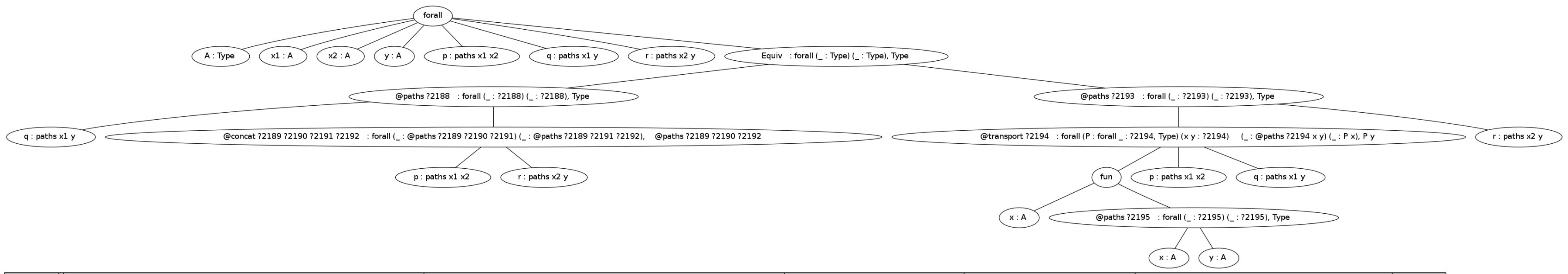
(*) The first feature-extraction algorithm collects statistical features from terms. In particular, given a term t, we generate its ML4PG-term-tree (cf. Figure 3) and from this term-tree we generate a feature-table that captures the structure of the term-tree and the terms and types at each node of the tree (cf. Figure 3).

Example 2 *Given the term of the statement of Lemma* `dpath_path_l`*, its ML4PG-term-tree and (a fragment of its) feature-table are given in Figure 3.*

The process of feature extraction mimics very closely the dependency graph for Coq terms. To populate the cells of the feature extraction table with feature values, the ML4PG tracks the dependencies of the auxiliary terms used within the term in question. For example, the term-tree for Lemma `dpath_path_l` (see Figure 3) contains a sub-expression `p : paths x1 x2`, and to extract a feature table for Lemma `dpath_path_l` one must compute the representative numeric value for `p : paths x1 x2`. To compute the latter, we must cluster `p : paths x1 x2` against other, previously defined terms and types, and they themselves must have feature tables, and so on, the process continues recursively until it reaches the basic library definitions, and includes dependencies of multiple theorems, definitions, lemmas etc. in the statistical pattern-recognition. In [4, 7], we call it *recurrent clustering.*

The construction of the term-feature-tables is a process that runs in the background of ML4PG. When the user asks ML4PG to show families of similar definitions or lemma statements, ML4PG processes the tables, sends them to a clustering algorithm implemented in Weka [3] (a machine-learning interface), and when the clustering algorithm is completed, ML4PG post-processes the results for showing them to the user.

Example 3 *ML4PG can be used to cluster the lemma statements of the Paths library of HoTT. The families of similar lemma statements are shown in Figure 4 — the current output of ML4PG is displayed in a buffer of Proof General using plain text with links to the lemmas (see Figure 5), the similarity graph of Figure 4 can be generated on user's demand. ML4PG differentiates without*


forall
A : Type
x1 : A
x2 : A
y : A
p : paths x1 x2
q : paths x1 y
r : paths x2 y
Equiv : forall (_ : Type) (_ : Type), Type
@paths ?2188 : forall (_ : ?2188) (_ : ?2188), Type
@paths ?2193 : forall (_ : ?2193) (_ : ?2193), Type
q : paths x1 y
@concat ?2189 ?2190 ?2191 ?2192 : forall (_ : @paths ?2189 ?2190 ?2191) (_ : @paths ?2189 ?2191 ?2192), @paths ?2189 ?2190 ?2192
@transport ?2194 : forall (P : forall _ : ?2194, Type) (x y : ?2194) (_ : @paths ?2194 x y) (_ : P x), P y
r : paths x2 y
p : paths x1 x2
r : paths x2 y
fun
p : paths x1 x2
q : paths x1 y
x : A
@paths ?2195 : forall (_ : ?2195) (_ : ?2195), Type
x : A
y : A

