Electronic Communications of the EASST
Volume 39 (2011)

Graph Computation Models
Selected Revised Papers from the
Third International Workshop on
Graph Computation Models (GCM 2010)

Coinductive Graph Representation: the Problem of Embedded Lists

Celia Picard and Ralph Matthes

24 pages

Guest Editors: Rachid Echahed, Annegret Habel, Mohamed Mosbah
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Coinductive Graph Representation: the Problem of Embedded Lists

Celia Picard and Ralph Matthes

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

Abstract: When trying to obtain formally certified model transformations, one may want to represent models as graphs and graphs as greatest fixed points. To do so, one is rather naturally led to define co-inductive types that use lists (to represent a finite but unbounded number of children of internal nodes). These concepts are rather well supported in the proof assistant Coq. However, their use in our intended applications may cause problems since the co-recursive call in the type definition occurs in the list parameter. When defining co-recursive functions on such structures, one will face guardedness issues, and in fact, the syntactic criterion applied by the Coq system is too rigid here.

We offer a solution using dependent types to overcome the guardedness issues that arise in our graph transformations. We also give examples of further properties and results, among which finiteness of represented graphs. All of this has been fully formalized in Coq.

Keywords: coinduction/corecursion, guardedness, theorem proving, dependent types, metamodels

1 The Problem: Explanation on an Example

It is recognized that the on-going engineering effort for modeling and meta-modeling has to be backed by rigorous formal methods. In this context, we aim at performing certified model transformations. In a first time, certification should be done by interactive theorem proving. This presupposes the representation of models and metamodels in the language of the theorem prover. We chose to represent models and metamodels as graphs and to use the Coq system [1] as a specification and verification tool. The Coq system offers a language with a rich notion of inductive and co-inductive types, i. e., data types that arise as least and greatest solutions of fixed-point equations, respectively.
This led us to represent node-labeled graphs with co-inductive types (in order to represent the infinite navigability in loops). The idea we had was that each node would have a label (of a type T) and a finite list of sons (graphs themselves). This type can be created through the following constructor:

Definition 1 (*Graph*, Viewed Coinductively)

$$mk\_Graph : T \to list\ (Graph\ T) \to Graph\ T$$

[1] See http://coq.inria.fr/

mk_Graph t l constructs a graph from the label t of type T and the list of graphs l. Since this is the greatest fixed point, no assumption about finite generation through *mk_Graph* is made. The empty list hides the base case.

Remark 1 (Lists) *For lists we use the Caml notation:* [] *for the empty list,* $[a_1;a_2;\ldots]$ *for an explicit enumeration and* $a :: l$ *for the cons operation.*

Remark 2 (Rose trees) *What we represent here are actually potentially infinite "rose trees" that may have cycles. They can be considered as a dual version of rose trees although the lists are not dualised. Finite rose trees form a well-known example for datatypes in the community around the Haskell programming language (see [Bir01] for example).*

Remark 3 *Here we deal (at first sight at least) with single-rooted connected graphs because they correspond best to co-inductive types. A more general solution is presented in Subsection 4.1, still within the expressive power of Coq.*

Remark 4 (Notations) *In the rest of the paper we will use the following notation:*

- T, U *for types and* t *for elements of type* T,
- n, m *and* k *for natural numbers (informally, but also for elements of type nat),*
- l *and* q *for lists and elements of ilist, defined in Section 2*
- f *for functions,*
- g *for elements of Graph,*
- R *for equivalence relations (if* R *is a relation on type* T *it has type* $T \to T \to Prop$*),*
- P *for predicates (if* P *is over type* T *it has type* $T \to Prop$*)*
- i *for elements of Fin n, defined in Section 2*

Remark 5 (Examples) *In all the examples that will be given in this paper, we will use natural numbers as labels of nodes, i. e.,* $T = nat$.

Example 1 (A simple example that does not use co-recursion: just a leaf)

$$Leaf_n := mk\_Graph\ n\ []$$

Example 2 (Example of a finite graph) *The graph of Figure 1 can be represented as a term of type Graph with the following co-recursive definition:*

$$Finite\_Graph := mk\_Graph\ 0\ [mk\_Graph\ 1\ [Finite\_Graph]]$$

Remark 6 *This graph is finite but unfolds into an infinite (regular) tree, and thus allows infinite navigation.*

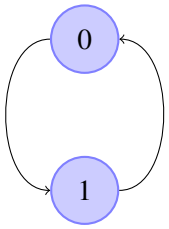


Figure 1: Example of a finite graph

Example 3 (Example of an infinite graph) *To represent the graph of Figure 2 as a term of type Graph, we first define a family of infinite graphs, parameterized by the label of the first node:*

$$Infinite\_Graph_n := mk\_Graph\ n\ [Infinite\_Graph_{n+1}]$$

The graph of Figure 2 corresponds to $Infinite\_Graph_0$.

