Permutations in Coinductive Graph Representation

Célia Picard, Ralph Matthes

Permutations in Coinductive Graph Representation

Celia Picard and Ralph Matthes*

Institut de Recherche en Informatique de Toulouse (IRIT),
University of Toulouse and C.N.R.S., France

Abstract. In the proof assistant Coq, one can model certain classes of graphs by coinductive types. The coinductive aspects account for infinite navigability already in finite but cyclic graphs, as in rational trees. Coq's static checks exclude simple-minded definitions with lists of successors of a node. In previous work, we have shown how to mimic lists by a type of functions and built a Coq theory for such graphs. Naturally, these coinductive structures have to be compared by a bisimulation relation, and we defined it in a generic way.
However, there are many cases in which we would not like to distinguish between graphs that are constructed differently and that are thus not bisimilar, in particular if only the order of elements in the lists of successors is not the same. We offer a wider bisimulation relation that allows permutations. Technical problems arise with their specification since (1) elements have to be compared by a not necessarily decidable relation and (2) coinductive types are mixed with inductive ones. Still, a formal development has been carried out in Coq, by using its built-in language for proof automation.
Another extension of the original bisimulation relation based on cycle analysis provides indifference concerning the root node of the term graphs.

1 Introduction

In [16], we have developed a complete solution to overcome guardedness issues regarding programming and subsequent verification with embedded lists in a coinductive type in the proof assistant Coq [19], with the aim of graph representation (we represent single-rooted, connected graphs). In the present article, we present a more versatile bisimulation relation over those graphs than the one studied in [16] that naturally followed from the obtained coinductive representation of graphs.

First of all, we give a brief summary of the main notions introduced in [16] that will be needed for the rest of the article. Although all of this reports on formalization work that has been carried out in Coq, we refrained from displaying Coq syntax (the Gallina language) or even of its underlying type theory – the calculus of inductive and coinductive constructions (CIC). We try to adopt

* This work has been funded by the project CLIMT of the French Agence Nationale de la Recherche (ANR-11-BS02-016).

standard mathematical or standard type-theoretical notation as much as possible. In case of doubt what the exact correspondence is with the CIC/Coq, we invite the reader to refer to our full proofs [17].

1.1 Summary of the Notions Introduced in [16]

The goal we were aiming at was coinductive graph representation. We explained that we needed a structure that would mimic lists without being inductive (this was necessary because of Coq's guardedness condition (see [4] and [11]) which is based on a productivity criterion (see [7])). Therefore, we decided to use functions (this idea has also been mentioned by Chlipala in [5]). The domain of definition of those functions is a set of n elements, n being the length of the list we want to mimic. It is defined inductively through the two following constructors:

Definition 1 (*Fin*, Viewed Inductively).

$$\frac{n : \mathbb{N}}{\mathit{first}\ n : \mathit{Fin}(n+1)} \qquad \frac{n : \mathbb{N} \qquad i : \mathit{Fin}\ n}{\mathit{succ}\ i : \mathit{Fin}(n+1)}$$

This definition has also been used by Altenkirch in [1] and by McBride and McKinna in [12]. By construction, *Fin* n is a type with precisely n elements. It actually corresponds to the container view of lists [18].

Remark 1 (Notations). In the rest of the paper we will use the following notations:

- T, U for types and t (resp. u) for elements of type T (resp. U),
- n, m and k for elements of $\mathbb{N}$,
- l and q for lists and elements of *ilist* ("indexed lists"), to be defined below,
- f for functions,
- g for elements of *Graph*, to be defined below,
- R for binary (endo-)relations: a relation on T has type $T \to T \to \mathit{Prop}$, with $\to$ right-associative (as always in this paper) and *Prop* the Coq universe of propositions (in standard mathematics, one would just have the two truth values in *Prop*, and the informal semantics of relations is the standard one, but the application notation $R\,t_1\,t_2$ is used instead of $R(t_1, t_2)$ in standard mathematics),
- i for elements of *Fin* n,
- if a relation S depends on a relation argument R, we say that R is the base relation of S_R, and that S preserves equivalence if, for every equivalence relation R, S_R is an equivalence relation (likewise with reflexivity, symmetry and transitivity alone).

The type of functions that mimics lists of length n is defined as follows:

Definition 2. $\mathit{ilistn}\ T\ n := \mathit{Fin}\ n \to T$

However, this is not yet satisfactory as *ilistn* still has the length parameter n while lists do not – it is inherent to them. Therefore, we define the following structure consisting of the length and a function in the corresponding *ilistn*:

Definition 3. $ilist\ T := \Sigma n : \mathbb{N}.\ ilistn\ T\ n$

Σ denotes a strong sum type (of pairs where the type of the second component depends on the first component). Consequently, the elements of *ilist* T are written as pairs $\langle n, ln\rangle$ with ln of type $ilistn\ T\ n$. We call the associated projection functions lg and fct, i. e., $lg\ \langle n, ln\rangle = n$ and $fct\ \langle n, ln\rangle = ln$. One can show that *ilist* is in pointwise bijection with lists that are preloaded from the standard library in Coq (to do so, we use two conversion functions *ilist2list* and *list2ilist* and show that their compositions are pointwise equal to the identity – since Coq lacks functional extensionality, this does not imply equality with the identity).

Remark 2. In the container view, n corresponds to the shape of the container, and $Fin\ n$ to the type of positions (an element of $Fin\ n$ is a position).

We need to lift relations on T to relations on *ilist* T expressing that the base relation holds element-wise. In our intended use, the type T will be the coinductive type for graphs on which Leibniz equality – the propositional equality = of Coq – cannot be used. We define the following generic operation *ilist_rel*:

Definition 4 (*ilist_rel*).

$$\forall l_1\ l_2 : ilist\ T, ilist\_rel_R\ l_1\ l_2 \Leftrightarrow \exists h : lg\ l_1 = lg\ l_2, \\ \forall i : Fin(lg\ l_1), R\ (fct\ l_1\ i)\ (fct\ l_2\ i'_h)$$

where i'_h *is the result of converting* i *to type* $Fin(lg\ l_2)$ *using hypothesis* h.

This conversion is needed since the theoretical basis of Coq – the CIC – is an intensional type theory, where equal types cannot enter the type-checking process more deeply than equal terms, i. e., if a term t has type A and one can prove $A = B$, the term t does not have type B, but there is a construction taking t and yielding an inhabitant of B. For more details about the pattern matching feature that allows us to make this type conversion in Coq, see [19, Chap. 1.2.13 and 4.5.4]. In this paper, we do not expose the technical issues with type conversion that have to be mastered in the reasoning involving *ilist_rel*.

We define the type of graphs of interest in our paper through the constructor:

Definition 5 (*Graph*, Viewed Coinductively). $$\frac{t : T \qquad l : ilist(Graph\ T)}{mk\_Graph\ t\ l : Graph\ T}$$

Coinductive types in Coq are written with their constructors, just as for inductive types although categorical duality would suggest to define coinductive types by their destructors (projection functions). The constructor-based format is usually preferred for programming purposes. In order to distinguish the constructor-based view of coinductive definitions from inductive definitions, we put a double line in the inference rules. This notation is also used in [14]. Call *label* and *sons* the two projection functions on *Graph*, i. e., $label(mk\_Graph\ t\ l) = t$ and $sons(mk\_Graph\ t\ l) = l$. They correspond to the unfolding of the greatest fixed point and are more basic than the constructors in coalgebraic approaches.