	level index 0	level index 1	level index 2	level index 3	level index 4	li 5
td0	([`forall`]$_{Gallina}$,-1,-1)	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)	...
td1	([`A`]$_{term}$,[`Type`]$_{type}$,0)	([`x1`]$_{term}$,[`A`]$_{type}$,0)	([`x2`]$_{term}$,[`A`]$_{type}$,0)	([`y`]$_{term}$,[`A`]$_{type}$,0)	([`p`]$_{term}$,[`paths x1 x2`]$_{type}$,0)	...
td2	([`paths ?X`]$_{term}$,[`?X -> ?X -> Type`]$_{type}$,7)	([`paths ?Y`]$_{term}$,[`?Y -> ?Y -> Type`]$_{type}$,7)	(0,0,0)	(0,0,0)	(0,0,0)	...

	li 4	level index 5	level index 6	level index 7
td0	...	(0,0,0)	(0,0,0)	(0,0,0)
td1	...	([`q`]$_{term}$,[`paths x1 y`]$_{type}$,0)	([`r`]$_{term}$,[`paths x2 y`]$_{type}$,0)	([`Equiv`]$_{term}$,[`Type -> Type -> Type`]$_{type}$,0)
td2	...	(0,0,0)	(0,0,0)	(0,0,0)

Figure 3: *ML4PG-term-tree and a fragment of the term-feature-table for the term of the statement of Lemma* ***`dpath_path_l`****.* The rows of the term-feature-table represent the depth of the tree, and the columns the index level. The element (i, j) of the table encodes the node of depth i and index-level j (e.g. the element $(1, 4)$ of the table encodes the node `p : paths x1 x2` of the term tree. Each node is represented by a triple that captures its term component, its type component and its parent node.

	tactics	n	*arg type*	*arg*	*symbols*	*goals*
g1	$([destruct]_{tac}, [simpl]_{tac}, 0, 0)$	2	$([$`paths x1 x2`$]_{type}$ $0, 0, 0)$	$(0, 0, 0, 0)$	$([$`<~>`$]_{term}, [=]_{term}, [=]_{term})$	1
g2	$([exact]_{tac}, 0, 0, 0)$	1	$([Prop]_{type}, 0, 0, 0)$	$([$`(equiv_concat_r (concat_1p r) q)`$]_{term}, 0, 0, 0)$	$([$`<~>`$]_{term}, [=]_{term}, [=]_{term})$	0

Table 1: *Feature table for the proof of Lemma* `dpath_path_l`*.*

any user interaction lemmas about transporting in path spaces (lemmas with the prefix `transport`*), the move family of equivalences (lemmas with the prefix* `isequiv_move`*), dependent paths (lemmas with the prefix* `dpath_path`*). Among these families, ML4PG also discovers lemma statements that are closer among them.*

Note that the dependencies of a term are captured with our feature-extraction method. Given a term t, the feature table of t stores the terms that form t; and, in turn, the dependencies of those terms are captured thanks to the recurrent clustering process.

The above mentioned method can cluster similar statements of all Coq terms, including lemmas and theorems. However, this method does not capture the interactive nature of Coq proofs. To solve this problem, we have a different method to extract features from Coq proofs, see [4].

(**) The method captures some important properties present in proof subgoals: tactics applied, type of the arguments of the tactics, lemmas applied, top symbols of the goal, and number of subgoals. These features are stored into numerical feature-tables that will be processed by machine-learning algorithms. To transform features like types, lemmas applied, or top symbols we again use recurrent term-clustering to assign similar values to objects that belong to the same cluster.

Example 4 *The feature table for the proof of Lemma* `dpath_path_l` *(cf. Example 1) is given in Table 1.*

The feature-tables obtained from the proofs of a library (or several libraries) are sent on user's demand to clustering algorithms to find families of similar proofs.