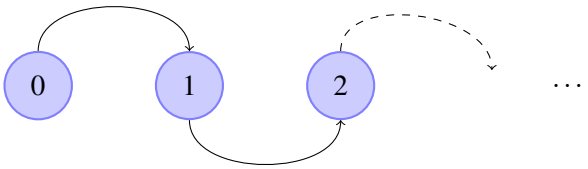


Figure 2: Example of an infinite graph

Remark 7 *This graph is infinite and unfolds into an infinite irregular tree.*

The previous definitions and examples are well-defined in Coq. However, problems arise when trying to apply a transformation on a graph. For example, it is forbidden to define the following co-recursive function $applyF2G : \forall\ T\ U,\ (T \to U) \to Graph\ T \to Graph\ U$ that applies a function f to each label of a graph:

Definition 2 $applyF2G\ f\ (mk\_Graph\ t\ l) := mk\_Graph\ (f\ t)\ \big(map\ (applyF2G\ f)\ l\big)$

Remark 8 *Here, map is the usual mapping function that maps a function over all the elements of a list, i. e.,* $map\ f\ [a_1; a_2; \ldots] = [f\ a_1; f\ a_2; \ldots]$.

The reason why *applyF2G* is not accepted by Coq is that the guardedness condition on co-inductive types is rather restrictive in Coq, and in this case, too restrictive. Indeed, in Coq the guardedness condition is based on productivity [Coq93]. Technically speaking, it says that a co-recursive call must always be the argument of some constructor of inductive or co-inductive type. Here, the co-recursive call is an argument of the *map* function, which is itself under the constructor. This is too indirect to satisfy the guardedness condition. For more details about the guardedness conditions in Coq see [BK08] and [GC07].

Basically, the idea of the guardedness condition is to ensure that potentially infinite objects are computable. This means that we can always obtain more information on the structure of the object in a finite amount of time. Consider the example of streams that are always infinite. The application of a filter on streams is actually a problem since we cannot ensure that the next "good" element will be found in a finite amount of time. But here the problem is quite different in nature: it is not about finding the next constructor but about the indirection of the co-recursive call through *map*. However, in the case of *map*, this indirection is harmless (we would only have to inspect in parallel the elements of that list).
So, Coq's guardedness condition forbids us to write semantically well-formed definitions: guardedness restrictions go beyond syntactic well-formedness and normal typing constraints but are still of a syntactic nature and thus only approximate the semantic notion of productivity that guarantees well-definedness.

In this article, we offer and study a solution to overcome the problem with the guardedness condition for definitions involving graphs. In Section 2 we explain the solution, and we will see how it solves our problem in Section 3. Finally, in Section 4 we present two extensions that bring us closer to a real metamodel representation.
Even though this article is not written in the formal language of the Coq system, all the work presented here has been formally proved in Coq (version 8.3). The whole development is available in [PM11].

2 The Solution: *ilist*

We develop here a solution that allows us to bypass the guardedness condition.

2.1 The Idea

The idea to solve the problem is to use a function that mimics the behaviour of lists (this idea has also been mentioned by Chlipala in [Chl10]). Lists can easily be seen as functions. If T is the type parameter, then a list can be considered as a function that associates to each element of a set of n elements (n being the length of the list) an element of type T. An element of the definition domain represents the position of the associated element in the list.

Example 4 *The list* $[10\,;2\,;5]$ *can be transformed into the function of Figure 3.*

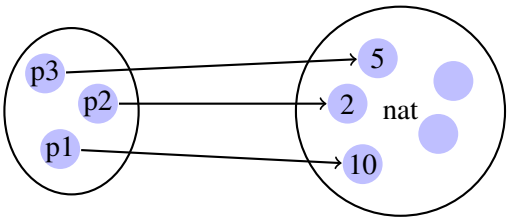


Figure 3: Representation of the function corresponding to the list $[10\,;\,2\,;\,5]$

But to be able to represent such a function, we need to have a set of n elements.