Had we used lists instead of *ilist*, we would have had an embedded inductive type in a coinductive type, with all the problems of guardedness as described in

[16]. However, another approach to solve the same kind of problem in Coq has been proposed by Dams in [8], where the embedded inductive type is even extended to a coinductive one. This solution seems to be quite heavy to manipulate as there are a lot of proofs to carry out. Niqui in [15] and Bertot and Komendantskaya in [4] propose solutions also for a coinductive embedded type (and not an inductive one). The latter solution is quite close to ours, but on streams instead of lists, and hence without the need to restrict to a finite domain. For the finite domain, we chose *Fin n*, and thus "objects" n enter types and we are forced to use techniques of dependently-typed programming. Moreover, we have to quantify existentially over n, and this constructively, as is done with the Σ-type.

The same kind of issues also appears in other proof assistants, e. g., in Agda[1], even though it has recently seen quite some progress in its termination checker. In [9] Danielsson describes an experimental solution to solve them (see also the extended case study with Altenkirch in [10]). This extension by datatypes that may have both inductive and coinductive constructors is still experimental. In Coq, this is just not available.

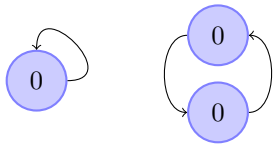
We lift relations R on T to relations Geq_R on $Graph\ T$ in a generic manner:

Definition 6 (*Geq*, Viewed Coinductively).

$$\frac{g_1\ g_2 : Graph\ T \qquad R\ (label\ g_1)\ (label\ g_2) \qquad ilist\_rel_{Geq_R}\ (sons\ g_1)\ (sons\ g_2)}{Geq_R\ g_1\ g_2}$$

It becomes apparent that $ilist\_rel_R$ had to be defined for arbitrary relations, not only Leibniz equality. Preservation of equivalence by *ilist_rel* cannot suffice to prove that *Geq* preserves equivalence, but still, preservation holds [16].

Remark 3. The two graphs on the right are equivalent through *Geq*. If one wanted to differentiate between these two structures, one could use the type $\{A, B\} \times \mathbb{N}$ instead of $\mathbb{N}$ and thus give an identity to the nodes.



1.2 Need for a More Liberal Relation on *Graph* - Content Overview

With *Geq*, we are forced to give an order to the nodes, horizontally and vertically, while in a classical set-theoretic graph representation, there is no such order. We aim at obtaining the more liberal equivalence of the classical representation on our constructive representation, e. g., we want the graphs of Fig. 1 or of Fig. 2 to be considered equivalent. However, with respect to *Geq*, they are not (here and in all further examples, we implicitly use Leibniz equality $=$ as the base relation). Indeed, their nodes are not in the same order. In the case of Fig. 1, the children of the root are inverted in the two graphs (different "horizontal" order). In Fig. 2 the second graph has been turned by 180° (different "vertical" order). The new equivalence relation should solve the two cases illustrated by Fig. 1 and Fig. 2. The first one corresponds to a permutation between the children of an element

[1] `http://wiki.portal.chalmers.se/agda/pmwiki.php`

of *Graph*. Thus, this corresponds also to a new equivalence relation on *ilist* (that indicates that two elements of *ilist* are permutations of each other). The second one corresponds to a change in the point of view of the observation of the graph.

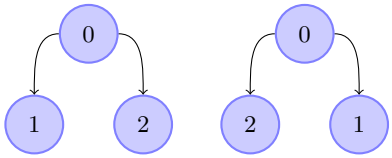


Fig. 1. Different order in children

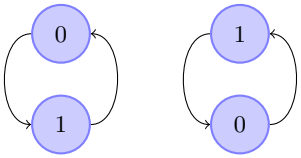


Fig. 2. Different roots

This paper presents the development of these new relations. In Sect. 2 we present and compare various ways to represent permutations on *ilist*. Sect. 3 uses our favourite one to define relation *GPerm* on *Graph* to capture permutations of children (see Fig. 1), with an extended discussion about how to interpret the coinductively defined relation. Sect. 4 shows a characterization of *GPerm* by a sequence of relations corresponding to increasingly deep observations of the graphs involved. Finally, in Sect. 5, we present a relation on *Graph* that solves the two issues mentioned above (corresponding to the features of Fig. 1 and Fig. 2).

All the ideas presented in the body of this paper are original with respect to [16]. All of the contents of this article has been formally modeled and proved in Coq (version 8.3). The development is available in [17].

2 Capturing Permutations on *ilist*

A first and standard solution would be to check that the elements of *ilist* we are comparing contain the same number of occurrences of each element. However, counting occurrences needs decidability of R, i. e., $\forall t_1\ t_2,\ (R\ t_1\ t_2)\ \vee\ \neg(R\ t_1\ t_2)$, with a strongly constructive form $\vee$ of disjunction that comes with evidence for the case that has been proven (in Coq, this is `sumbool`, which belongs to the universe `Set` of computationally relevant types).

The relation that ensures the same numbers of occurrences of all elements cannot replace *ilist_rel* in the definition of *Geq* as its index relation is the one that we would be defining, and for which decidability cannot be already known. Anyway, *a posteriori*, decidability for *Geq* cannot be expected.

We could turn to an inductive characterization of whether an element of *ilist* has a certain number of occurrences. Instead, we preferred to give a direct inductive characterization of when permutations exist. In the next section we will give two equivalent such characterizations, and in Sect. 2.2 we will enrich with "witnesses" one of these. In Sect. 2.3, we give a declarative notion that does not depend on induction. In Sect. 2.4, we give a formal comparison of these notions, while we finish with a more informal discussion in Sect. 2.5.

2.1 Inductive Definitions of Permutations on *ilist*

We now explore notions of permutations that do not depend on decidability of the base relation. Ideally, we want to remain in the realm of logical/declarative approaches. Instead of saying what a permutation is, we will specify when it exists. We do this constructively – by an inductive generation process. Read backwards, we basically remove equivalent elements one by one.
To do so, we need a function that removes an element from an *ilist*. The idea is that it only keeps the elements "before" and "after" the one we want to remove. We call this function *remEl* but we do not give a formal definition here because it is rather technical and we want to keep the article free from difficulties of dependently-typed programming. It has type $remEl\ (l : ilist\ T) : Fin\ (lg\ l) \to ilist\ T$. It is characterized by the following assertions:

$$lg(remEl\ \langle n+1, ln\rangle\ i) = n \tag{1}$$
$$i' <_{Fin} i \Rightarrow fct(remEl\ \langle n+1, ln\rangle\ i)\ i' = ln(weakFin\ i') \tag{2}$$
$$i \leq_{Fin} i' \Rightarrow fct(remEl\ \langle n+1, ln\rangle\ i)\ i' = ln(succ\ i') \tag{3}$$

where $weakFin : \forall n,\ Fin\ n \to Fin(n+1)$ is recursively defined by:
$weakFin(first\ n) := first(n+1)$ and $weakFin(succ\ i') := succ(weakFin\ i')$
This function only increases the type indices in its argument. Logically speaking, this is a kind of weakening. Please, note that 2 and 3 only type well modulo 1.

Remark 4 (Order relation on Fin). We informally use $<_{Fin}$, $\leq_{Fin}$ and $=_{Fin}$ to represent the order relation on *Fin*. Basically, we order the elements of *Fin* by the number of *succ* in their definition, disregarding the type indices, e. g.,
$first\ 2 <_{Fin} succ(first\ 1) =_{Fin} succ(first\ 2)$. Thus, elements of different types are put into relation. One can show $\forall i, weakFin\ i =_{Fin} i$.