Example 5 *ML4PG can be used to cluster the proof of the Paths library of HoTT. The families of similar proofs are shown in Figure 6. We can distinguish 5 proof strategies in this library:*

- *A first group of proofs (*`dpath_path_lr`*,* `dpath_path_FlFr`*, . . .) is proved using first case analysis in one path, then a refinement step; and the proofs finished using two* `exact` *tactics.*
- *A second group of proofs (*`dpath_path_l`*,* `transport_paths_lr`*, . . .) is proved using first case analysis in one path and then the* `exact` *tactic (with different arguments).*

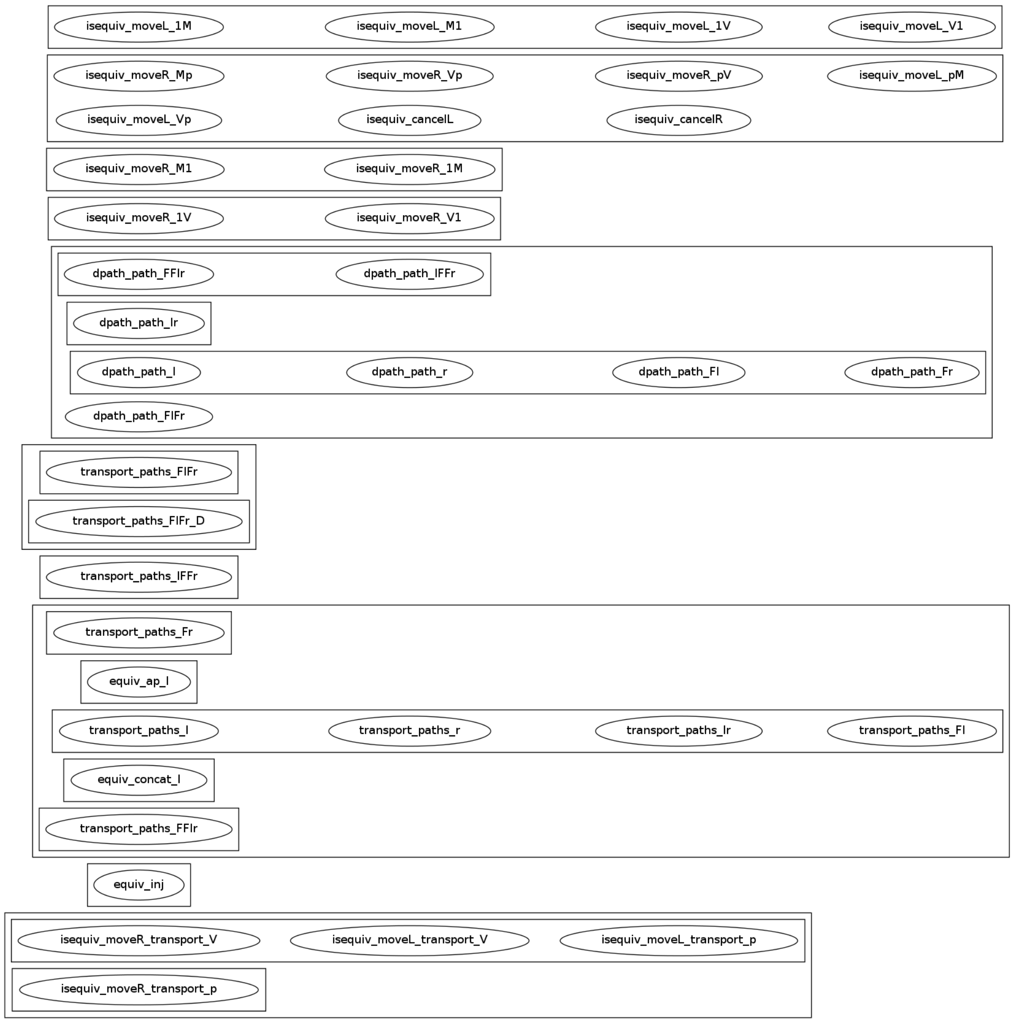
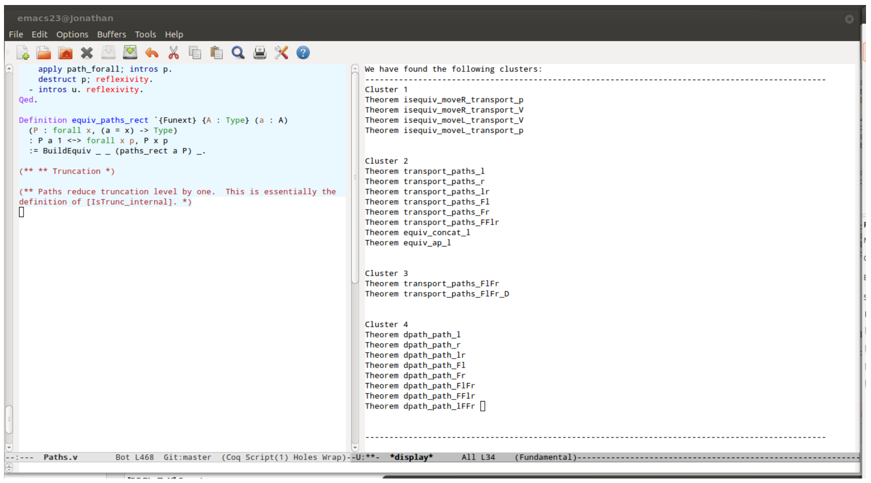