2.2 *Fin* – a Family of Types for Finite Index Sets

It is trivial to get an inductive type with n elements, for $n = 0, 1, 2, \ldots$ but it is not for an indeterminate n. Here we need n to be a parameter of the type. To represent a set of n elements, we have chosen to use the representation that has also been used by Altenkirch in [Alt93] and by McBride and McKinna in [MM04]. We actually represent a family of sets parameterized by the number of elements they contain (in our case, the length of the list). This family is called *Fin*. *Fin* has type $nat \to Set$. It is defined through the two following constructors:

Definition 3 (*Fin*, Viewed Inductively)

$$\begin{array}{lll} first & n: & Fin\ (n+1) \\ succ & n: & Fin\ n \to Fin\ (n+1) \end{array}$$

Remark 9 *Fin is a Generalized Algebraic Data Type (GADT). Those data types are also available in current implementations of the Haskell programming language.*

Remark 10 *The first argument n to succ is determined by the type of the second argument. Therefore, we tend to omit this first argument.*

First of all, we want to prove that *Fin n* indeed is a set of n elements.

Remark 11 *We use card to represent the cardinality of the set in an informal way.*

Lemma 1 $\forall n,\ card\ \{i \mid i : Fin\ n\} = n.$

Proof (by induction).

[Case 0] No constructor allows to create an element of type *Fin* 0.
Therefore $card\ \{\ i \mid i\ :\ Fin\ 0\ \}\ =\ 0$

[Case n+1] With the constructor *succ*, we can construct as many elements of $Fin\ (n+1)$ as there are in *Fin n*. The constructor *first* allows us to construct one more element of $Fin\ (n+1)$. Therefore, $card\ \{i \mid i : Fin\ (n+1)\} = card\ \{i \mid i : Fin\ n\} + 1 = n+1$, using the induction hypothesis.

□

Remark 12 *This informal proof cannot be formalized in Coq because there is no such card operation. With the card operation the following result would have been a triviality.*

Lemma 2 $\forall n\ m,\ n = m \Leftrightarrow Fin\ n = Fin\ m.$

Proof.

[Direction ⇒] The proof here is straightforward, it is only a matter of rewriting and we directly have the property. In informal mathematics this would not even be stated.

[Direction $\Leftarrow$] This direction is much trickier than the first one. Indeed, the first idea we had was to show that all the elements of *Fin n* are in *Fin m* too, doing a type rewrite on the type of the elements. However, that does not seem to work in Coq (at least, we did not find a way to do it).
In order to prove this property, we defined the type of segments of natural numbers (let's call it *NatSeg*): $NatSeg\ n := \{\ m \mid m < n\ \}$. We proved that if there is a bijection between *NatSeg n* and *NatSeg* n' then $n = n'$. The proof is not straightforward, but at least we could do it [2]. Then we could prove that there is a bijection between *Fin n* and *NatSeg n* and that therefore, $\forall n\ m,\ Fin\ n = Fin\ m \Rightarrow n = m$.
□

Remark 13 *One may wonder why we did not use directly NatSeg instead of Fin to represent a set of n elements. The reason is that it is much more comfortable to have an inductive type (with concrete finite elements). The elements of NatSeg n contain a proof of* $m < n$*, and we consider Fin more elementary.*

2.3 *ilist* Implementation

Using the preceding definition of the domain of the functions to be used, we can define the type of functions itself (let's call it *ilistn*).

2.3.1 The Type of Functions *ilistn*

It has two parameters: the type of the elements of the list and its length. It is defined as follows:

Definition 4 $ilistn\ T\ n := Fin\ n \to T$

Elements of *ilistn* "mimic" lists. To each element of a set of *n* elements, it associates an element of type T. However, one problem remains. Indeed, as we said, *ilistn* needs two parameters. But for a list, the length is not one of its parameters, it is inherent to it.

2.3.2 The List Counterpart, *ilist*

To solve this problem, we create a new type that combines the length of the list and the corresponding *ilistn*. We call it *ilist* :

Definition 5 $ilist\ T := \Sigma n : nat.\, ilistn\ T\ n$

Here, we use the dependent pair that is generically denoted as $\Sigma x : A.B(x)$. Elements of this type consist of an element a of type A and an element b of type $B(a)$.

The two projection functions on *ilist* are called *lgti* (which stands for the length of an element of *ilist*) and *fcti* (which stands for the function in an element of *ilist*). If we note $\langle \ldots, \ldots \rangle$ the constructor for elements of type $\Sigma x : A.B(x)$, then an element l of type $ilist\ T$ can be "reconstructed" as $\langle lgti\ l, fcti\ l \rangle$.

[2] We had the confirmation by other members of the Coq user community that no simple proof was known yet.