The permutation relation can be defined using any one of the two following definitions (we will prove that they are equivalent):

Definition 7 (*iperm*, Viewed Inductively).

$$\forall l_1\ l_2,\ iperm_R\ l_1\ l_2 \Leftrightarrow \begin{cases} lg\ l_1 = lg\ l_2 = 0 \quad \text{or} \\ \exists i_1\ i_2, R\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2) \\ \qquad \wedge\ iperm_R\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2) \end{cases}$$

Definition 8 (*iperm′*, Viewed Inductively).

$$\forall l_1\ l_2,\ iperm'_R\ l_1\ l_2 \Leftrightarrow lg\ l_1 = lg\ l_2\ \wedge\ \big(\forall i_1 \exists i_2, R\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2) \wedge\ iperm'_R\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2)\big)$$

Note that, *iperm* gives us one pair of equivalent elements that we can remove at each "level", while *iperm′* says that at each level all elements of l_1 have an equivalent one in l_2 (and this recursively). Roughly, *iperm* can be seen as a particular case of *iperm′* since, if $lg\ l_1 > 0$, there is the canonical choice of i_1 to be the first element of its domain.

Before proving the equivalence of these definitions, we observe that the following assertion: $\forall l_1 l_2, iperm_R\ l_1\ l_2 \Rightarrow lg\ l_1 = lg\ l_2$ holds, to be proven by induction.

Theorem 1. $\forall l_1\ l_2, iperm_R\ l_1\ l_2 \Leftrightarrow iperm'_R\ l_1\ l_2$

Proof.

[$iperm' \Rightarrow iperm$] This was already discussed. Formally, it is done by induction on $iperm'$.

[$iperm \Rightarrow iperm'$] To prove this, we reason by induction on $lg\ l_1$.

- If $lg\ l_1 = 0$ then $lg\ l_2 = 0$ (using the above observation), hence $iperm'$is trivially true since there is no $i_1 : Fin\ 0$.
- If l_1 and l_2 are not empty, the proof is much harder. As we already know that $lg\ l_1 = lg\ l_2$, what we have to prove is:

$$iperm_R\ l_1\ l_2 \Rightarrow \big(\forall i_1 \exists i_2, R\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2) \wedge iperm'_R\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2)\big)$$

  Its proof is quite subtle because it requires a precise case analysis on the index i_1, which is complicated in this dependently-typed setting. A crucial role is played by interchange lemmas for two subsequent applications of *remEl*. For its proof, one has to think of a complete unfolding of *iperm* until the base case is reached. Then, one may permute the order of the obtained pairs in the relation R, just as in the classical mathematical theory of permutations. □

The usefulness of having these two equivalent definitions is that they are quite different in structure. Therefore, some properties are easier to show on one or the other. In particular, this is the case for the proof that *iperm* and $iperm'$ preserve equivalence: symmetry is easier with Definition 7 while transitivity is easier with Definition 8 (for i_1 in l_1, one gets i_2 in l_2 and then i_3 in l_3, while Definition 7 yields two possibly different indices in l_2). These proofs are not detailed here.

We have also proved that *iperm* (resp. $iperm'$) preserves decidability (see the definition of decidability in the introduction of Sect. 2) :

Lemma 1. *If R is decidable, then $iperm_R$ (resp. $iperm'_R$) is decidable.*

Proof. One possibility is a proof by induction using Definition 8 (and Theorem 1).

2.2 A Richer Notion of Inductive Permutations

However, sometimes, knowing that a permutation exists will not be enough, for example when considering intersection of base relations, as in Lemma 3 below. Definitions 7 and 8 do not allow us to manipulate permutations directly since only their existence is specified. Thus, we have to enrich our definition with a notion of skeleton that witnesses the existence of a permutation. The skeleton records the choices for the existentially quantified variables according to Definition 7. Hence, each skeleton consists of a tuple of pairs of elements of *Fin*. In each pair, the first element of *Fin* corresponds to an index in the first *ilist*, and the second one to the corresponding element in the second *ilist*. These skeletons form types *skel_type* n, defined as $\mathbb{N}$-indexed family of types:

Definition 9 (*skel_type*, Defined Recursively).

$$\begin{aligned} &\mathit{skel\_type}\ 0 := \mathit{unit} \\ &\mathit{skel\_type}\ (n+1) := (\mathit{Fin}(n+1) \times \mathit{Fin}(n+1)) \times \mathit{skel\_type}\ n \end{aligned}$$

Here, *unit* is a one-element type (one might take $\mathit{unit} := \mathit{Fin}\ 1$, but *unit* has nothing to do with indices in *ilist*).

Now, we can define a new notion of permutation (let's call it *iperm_skel*) using this notion of skeletons. Definition 8 would certainly not fit well with it, but Definition 7 does. To be able to use *skel_type*, we need one natural number as index for both components of the pair, in other words, we need to know that $lg(\mathit{sons}\ g_1) = lg(\mathit{sons}\ g_2)$. Therefore, we have chosen to add this as the hypothesis H_{lg} for *iperm_skel*. However, this is reasonable as we know that it is a direct consequence of *iperm* (as observed right after the definition of iperm'). We define *iperm_skel* in lockstep with Definition 7, but with the extra data:

Definition 10 (*iperm_skel*, Viewed Inductively).

$$\forall l_1\ l_2\ H_{lgti}\ s,\ \mathit{iperm\_skel}_R\ l_1\ l_2\ H_{lgti}\ s \Leftrightarrow \begin{cases} lg\ l_1 = 0 \quad \text{or} \\ \exists i_1\ i_2\ s', R\ (\mathit{fct}\ l_1\ i_1)\ (\mathit{fct}\ l_2\ i_2)\ \wedge\ \text{“}s = ((i_1, i_2), s')\text{”}\ \wedge \\ \mathit{iperm\_skel}_R\ (\mathit{remEl}\ l_1\ i_1)\ (\mathit{remEl}\ l_2\ i_2)\ H'_{lgti}\ s' \end{cases}$$

where H_{lg} is the aforementioned variable assuming $lg\ l_1 = lg\ l_2$ and is used to build a proof H'_{lgti} of $lgti(\mathit{remEl}\ l_1\ i_1) = lgti(\mathit{remEl}\ l_2\ i_2)$, s is of type $\mathit{skel\_type}(lg\ l_1)$ and “$s = ((i_1, i_2), s')$” is only correct up to several type conversions that are not detailed here. Notice that i_1 is of type $\mathit{Fin}(lg\ l_1)$, i_2 is of type $\mathit{Fin}(lg\ l_2)$ and s' is of type $\mathit{skel\_type}(lg(\mathit{remEl}\ l_1\ i_1))$.

We can show that *iperm* and *iperm_skel* are equivalent (proof not given here):

Lemma 2 ($\mathit{iperm} \Leftrightarrow \mathit{iperm\_skel}$).

$$\forall l_1\ l_2\ H_{lg}, \mathit{iperm}_R\ l_1\ l_2 \Leftrightarrow \exists s, \mathit{iperm\_skel}_R\ l_1\ l_2\ H_{lg}\ s$$

We can also show that *iperm_skel* is monotone in its relation argument (this was also true for *iperm*). Finally, we also obtain a lemma about relation intersection for *iperm_skel* given a specific skeleton:

Lemma 3 (Intersections in the Relation Argument of *iperm_skel*). *For fixed l_1, l_2, H_{lg} and skeleton s, $\mathit{iperm\_skel}_R\ l_1\ l_2\ H_{lgti}\ s$ commutes with arbitrary intersections of non-empty sets of relations R. In particular, if for all n $\mathit{iperm\_skel}_{R_n}\ l_1\ l_2\ H_{lgti}\ s$, then $\mathit{iperm\_skel}_{\cap_n R_n}\ l_1\ l_2\ H_{lgti}\ s$.*

Note that, without the skeleton – for *iperm* – this cannot be expected. This lemma will be necessary for the proof of Lemma 10 (see Sect. 4.2).