Figure 4: *Visualisation of the term-clusters of the Paths library.* The boxes of the diagram represent the different clusters, and the ellipses inside the boxes are the lemmas of each cluster. The precision of clusters can be adjusted using the granularity value – an ML4PG parameter (whose value is between 1 and 5) to adjust the precision of the clusters (1=low precision, 5=high precision) [7] – in particular, the clusters obtained with a low granularity value are usually split into smaller clusters when the granularity value is increased. We run twice the clustering algorithm to generate the visualisation of term-clusters: first with granularity value 3 and then with granularity value 5. If a cluster that appears with granularity value 3 is split in smaller clusters when increasing the granularity value, we represent this situation using boxes inside boxes (e.g. Lemmas `isequiv_moveR_transport_V`, `isequiv_moveL_transport_V`, `isequiv_moveL_transport_p` and `isequiv_moveR_transport_p` are grouped together with granularity 3; however, when increasing the granularity value, only the first three lemmas are grouped together, cf. the last cluster of the figure).



```
    apply path_forall; intros p.
    destruct p; reflexivity.
  - intros u. reflexivity.
Qed.

Definition equiv_paths_rect `{Funext} {A : Type} (a : A)
  (P : forall x, (a = x) -> Type)
  : P a 1 <~> forall x p, P x p
  := BuildEquiv _ _ (paths_rect a P) _.

(** ** Truncation *)

(** Paths reduce truncation level by one.  This is essentially the
definition of [IsTrunc_internal]. *)
```

```
We have found the following clusters:
-----------------------------------------------------------------------------------------------
Cluster 1
Theorem isequiv_moveR_transport_p
Theorem isequiv_moveR_transport_V
Theorem isequiv_moveL_transport_V
Theorem isequiv_moveL_transport_p


Cluster 2
Theorem transport_paths_l
Theorem transport_paths_r
Theorem transport_paths_lr
Theorem transport_paths_Fl
Theorem transport_paths_Fr
Theorem transport_paths_FFlr
Theorem equiv_concat_l
Theorem equiv_ap_l


Cluster 3
Theorem transport_paths_FlFr
Theorem transport_paths_FlFr_D


Cluster 4
Theorem dpath_path_l
Theorem dpath_path_r
Theorem dpath_path_lr
Theorem dpath_path_Fl
Theorem dpath_path_Fr
Theorem dpath_path_FlFr
Theorem dpath_path_FFlr
Theorem dpath_path_lFFr


-----------------------------------------------------------------------------------------------
```

```
--:---  Paths.v       Bot L468  Git:master  (Coq Script(1) Holes Wrap)--U:**-  *display*      All L34     (Fundamental)----
```

Figure 5: *ML4PG's output for term-clustering of lemmas of the Paths library using granularity value 3.* The Proof General screen has been split into two buffers. The left buffer shows the current proof-development (in this case, the `Paths` library of HoTT); and, the right buffer shows the families of similar lemma statements.

- *A third group of proofs (*`transport_paths_r`*,* `transport_paths_l`*, ...) is proved using case analysis in two paths and then reflexivity.*
- *A fourth group of proofs (*`isequiv_cancelR` *and* `isequiv_cancelL`*) is proved using case analysis in two paths, then simplification and finally the* `apply` *tactic.*
- *The last group of proofs (*`isequiv_moveR_1V`*,* `isequiv_moveR_1M`*, ...) is proved using case analysis in one path and then the proofs are finished using the* `apply` *tactic.*

As in the case of feature-extraction for terms, the feature-extraction for proofs capture the dependencies of proofs.

4 Relation between the Dependency Graphs and ML4PG

In this section, we explain the relation of ML4PG to dependency graphs. We divide this section into two parts – as there are two kinds of dependency graphs present in Coq.

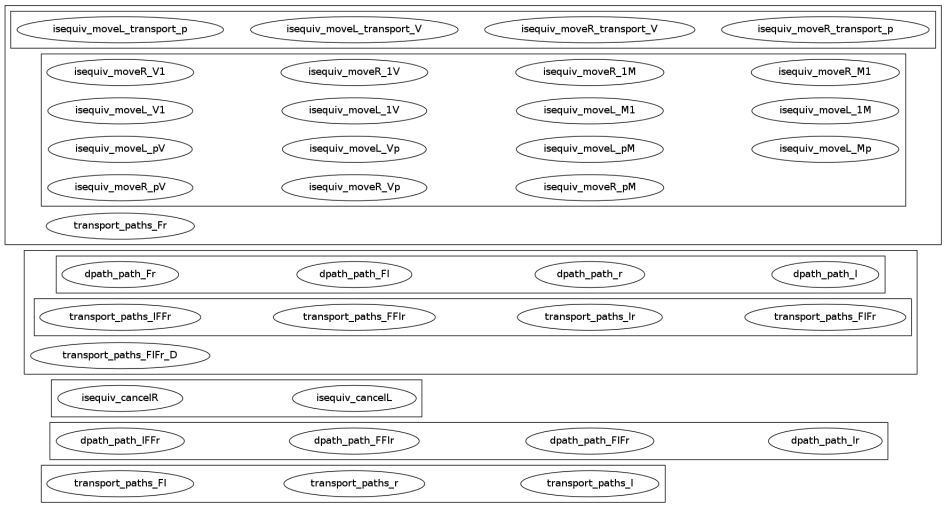


Figure 6: *Visualisation of the proof-clusters of the Paths library; based on analysis of their term structure and proofs.*

4.1 Dependency graphs for the single lemma or term

If we look into Figure 1 of the dependency graph for Lemma `dpath_path_l`, we will see that:

1. it is designed to work with proofs rather than object definitions;
2. it does not show the term structure *per se*, but only the dependencies between the auxiliary lemmas/constructors used;
3. it gives a complete information of all the results that were necessary to prove the lemma.
4. extracting this information does not require any statistical machine-learning (just careful tracing of dependencies...)
5. it may be hard to read, due to the presence of this complete information.
6. hence, statistical machine learning may be useful for data-mining the above information in order to discriminate unimportant features of the lemma, and highlight those that are important.

ML4PG essentially does what the last item says. The feature extraction of ML4PG known as "recurrent clustering" (see [4]) takes into account the information represented in the dependency graph as follows:

- ML4PG captures term and lemma dependencies recurrently, via the feature extraction which itself involves recurrent clustering of all previously defined terms;

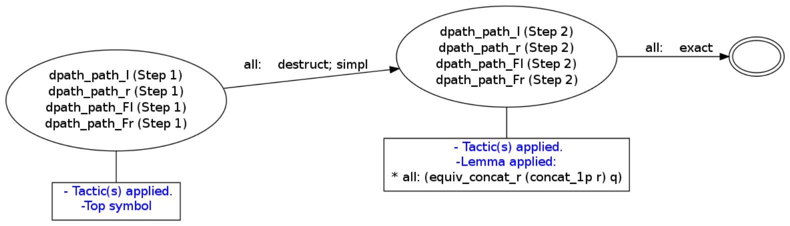


Figure 7: Automaton for one of the clusters of the Paths library using granularity value 5.

- ML4PG's proof clustering in particular is very close to dependency graphs for lemmas, in that both are capturing mutual or inductive dependencies of all lemmas needed to prove the given statement.

As a result, ML4PG can be seen as a post-processing tool for dependency graphs. Given a big graph as shown in Figure 1, what is the right way to discriminate its unimportant features? How to decide which of them are important or not? One of the ways to do this, is to determine that relatively to other lemmas in the library. This is achieved by applying clustering in ML4PG. Taking all of the above features into account, ML4PG can associate various object definitions and lemma statements and lemma proofs. See Figures 4 and 6.