2.3 Permutations on *ilist* with Bijective Functions

The previous solution is constructive and does not require decidability but remains inductive. Hence, when we mix it with a coinductive definition on *Graph* (when defining an equivalence relation on *Graph*, for instance), we will have the same kind of problems as the ones we had when mixing lists with *Graph* (even though, as we will see in Sect. 3, since we are in the universe of propositions – not of sets, we have more ways to overcome the guardedness restrictions). Thus, we present here a last solution that does not require decidability and that is not inductive. It is based on an idea similar to the one we used for *ilist* as we use functions. This solution says that two elements of *ilist* are permutations of each other if there is a bijective function from the indices of the first one to the indices of the second one and when each pair of indices in relation points to equivalent elements.

First of all, we decide to express that a function is bijective by the following definition (where one has to give explicitly the inverse of the function) :

Definition 11 (*bij*).

$$\forall (f : T \to U)\ (g : U \to T), bij\ f\ g \Leftrightarrow (\forall t,\ g(f\ t) = t) \wedge (\forall u,\ f(g\ u) = u)$$

Now we can define the permutation relation using *bij*:

Definition 12 (*ipermb*).

$$\forall l_1\ l_2, ipermb_R\ l_1\ l_2 \Leftrightarrow \exists f\ g, bij\ f\ g\ \wedge\ (\forall i, R\ (fct\ l_1\ i)\ (fct\ l_2\ (f\ i)))$$

Remark 5. Note that *ipermb* has exactly the same logical structure as *ilist_rel*. Actually, *ilist_rel* can be seen as a special case of *ipermb*.

One easily shows that *ipermb* preserves equivalence, that it is monotone in its base relation and the following minimal requirement for a definition of permutation:

Lemma 4. $\forall l_1\ l_2, ipermb_R\ l_1\ l_2 \Rightarrow lg\ l_1 = lg\ l_2$

2.4 Equivalence Proofs Between the Definitions

In order to validate Definitions 7 and 12, we show that they are equivalent and that they are equivalent to a permutation relation on lists given by Contejean [6] (and thanks to Theorem 1 this will also validate Definition 8).

Equivalence between *iperm* and *ipermb* These definitions have the same coverage, in particular, they do not require decidability of R, and they achieve the same objective. We will show here that they are equivalent.

Theorem 2. $\forall l_1\ l_2, iperm_R\ l_1\ l_2 \Leftrightarrow ipermb_R\ l_1\ l_2$

Proof.

[**Direction** $\Rightarrow$] Actually here, what we really do is to prove $ipermb_R\ l_1\ l_2$ from $iperm\_skel_R\ l_1\ l_2$ (thanks to Lemma 2). We obtain the two functions needed for *ipermb* from the skeleton provided by *iperm_skel*. Then we finish the proof (for the $\forall i, R\ (fct\ l_1\ i)\ (fct\ l_2\ (f\ i))$ part) by induction on $lg\ l_1$.

[**Direction** $\Leftarrow$] Here, the proof is done by a simple induction on $lg\ l_1$. The main difficulty is that we have to recalculate all the indices in the functions for the new elements of *ilist* (where one element has been removed) in order to use the induction hypothesis.

Equivalence with Contejean In [6], Contejean treats the same problem as ours but on lists. She bases her solution on the classical one presented in [20] but adds a base relation. It is defined as follows, using @ for list concatenation:

Definition 13 (*permcont*, viewed inductively).

$$\forall R\ l_1\ l_2, permcont_R\ l_1\ l_2 \\ \Leftrightarrow \begin{cases} or & l_1 = l_2 = [] \\ & \exists a\ b\ l\ l'_1\ l'_2,\ l_1 = a :: l'_1 \wedge l_2 = l@b :: l'_2 \wedge R\ a\ b \wedge permcont_R\ l'_1\ (l@l'_2) \end{cases}$$

Remark 6. This definition is close to the spirit of Definition 7 but not with built-in symmetry since always the first element of l_1 is removed. Note that access to b in $l@b :: l'_2$ is not a native operation of lists while it is in the spirit of *ilist*.

In order to validate the definitions we proposed here, we are going to show that *iperm* is equivalent to *permcont* (and then, by Theorem 2, *ipermb* will also be equivalent). As *permcont* applies to lists, we will need the conversion functions *ilist2list* and *list2ilist* (see Sect. 1.1 right after Definition 3).

Theorem 3. $\forall l_1\ l_2, permcont_R\ l_1\ l_2 \Rightarrow iperm_R\ (list2ilist\ l_1)\ (list2ilist\ l_2)$

Proof. We work by induction on $permcont_R\ l_1\ l_2$, with trivial base case. The inductive case gives us a and b that we remove from l_1 and l_2 using Definition 7.

We also show the other direction (this time using *ilist2list*) :

Theorem 4. $\forall l_1\ l_2, iperm'_R\ l_1\ l_2 \Rightarrow permcont_R\ (ilist2list\ l_1)\ (ilist2list\ l_2)$

Proof. We work by induction on $iperm'_R\ l_1\ l_2$. The main technical difficulty here is that we need to obtain elements of the form $a :: l$ and $l'_1@b :: l'_2$. The first one is easy (directly given by Definition 8 – Definition 7 would not be enough here) but for the second one we will need to manipulate the "right part" and the "left part" of an *ilist*, which is not so trivial.

2.5 Comparison of the Obtained Notions

We have presented with details four different definitions of permutations on *ilist*. All of them have been shown extensionally (i. e., pointwise) equivalent. Still, they are conceptionally quite different.

The first two, i. e., Definitions 7 and 8, are declarative in the sense that they do not say what a permutation is, and they are direct in the sense that they only use the concepts of *ilist*. We consider Definition 7 to capture the intuition and to be conceptually the definition of our choice (in the sequel, we

will always work with Definition 7, unless specified otherwise). Definition 8 shares the same intuition of recursively removing pairs of elements, but with built-in extra uniformity for the possibility of choosing a pair. As mentioned before, this is good for proving transitivity but lacks symmetry in its construction, which is why we prefer Definition 7 as the reference definition. The variant with skeletons, *iperm_skel*, only makes explicit the information contained in a derivation according to Definition 7. Having this otherwise hidden information is necessary if several base relations are considered, as it is done in Lemma 3. The skeletons represent bijective functions on the indices but this information is kept totally implicit. Indeed, we only register indices corresponding to intermediate situations – not to the initial *ilist*.

Definition *ipermb* is based on an explicit representation of bijective functions on the indices. Surjectivity is quite delicate from a constructive point of view. And instead of requiring the existence of an inverse, it is better to exhibit it directly and make it part of the definition of bijectivity as done in *bij*. For *ipermb* to hold, one requires the existence of a bijection such that the elements pointed to by the pairs of indices according to the bijection are in the base relation. Conceptually, this separates the process of finding a permutation into first giving all the pairs of indices and second verifying that the elements corresponding to each pair indeed are in the base relation.

Finally, definition *permcont* given by Contejean immediately allows us to define a fifth notion of permutation:

Definition 14. $\forall l_1\ l_2, ipermcont_R\ l_2\ l_2 \Leftrightarrow permcont_R\ (ilist2list\ l_1)\ (ilist2list\ l_2)$

Since *ilist2list* is a bijection, *ipermcont* is extensionally equivalent to *iperm*, by Theorem 3 and Theorem 4. However, as we have already explained, using lists in the coinductive definition of *Graph* is not an option. Therefore, we prefer a direct definition of permutations on *ilist* rather than through conversion between lists and *ilist*. Moreover, the basic notions on lists are the notions of head and tail, while getting the n^{th} element is not built into the structure. On the contrary, on *ilist*, we can get the n^{th} element directly but it is not trivial to get the tail.