On the basis of this association, we can essentially prune the statically generated dependency graphs to much smaller pictures, showing just some features of the given lemma and/or its proof, see Figures 7 and 8.

ML4PG's term-structure clustering goes beyond dependency graphs for terms, and carefully captures the tree term shape, structure, and dependency on other term- and type- structures present in the library (see Figure 3):

- The above approach makes sure that terms appearing in the lemma (as well as their structures) are taken into account.
- Dependency of auxiliary terms on other terms defined in Coq libraries is taken into account recurrently.
- Types used in the statements of lemmas are taken into account.
- If the types have been defined previously and depend on other types, all these hierarchies of dependencies are taken into account recurrently.
- Proof-clustering uses results of term-clustering via recurrent feature extraction.
- Unlike dependency graphs, tactics (not only auxiliary lemmas) used in the proof are taken into account.
- Type structures of tactic arguments are taken into account.

The work is under way to improve visualisation of the clustering results obtained by ML4PG. So far, we have two main directions:

1. Visualisation of the proof-flow.
2. Visualisation of the synthesized (generalized) term-tree.

The visualisation of the proof-flow summarising the important features of the proofs grouped into one cluster already exists, as a prototype, see Figures 7 and 8. The synthesis of important features from several term-trees is a harder task; the problem is discussed in detail in [6] – currently, ML4PG can only show the term tree associated with a statement (cf. Figure 3).

Note also that ML4PG, similarly to dependency graphs of type DG-1 takes into account mutual dependency of tactics and Coq object definitions, but for the post-processing stage, it makes differentiation between features of the object definitions and features of the proofs. Thus, the user can choose to see the analysis of similarities of object definitions (cf. Figure 3); or the analysis of similarities among proofs of the library (cf. Figures 7 and 8). This is made to facilitate proof development, as we assume ML4PG most common use would be to use clustering outputs as hints in new proofs; see [5] for a detailed analysis of this user scenario.

4.2 Dependency graphs showing relations of the (lemmas in the) library

ML4PG's clustering results can be compared to the dependency graphs of type DPG-2 as follows. Clustering results show similarity, rather than dependency graphs. Two proofs or terms may use completely different data structures (one e.g. nat – and another – list), and still they could be grouped together because they are structurally similar.

Example 6 *The lemma* `dpath_path_l` *(cf. Example 1) is clustered together with the following lemma:*

```
Lemma transport_paths_lr {A : Type} {x1 x2 : A} (p : x1 = x2)
  (q : x1 = x1) : transport (fun x => x = x) p q = p^ @ q @ p.
Proof.
  destruct p; simpl.
  exact ((concat_1p q)^ @ (concat_p1 (1 @ q))^).
Qed.
```

The statements of these two lemmas are quite different and term-clustering does not group them together (cf. Figure 4). However, the proofs of both results are really similar (see Figure 8) and for this reason they are clustered together regarding their proofs (cf. Figure 6).

On the contrary, there can be two lemmas/terms depending on the same term or lemma but being content-wise very different, and then ML4PG will not group them together:

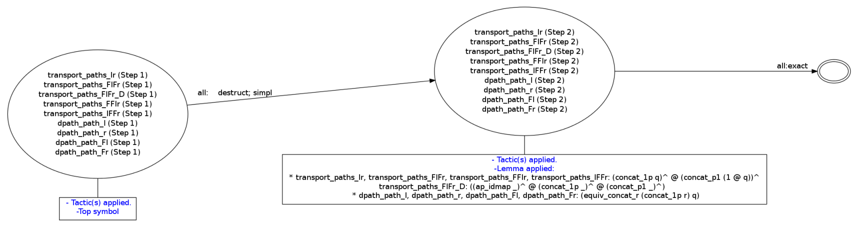

transport_paths_lr (Step 1)
transport_paths_FlFr (Step 1)
transport_paths_FlFr_D (Step 1)
transport_paths_FFlr (Step 1)
transport_paths_lFFr (Step 1)
dpath_path_l (Step 1)
dpath_path_r (Step 1)
dpath_path_Fl (Step 1)
dpath_path_Fr (Step 1)
- Tactic(s) applied.
-Top symbol
all: destruct; simpl
transport_paths_lr (Step 2)
transport_paths_FlFr (Step 2)
transport_paths_FlFr_D (Step 2)
transport_paths_FFlr (Step 2)
transport_paths_lFFr (Step 2)
dpath_path_l (Step 2)
dpath_path_r (Step 2)
dpath_path_Fl (Step 2)
dpath_path_Fr (Step 2)
- Tactic(s) applied.
-Lemma applied:
* transport_paths_lr, transport_paths_FlFr, transport_paths_FFlr, transport_paths_lFFr: (concat_1p q)^ @ (concat_p1 (1 @ q))^
transport_paths_FlFr_D: ((ap_idmap _)^ @ (concat_1p _)^ @ (concat_p1 _)^)
* dpath_path_l, dpath_path_r, dpath_path_Fl, dpath_path_Fr: (equiv_concat_r (concat_1p r) q)
all:exact

Figure 8: *Automaton for one of the clusters of the Paths library using granularity value 3, showing that the prrofs follows the same strategy (hence single, rather than mutiple, transitions in the automaton.)*

Example 7 *The statements of lemmas* `dpath_path_l` *and* `dpath_path_lr` *are similar and they are grouped in the same cluster using term-clustering (cf. Figure 4). However, their proofs are different and they are not grouped in the same cluster using proof-clustering (cf. Figure 6).*

Visualizing results that ML4PG obtains after recurrent library clustering, we get *similarity graphs* as opposed to *dependency graphs*, see Figures 4 and 6. In those figures, we see how Coq objects are related, based on their structures and recurrent relations to other Coq terms in the library.

5 Conclusions

ML4PG is a statistical machine-learning tool that could be used to post-process information available in the dependency graphs

- ML4PG is discriminative (unlike DG-1), and allows to abstract away from "unimportant features". Importance of features is decided after clustering makes statistical comparison of all objects (definitions and proofs) of the given libraries.
- ML4PG separates the output for term-structure and proof-structure, for readability and convenience, although it processes term structures, type structures and proof structures in their integrity during the recurrent library clustering.
- ML4PG can work beyond direct cross-referencing; hence can show similarities that are not captured by dependencies. In the opposite direction, it discriminates the information about objects that depend on same structures in case these objects are dissimilar on the "big picture".

References

[1] J. Alama, L. Mamane, and J. Urban. Dependencies in formal mathematics: Applications and extraction for coq and mizar. In *Intelligent Computer*

Mathematics, volume 7362 of *Lecture Notes in Computer Science*, pages 1–16. 2012.

[2] G. Gonthier et al. A Machine-Checked Proof of the Odd Order Theorem. In *4th Conference on Interactive Theorem Proving (ITP'13)*, volume 7998 of *LNCS*, pages 163–179, 2013.

[3] M. Hall et al. The WEKA Data Mining Software: An Update. *SIGKDD Explorations*, 11(1):10–18, 2009.

[4] J. Heras and E. Komendantskaya. Proof Pattern Search in Coq/SSReflect, 2014. http://arxiv.org/abs/1402.0081.

[5] J. Heras and E. Komendantskaya. Recycling Proof Patterns in Coq: Case Studies. *Journal Mathematics in Computer Science, accepted*, 2014.

[6] J. Heras, E. Komendantskaya, M. Johansson, and E. Maclean. Proof-Pattern Recognition and Lemma Discovery in ACL2. In *19th Logic for Programming Artificial Intelligence and Reasoning (LPAR-19)*, volume 8312 of *LNCS*, pages 389–406, 2013.

[7] E. Komendantskaya et al. Machine Learning for Proof General: interfacing interfaces. *Electronic Proceedings in Theoretical Computer Science*, 118:15–41, 2013.

[8] A. Pacalet and Y. Bertot. The dpdgraph tool, 2009–2013. https://anne.pacalet.fr/dev/dpdgraph/.

[9] B. Spitters and E. Van Der Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21:795–825, 8 2011.

[10] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* http://homotopytypetheory.org/book, Institute for Advanced Study, 2013. https://github.com/HoTT/HoTT/wiki.