3 A Relation on *Graph* Including Permutations on *ilist*

In this section, we use *iperm* to define a relation on *Graph*. This relation will only solve the problem of permutation between children (see Fig. 1). A refinement that also solves the other problem (change in the point of view for the observation of the graph, see Fig. 2) is presented in Sect. 5.

We define the relation transformer *GPerm* using the same model as for *Geq*:

Definition 15 (*GPerm*, Viewed Coinductively).

$$\frac{g_1\ g_2 : Graph\ T \qquad R\ (label\ g_1)\ (label\ g_2) \qquad iperm_{GPerm_R}\ (sons\ g_1)\ (sons\ g_2)}{GPerm_R\ g_1\ g_2}$$

3.1 An Unsuccessful Try with Guardedness as Implemented in Coq

Proving that *GPerm* preserves reflexivity should be an easy exercise. Notice that this cannot be done by induction on g in *Graph* T since that type is coinductive. Since the proposition to be shown is coinductively defined, the proof naturally has to be carried out by coinduction.

Lemma 5 (*GPerm* Preserves Reflexivity). $\forall R, R\,\mathit{reflexive} \Rightarrow \forall g,\ \mathit{GPerm}_R\ g\ g$

Proof (Coinductive proof). After assuming R and its reflexivity, we try to show our goal by the `cofix` tactic of Coq. The coinductive hypothesis *Hc* is identical with our goal $\forall g,\ \mathit{GPerm}_R\ g\ g$, and it suffices to prove for given g:

$$R\ (\mathit{label}\ g)\ (\mathit{label}\ g) \wedge \mathit{iperm}_{\mathit{GPerm}_R}\ (\mathit{sons}\ g)\ (\mathit{sons}\ g)$$

- $R\ (\mathit{label}\ g)\ (\mathit{label}\ g)$ holds by reflexivity of R
- To prove $\mathit{iperm}_{\mathit{GPerm}_R}\ (\mathit{sons}\ g)\ (\mathit{sons}\ g)$ we may want to use the lemma that *iperm* preserves reflexivity. Thus, to prove $\mathit{iperm}_{\mathit{GPerm}_R}\ (\mathit{sons}\ g)\ (\mathit{sons}\ g)$, we would only have to prove that $\forall g,\ \mathit{GPerm}_R\ g\ g$, which is *Hc*. However, this proof is not correct for Coq because the use of *Hc* is not guarded (it should be used directly under a coinductive constructor and not through a lemma). So, we rather inline the proof of reflexivity of $\mathit{iperm}_{\mathit{GPerm}_R}$ for the argument *sons g*. However, this part of the proof is by induction inside the initial coinduction (the corecursive call is in a recursive construction). Thus, the resulting proof does not pass the guardedness check of Coq.

  Abort proof.

This failure should not be surprising since it just repeats on the proof level the problems we would have had on the level of programming with *Graph*, had we not replaced the use of lists by *ilist* in its definition. In general, Coq support for coinductive types is quite comfortable, which is why the burden of working with *ilist* is compensated for by flexible coinductive definitions of graph structures. On the level of coinductive *proofs* about coinductively defined relations such as *GPerm*, this support does not seem of equal importance to us. Thus, we do not try to solve the issue of the rigid guardedness criterion of Coq as we did for *Graph*, but give up the use of the `CoInductive` command of Coq for getting a coinductive definition of *GPerm*. We rather define *GPerm* "manually" by an impredicative encoding, as shown next.

3.2 Alternative Reading of the Coinductive Definition of *GPerm*

We still view Definition 15 coinductively, but not according to the `CoInductive` keyword of Coq, as we did for all other coinductive definitions. We rather do this by an impredicative definition. This is based on Tarski's fixed point theorem that constructs a greatest fixed point in a complete lattice as the supremum of all of its post-fixed points. In the categorical picture, post-fixed points correspond to coalgebras, but in a complete lattice, we can even enforce finality without continuity. The Tarski construction can be replayed in polymorphic lambda-calculus as an impredicative encoding of greatest fixed points. All this is well

explained in [22], together with historical remarks that, for least fixed points, the corresponding encodings go back to Böhm/Berarducci and Leivant.

While the CIC was an extension of polymorphic λ-calculus (hence, with impredicative universe `Set`) this impredicativity is now optional (so, we did not use it for *Graph*), but the universe of propositions `Prop` remains impredicative. In particular, we can form propositions by quantifying over all propositions or all relations, as they are building blocks of propositions. We use the well-known impredicative construction on `Prop` instead of `Set`. We thus *implement* $GPerm_R\ g_1\ g_2$ by the *proposition* (with $\mathcal{R}$ ranging over relations on $Graph\ T$):
$\exists \mathcal{R}, \big(\forall g_1' g_2', \mathcal{R}\ g_1'\ g_2' \Rightarrow R\ (label\ g_1')\ (label\ g_2') \wedge iperm_{\mathcal{R}}(sons\ g_1')\ (sons\ g_2')\big) \wedge \mathcal{R}\ g_1\ g_2$

Lemma 6 (Coinduction Principle for *GPerm*). *Let us assume*
$\forall g_1\ g_2,\ \mathcal{R}\ g_1\ g_2 \Rightarrow R\ (label\ g_1)\ (label\ g_2) \wedge iperm_{\mathcal{R}}\ (sons\ g_1)\ (sons\ g_2)$
with $\mathcal{R}$ a relation on Graph T. Then, $\forall g_1\ g_2,\ \mathcal{R}\ g_1\ g_2 \Rightarrow GPerm_R\ g_1\ g_2$.

Proof. Trivial by implementation of *GPerm*. □

Lemma 7 (Unfolding Principle for *GPerm*).
$\forall g_1 g_2, GPerm_R\ g_1\ g_2 \Rightarrow R\ (label\ g_1)\ (label\ g_2) \wedge iperm_{GPerm_R}\ (sons\ g_1)\ (sons\ g_2)$

Lemma 8 (Constructor for *GPerm*).
$\forall g_1 g_2, R\ (label\ g_1)\ (label\ g_2) \wedge iperm_{GPerm_R}\ (sons\ g_1)\ (sons\ g_2) \Rightarrow GPerm_R\ g_1\ g_2$

This final result corresponds to the way coinductive definitions are perceived for definitions by `CoInductive` in Coq. In our implementation, the coinduction principle is the obvious result, and the constructor the non-obvious (but well-known for a long time by now). Therefore, these proofs are not shown here.

Remark 7. *GPerm* can also be captured in a Mendler-style coinduction (see [13]) that is directly expressible by `CoInductive` in Coq and also essentially uses impredicativity of `Prop`. This alternative gives a provably equivalent solution, as shown in our Coq development [17].

3.3 Proof that *GPerm* Preserves Equivalence

With the impredicative implementation of *GPerm*, Lemma 5 is now easy to prove.

Proof. Assume that R is reflexive. Instead of our goal $\forall g, GPerm_R\ g\ g$, we show the more general $\forall g_1\ g_2,\ g_1 = g_2 \Rightarrow GPerm_R\ g_1\ g_2$, which is of the right format for Lemma 6, taking $\mathcal{R} := =_{Graph\ T}$. It suffices to show that $\mathcal{R}\ g_1\ g_2$ implies $R\ (label\ g_1)\ (label\ g_2) \wedge iperm_{\mathcal{R}}\ (sons\ g_1)\ (sons\ g_2)$. Since $\mathcal{R}$ is Leibniz-equality, we may replace g_1 by g_2 everywhere. The first conjunct is then done by reflexivity of R, and the second follows from preservation of reflexivity by *iperm*. □

GPerm also preserves symmetry and transitivity. For the former, the use of Lemma 6 is with $\mathcal{R}$ the "flipped" version of $GPerm_R$, hence with interchanged arguments, and uses the good flipping behaviour of *iperm* and Lemma 7.
For transitivity preservation, we set $\mathcal{R}\ g_1\ g_2 \Leftrightarrow \exists g', GPerm_R\ g_1\ g' \wedge GPerm_R\ g'\ g_2$ and use Lemma 6 for it. Besides Lemma 7, this requires a quite special form of transitivity of *iperm* whose proof must be based on Definition 8 and Theorem 1:
$iperm_{R_1} l_1\ l' \wedge iperm_{R_2} l'\ l_2 \Rightarrow iperm_{R_3} l_1\ l_2$ with $R_3\ t_1\ t_2 \Leftrightarrow \exists t',\ R_1\ t_1\ t' \wedge R_2\ t'\ t_2$

4 An Equivalent Approach Based on Observations

As we saw in Sect. 3.1, the problem was the embedding of inductive proofs (the existence of a permutation) in a coinductive proof that *GPerm* holds. This could be overcome by the impredicative encoding of *GPerm*, but there is also the well-known option of abandoning coinduction in favour of some observational equivalence. Here, the idea is to use inductive trees (the finite counterpart to *Graph*) to represent the unfolding of elements of *Graph* until a certain depth. Then, we define a relation transformer on these trees using *iperm*, and we prove that it preserves equivalence. To "observe" until a depth n means cutting off the graphs all the structure beyond depth n, which yields such inductive trees, so we can relate them by the relation on trees (obviously, we expect the relation to be decidable). If the relation holds for all depths, the graphs are considered "observationally equal". Thus, on the logical side, we only work on inductive structures (while the graphs are still coinductive structures).

Finally, adding a version of the infinite pigeonhole principle as an axiom to Coq (that could in turn be justified by the classical law of excluded middle), we prove that observational equivalence is equivalent to *GPerm* (for any base relation). Therefore, we get an alternative proof that *GPerm* preserves equivalence.

4.1 Definitions Based on Inductive Trees

We define the type of inductive trees (we call it *iTree*) as follows:

Definition 16 (*iTree*, Viewed Inductively).

$$\frac{t : T \qquad l : \mathit{ilist}(\mathit{iTree}\ T)}{\mathit{mk\_iTree}\ t\ l : \mathit{iTree}\ T}$$

We call *labT* and *sonsT* the two projection functions on *iTree*. We use t_1 and t_2 to denote elements of *iTree* T.

Now, we can define a relation on *iTree* using the permutation relation on *ilist*. We define it in analogy with *GPerm*, but inductively:

Definition 17 (*TPerm*, Viewed Inductively).

$$\frac{t_1\ t_2 : \mathit{iTree}\ T \quad R\ (\mathit{labT}\ t_1)\ (\mathit{labT}\ t_2) \quad \mathit{iperm}_{\mathit{TPerm}_R}\ (\mathit{sonsT}\ t_1)\ (\mathit{sonsT}\ t_2)}{\mathit{TPerm}_R\ t_1\ t_2}$$

To prove that *TPerm* preserves equivalence, we reason by induction. As *iperm* is also inductive, the proofs are straightforward and do not present any difficulties.

As we have said, the purpose of using inductive trees built with the same pattern as *Graph* is to capture the unfolding of an element of *Graph* until a depth n. Therefore, we have to define a function that transforms an element of *Graph* into an element of *iTree*. We call this function *G2iT*.

Definition 18 (*G2iT*, Defined Recursively).

$$\begin{aligned}
&\mathit{G2iT} : \forall T, \mathbb{N} \to \mathit{Graph}\ T \to \mathit{iTree}\ T\\
&\mathit{G2iT}\ T\ 0\ g := \mathit{mk_iTree}\ (\mathit{label}\ g)\ [\![]\!]\\
&\mathit{G2iT}\ T\ (n{+}1)\ (\mathit{mk_Graph}\ t\ l) := \mathit{mk_iTree}\ t\ (\mathit{imap}\ (\mathit{G2iT}\ T\ n)\ l)
\end{aligned}$$

where $[\![]\!]$ *stands for the empty ilist and imap is the ilist analogue of the map function on lists (see [16]) and has type* $\forall T\ U,\ (T \to U) \to ilist\ T \to ilist\ U$.

We also define a notation to say that two elements of *Graph* extracted up to level n are equivalent through *TPerm*. We call it $\equiv$:

Definition 19. $\forall n\ g_1\ g_2,\ g_1 \equiv_{R,n} g_2 \Leftrightarrow TPerm_R\ (G2iT\ n\ g_1)\ (G2iT\ n\ g_2)$

Lemma 9 (Recursive description of $\equiv_{R,n}$ as a function of n).

$$\forall g_1\ g_2, g_1 \equiv_{R,0} g_2 \Leftrightarrow R\ (label\ g_1)\ (label\ g_2)$$

$$\forall g_1\ g_2, g_1 \equiv_{R,n+1} g_2 \Leftrightarrow R\ (label\ g_1)\ (label\ g_2) \wedge iperm_{\equiv_{R,n}}\ (sons\ g_1)\ (sons\ g_2)$$

Remark 8. As should be expected from a relation based on finite observations, $\equiv_{R,n}$ preserves decidability (of R). The proof is by induction on n using Lemmas 1 and 9.

Our goal here is to prove that, if for all n, two elements of *Graph* transformed to *iTree* by *G2iT* n are equivalent through *TPerm*, then they are equivalent through *GPerm* and vice versa. We first define the relation on *Graph* that uses *TPerm*. We want to prove that it is equivalent to *GPerm*. We call it *GTPerm*.

Definition 20 (*GTPerm*). $\forall g_1\ g_2, (GTPerm_R\ g_1\ g_2 \Leftrightarrow \forall n, g_1 \equiv_{R,n} g_2)$

One shows that *GTPerm* preserves equivalence using the fact that *TPerm* does.

4.2 Main Theorem

Theorem 5. $\forall g_1\ g_2, GPerm_R\ g_1\ g_2 \Leftrightarrow GTPerm_R\ g_1\ g_2$

The proof extends until the end of this section and makes use of Axiom 1 below.

Proof.

[**Direction** $\Rightarrow$] This is proved by induction on n (depth of observation of g_1 and g_2) using Lemma 9.

[**Direction** $\Leftarrow$] This direction is much harder, and needed concepts and results will be introduced after showing the overall structure. We do it as a coinductive proof, i. e., we apply Lemma 6. Obviously, we use it with $\mathcal{R} := GTPerm_R$. Given g_1, g_2 and $GTPerm_R\ g_1\ g_2$, we have to show $R\ (label\ g_1)\ (label\ g_2)$ and $iperm_{GTPerm_R}\ (sons\ g_1)\ (sons\ g_2)$. For the first part, we use the definition of $GTPerm_R$ with $n := 0$ and the base case of Lemma 9. The second part is guaranteed by Lemma 10. □

Lemma 10. $\forall g_1\ g_2, GTPerm_R\ g_1\ g_2 \Rightarrow iperm_{GTPerm_R}\ (sons\ g_1)\ (sons\ g_2)$

[**Proof of Lemma 10**] Actually, here, the main challenge is a problem of continuity. Indeed, unfolding the definition of *GTPerm*, we obtain the goal:

$$\forall g_1\ g_2, (\forall n, g_1 \equiv_{R,n} g_2) \Rightarrow iperm_{\cap_n \equiv_{R,n}}\ (sons\ g_1)\ (sons\ g_2)$$

So this actually means that we can "fix" a permutation for $(sons\ g_1)$ and $(sons\ g_2)$ that will be valid for any depth of extraction through $G2iT$. This is not trivial, because some permutations may be valid until a certain depth but not afterwards. For example, in Fig. 3, only the permutation symbolized by the thick plain arrows would be valid for the level with $depth = 2$. The thick dashed one would be valid for $depth = 1$ but not afterwards (given standard equality on the labels). However, in the opposite direction, there is no problem. Indeed, if a permutation is valid at a certain depth n, then for all shallower depths m ($m \leq n$), this same permutation is valid. We express this result only partly in Lemma 11 since we do not capture in what sense this is the "same permutation". The proof is not detailed here.

Lemma 11. $\forall g_1\ g_2\ n\ m, m \leq n\ \wedge\ g_1 \equiv_{R,n} g_2 \Rightarrow g_1 \equiv_{R,m} g_2$

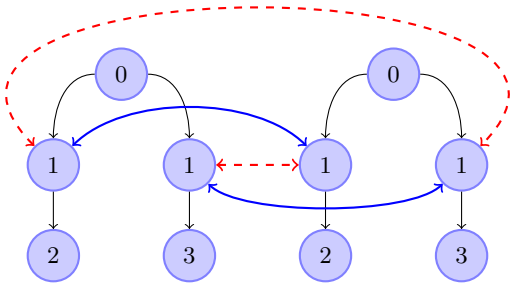


Fig. 3. Graphs with various possibilities of permutations on the first level

To prove Lemma 10, we use a specific formulation of the infinite pigeonhole principle. It informally states that if an infinity of items is put into a finite number of holes, then at least one hole must contain an infinity of items. Although obviously true, this principle has no constructive justification since no finite observation of the infinite process of putting the items into the holes allows to determine which hole is used infinitely often.
For our problem, the infinitely many items are the unfolding levels (for the elements of *Graph*) and the holes are the possible permutations (as $sons\ g_1$ and $sons\ g_2$ are finite, the number of possible permutations also is, as we only consider the permutations on the top level).
However, this requires us to be able to manipulate permutations. Hence, we have to use the definition of *iperm_skel* given in Sect. 2.2. We can do this directly since we showed that it was equivalent to *iperm*.
The infinite pigeonhole principle will only be needed for the finite types of the form $skel\_type\ m$. We have to assume it as an axiom on top of the intuitionistic type theory CIC that underlies Coq. It could be justified by the classical law of excluded middle which is consistent with the CIC.

Axiom 1 (Infinite Pigeonhole Principle).

$$\forall m\ \forall P : \mathbb{N} \to skel\_type\ m \to Prop,\ (\forall n \exists s : skel\_type\ m, P\ n\ s) \to \exists s_0 : skel\_type\ m, (\forall n \exists n', n' \geq n \wedge P\ n'\ s_0)$$

Here, we use the letter P for the relation that describes which "pigeon" n "goes" into which "hole"/skeleton s. We can now come back to Lemma 10 itself. We want to prove that:

$$GTPerm_R\ g_1\ g_2 \Rightarrow iperm_{GTPerm_R}\ (sons\ g_1)\ (sons\ g_2)$$

It is easy to prove $lg(sons\ g_1) = lg(sons\ g_2)$ (call its proof H_{lg}) using the definition of $GTPerm$ with n:=1. We can also prove the following assertion H_1:

$$\forall n \exists s : skel\_type(lg(sons\ g_1)), iperm\_skel_{\equiv_{R,n}}\ (sons\ g_1)\ (sons\ g_2)\ H_{lg}\ s$$

Indeed, for argument n, the definition of $GTPerm_R\ g_1\ g_2$ can be used at depth $n+1$, and Lemma 9 allows to infer $iperm_{\equiv_{R,n}}\ (sons\ g_1)\ (sons\ g_2)$, from which we get the skeleton by Lemma 2.
The application of the pigeonhole principle (Axiom 1) gives us the "good" permutation skeleton s_0 for infinitely many levels, as expressed in the property

$$\forall n \exists n', n' \geq n \wedge iperm\_skel_{\equiv_{R,n'}}\ (sons\ g_1)\ (sons\ g_2)\ H_{lg}\ s_0 \quad (H_2)$$

In fact, s_0 is already the "good" skeleton for all levels: using Lemma 2, we can change the goal of the proof to:

$$iperm\_skel_{GTPerm_R}\ (sons\ g_1)\ (sons\ g_2)\ H_{lg}\ s_0$$

and by Lemma 3, we can transform it again into

$$\forall n, iperm\_skel_{\equiv_{R,n}}\ (sons\ g_1)\ (sons\ g_2)\ H_{lg}\ s_0$$

Given n, we choose n' according to H_2. Then, by Lemma 11, $\equiv_{R,n'}$ is a subrelation of $\equiv_{R,n}$. Therefore, using the monotonicity of *iperm_skel* in its relation argument, we obtain the goal from the main property of n' that H_2 guarantees.

This ends the proof of Lemma 10 and hence completes that of Theorem 5. □

5 The Final Relation

The preceding two sections solve the issue of permutations in the children. We are going to integrate it in a solution that also solves the issue of change of root.

5.1 The Idea

Graph allows to represent single-rooted connected graphs, i. e., there is a path to any node of the graph from the root. If we are able to change the "point of view", i. e., to represent the same graph but with another node as the root (see Fig. 2), then there is a path from the new root to the old one and inversely.

Remark 9. This change is only allowed for a "general" view, not for the inner nodes. For example, we do not want the graphs of Fig. 4 to be equivalent.

Fig. 4. Graphs with inner cycle turned

We will call g_1 and g_2 the two elements of *Graph* that represent the same graph but with a different root. Saying that there is a path from g_1 to g_2 and from g_2 to g_1 means that g_1 is "included" in g_2 and that g_2 is "included" in g_1. In [16] we have presented a notion of strict inclusion that we have called *GinG*. This inclusion was built using *Geq*. Here, we want to add permutations, so we want to have the same kind of relation but using *GPerm*. So we can define it in a general way (for any relation on *Graph*) and then we will instantiate it for *GPerm*. We define it as follows (with R_G a relation on *Graph* T):

Definition 21 (*GinG, Viewed Inductively).**

$$\frac{R_G\ g_{in}\ g_{out}}{GinG^*_{R_G}\ g_{in}\ g_{out}} \qquad \frac{GinG^*_{R_G}\ g_{in}\ (fct(sons\ g_{out})\ i)}{GinG^*_{R_G}\ g_{in}\ g_{out}}$$

To find properties of $GinG^*_{,}$ we need to know more about the behaviour of R_G regarding propagation towards the children. We instantiate *GinG** for *GPerm*:

Definition 22 (*GinGP*). $GinGP_R := GinG^*_{GPerm_R}$

For *GinGP*, we can prove various properties among which preservation of transitivity and compatibility with relation $GPerm_R$ in its two arguments.

5.2 *GeqPerm*

Using this relation of inclusion, we can now define the final relation on *Graph* that takes into account permutations over the children and a change in the point of view of the representation. We call it *GeqPerm*. It checks that the first element of *Graph* is included in the second, and inversely. The case when there is no change in the point of view for the observation is managed by the fact that *GinG** preserves reflexivity (first case of the definition). We define *GeqPerm* as follows:

Definition 23 (*GeqPerm*).

$$\forall g_1\ g_2,\ GeqPerm_R\ g_1\ g_2 \Leftrightarrow GinGP_R\ g_1\ g_2 \wedge GinGP_R\ g_2\ g_1$$

As *GinGP* preserves reflexivity and transitivity, *GeqPerm* preserves equivalence.

It is now possible to show that the graphs of Fig. 1 and Fig. 2 are equivalent and that the graphs of Fig. 4 are not. The proofs that two elements of *Graph* are not equivalent may be tedious because we have to test all possible permutations (in the example of Fig. 4 the nodes only have one child, so the proof is short). Of course, one can also mix the features of Fig. 1 and Fig. 2.

Remark 10. We now have at our disposal the relation on *Graph* that we would natively have obtained with a standard set-theoretic definition of graphs. But, we have the computational advantages of our constructive definitions.

6 Conclusions

In this paper, we have presented and compared various solutions to capture permutations on *ilist* modulo some possibly undecidable base relation R on the elements of the *ilist*. This was crucial since decidability cannot be guaranteed (not even expected) for our graph representation *Graph*. We have shown that these relation transformers preserve equivalence (map equivalence relations to equivalence relations). And we proved that they are all equivalent. Then, we have integrated one of these relations (the one that we considered to capture best the intuition of permutations) into a relation *GPerm* to include permutations over children. Using *ilist* instead of lists was needed since with lists, the coinductive representation of graphs would not have allowed programming in Coq.

We have also presented a solution that allows us to observe our graphs on finite structures (the inductive trees *iTree*) via the function *G2iT* that converts elements of *Graph* to elements of *iTree* up to a certain depth. We provide a relation transformer *TPerm* for *iTree* in analogy with what we did for *Graph*, including permutations over the children of the elements of *iTree*. We can thus formulate that two graphs are "bisimilar" up to any depth by reference to *G2iT* and *TPerm*, which yields the relation *GTPerm*. For a decidable base relation, we obtained that "bisimilarity" up to any given depth is decidable. We have included into the main text a proof of equivalence of *GPerm* with *GTPerm* (Theorem 5), hence between the coinductive concept and the one based on observations.

Finally, we included our permutation-aware relation on *Graph* into a wider relation taking into account changes in the point of view of the representation of the elements of *Graph* (*GeqPerm*). We proved that this final relation (transformer) preserves equivalence and that it indeed solved the issues presented in Sect. 1.2.

In the course of our research we observed that even for lists, the standard library would not have solved our problems concerning permutations parameterized freely by a possibly undecidable relation (see [20] and [21]). We are currently working on an extension of Contejean's [6] in the style of our solutions for *ilist*.

If we chose to ignore multiplicities in *ilist*, we could define a new relation corresponding to the equivalence between two sets. This could be an interesting direction to investigate to obtain an even more liberal relation on graphs.

Berger [3] uses an embedded inductive definition in a coinductive one for the purpose of real number computability, with extraction of Haskell programs. We wonder if his work could be formalized in Coq using the methods presented here.

Finally, it would be interesting to extend *GeqPerm* to a wider class of graphs covering also non-connected graphs, e. g., by using the notion of *forest* of graphs.

Acknowledgements We warmly thank our reviewers for their attentive reading and interesting propositions and challenges.

References

1. Altenkirch, T.: A formalization of the strong normalization proof for system F in LEGO. In: Bezem, M., Groote, J.F. (eds.) Typed Lambda Calculi and Applications, International Conference. LNCS, vol. 664, pp. 13–28. Springer (1993)

2. Berardi, S., Damiani, F., de'Liguoro, U. (eds.): Types for Proofs and Programs, International Conference, TYPES 2008, Revised Selected Papers, Lecture Notes in Computer Science, vol. 5497. Springer (2009)
3. Berger, U.: From coinductive proofs to exact real arithmetic: theory and applications. Logical Methods in Computer Science 7(1) (2011)
4. Bertot, Y., Komendantskaya, E.: Using structural recursion for corecursion. In: Berardi et al. [2], pp. 220–236
5. Chlipala, A.: In Coq club thread "is Coq being too conservative?" (January 2010), https://sympa-roc.inria.fr/wws/arc/coq-club/2010-01/msg00089.html
6. Contejean, E.: Modeling permutations in Coq for Coccinelle. In: Comon-Lundh, H., Kirchner, C., Kirchner, H. (eds.) Rewriting, Computation and Proof. Lecture Notes in Computer Science, vol. 4600, pp. 259–269. Springer (2007)
7. Coquand, T.: Infinite objects in type theory. In: Barendregt, H., Nipkow, T. (eds.) Types for Proofs and Programs, International Conference, TYPES 1993. Lecture Notes in Computer Science, vol. 806, pp. 62–78. Springer (1993)
8. Dams, C.: In Coq club thread "is Coq being too conservative?" (January 2010), https://sympa-roc.inria.fr/wws/arc/coq-club/2010-01/msg00085.html
9. Danielsson, N.A.: Beating the productivity checker using embedded languages. In: Bove, A., Komendantskaya, E., Niqui, M. (eds.) Proceedings Workshop on Partiality and Recursion in Interactive Theorem Provers. EPTCS, vol. 43, pp. 29–48 (2010)
10. Danielsson, N.A., Altenkirch, T.: Subtyping, declaratively. In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) Mathematics of Program Construction (MPC'10). Lecture Notes in Computer Science, vol. 6120, pp. 100–118. Springer (2010)
11. Giménez, E., Castéran, P.: A tutorial on [co-]inductive types in Coq (2007), www.labri.fr/perso/casteran/RecTutorial.pdf
12. McBride, C., McKinna, J.: The view from the left. Journal of Functional Programming 14(1), 69–111 (2004)
13. Nakata, K., Uustalu, T.: Resumptions, weak bisimilarity and big-step semantics for while with interactive I/O: An exercise in mixed induction-coinduction. In: Aceto, L., Sobocinski, P. (eds.) SOS. EPTCS, vol. 32, pp. 57–75 (2010)
14. Nakata, K., Uustalu, T., Bezem, M.: A proof pearl with the fan theorem and bar induction - walking through infinite trees with mixed induction and coinduction. In: Yang, H. (ed.) APLAS. LNCS, vol. 7078, pp. 353–368. Springer (2011)
15. Niqui, M.: Coalgebraic reasoning in Coq: Bisimulation and the lambda-coiteration scheme. In: Berardi et al. [2], pp. 272–288
16. Picard, C., Matthes, R.: Coinductive graph representation: the problem of embedded lists. Electronic Communications of the EASST 39 (2011), 24 pp
17. Picard, C., Matthes, R.: Formalization in Coq for this article (2012), www.irit.fr/~Celia.Picard/Coq/Permutations/
18. Prince, R., Ghani, N., McBride, C.: Proving properties about lists using containers. In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS. Lecture Notes in Computer Science, vol. 4989, pp. 97–112. Springer (2008)
19. The Coq Development Team: The Coq Proof Assistant Reference Manual, http://coq.inria.fr
20. The Coq Development Team: The Coq Proof Assistant Standard Library, http://coq.inria.fr/stdlib/Coq.Sorting.Permutation.html
21. The Coq Development Team: The Coq Proof Assistant Standard Library, http://coq.inria.fr/stdlib/Coq.Sorting.PermutSetoid.html#permutation
22. Uustalu, T., Vene, V.: Least and greatest fixed points in intuitionistic natural deduction. Theoretical Computer Science 272, 315–339 (2002)