2.3.3 An Equivalence on *ilist*

It is very useful to be able to compare two elements of the same type. Here, of course, we would like to be able to compare two elements of *ilist*. For *Fin* there was no problem, it is inductive and does not have any type parameter, so Leibniz equality is fine. To recall, Leibniz equality is the propositional equality that allows the replacement of Leibniz-equal elements in any context. In this paper (as in Coq), it is denoted by the infix "=" symbol or the prefix *eq* relation symbol.
Here, the problem is different. We intuitively see that in order to compare elements of *ilist*, we will have to compare two different things: the two parts of its definition. The first one, its length, is the easy one: it is a natural number, no problem here. But the second one is trickier. Indeed, we have to make sure that the two elements of *ilist* we are comparing are equivalent element-wise. And we have no insurance that they are comparable w. r. t. Leibniz equality (actually, in our graph representation, they are not, they are only comparable through bisimulation). We thus define an inductive proposition (let's call it *ilist_rel* because it is the lifting of *ilist* to relations) that relates two elements of *ilist*. Apart from the elements of *ilist* we are comparing and the type parameter (let's call it T), the proposition needs a given base relation R on type T. Then, *ilist_rel R* has type $ilist\ T \to ilist\ T \to Prop$.
Intuitively, we would like to define *ilist_rel* such that:

$$\forall l_1\ l_2 : ilist\ T, ilist\_rel\ R\ l_1\ l_2 \Leftrightarrow lgti\ l_1 = lgti\ l_2 \wedge \forall i : Fin\ (lgti\ l_1), R\ (fcti\ l_1\ i)\ (fcti\ l_2\ i)$$

However, this expression is not well-typed. Indeed, $fcti\ l_2$ has type $Fin\ (lgti\ l_2) \to T$ and i has type $Fin\ (lgti\ l_1)$. We know that $lgti\ l_1 = lgti\ l_2$ but the types $Fin\ (lgti\ l_1)$ and $Fin\ (lgti\ l_2)$ are still syntactically different. Therefore, we must convert i to type $Fin\ (lgti\ l_2)$ (the hypothesis $lgti\ l_1 = lgti\ l_2$ and Lemma 2 "⇒" ensure that we have the right to do it). In Coq, there is a special pattern matching feature that allows us to make this type rewrite. We do not detail it here, for more information see [TCDT, Chapter 1.2.13 and 4.5.4].
In a context where $h : lgti\ l_1 = lgti\ l_2$ and $i : Fin\ (lgti\ l_1)$, we call i'_h the result of converting i to type $Fin\ (lgti\ l_2)$.
With this we can properly write our definition for *ilist_rel*:

Definition 6 (*ilist_rel*)

$$\forall l_1\ l_2 : ilist\ T, ilist\_rel\ R\ l_1\ l_2 \Leftrightarrow \exists h : lgti\ l_1 = lgti\ l_2,\ \forall i : Fin\ (lgti\ l_1), R\ (fcti\ l_1\ i)\ (fcti\ l_2\ i'_h)$$

Using advanced dependently typed pattern matching techniques, one can show that *ilist_rel R* is an equivalence relation if R is one which we assume throughout.

Remark 14 *In the sequel, we will put argument R as an index to ilist_rel, i. e., we will write* $ilist\_rel_R$ *for ilist_rel R, and we will do so in all similar cases.*

Remark 15 (*list_rel*) *Had we worked with lists, we would have needed to be able to compare two lists. However, the commonly used relation on lists is Leibniz equality, but this relation supposes that the elements of the lists are comparable through Leibniz equality, too. As we will explain in Section 3, in our case (graph representation through Graph) they are only comparable*

through bisimulation. Therefore, we would have needed to define a relation on lists parameterized by a relation on the type of its elements and prove that it is an equivalence relation (which is rather easy to do).

Definition 7 (*list_rel*)

$$\forall (l_1\ l_2 : list\ T), list\_rel_R\ l_1\ l_2 \Leftrightarrow \begin{cases} or & l_1 = [] \wedge l_2 = [] \\ & \exists\ t_1\ t_2\ q_1\ q_2,\ l_1 = t_1 :: q_1 \wedge l_2 = t_2 :: q_2 \wedge R\ t_1\ t_2 \wedge list\_rel_R\ q_1\ q_2 \end{cases}$$

It is easy to prove that $list\_rel_{eq}\ l_1\ l_2 \Leftrightarrow l_1 = l_2$*, but it is not provable for ilist_rel since our type theory is not extensional.*

2.3.4 Bijection Between *ilist* and Lists

In order to show that there is a bijection between *ilist* and lists, we define the two following functions that respectively transform an element of *ilist* into a list (*ilist2list*) and a list into an element of *ilist* (*list2ilist*).
To define *ilist2list*, we proceed in two steps. First we create a list containing all the "indices" of the *ilist* (i. e., containing all the elements of $Fin\ (lgti\ l)$. For example, for $lgti\ l = 2$ this list will be $[first\ 1\ ;\ succ\ (first\ 0)]$). To do so, we write the function *makeListFin*, that takes as argument a natural number n and that returns the list of all the elements of $Fin\ n$. Then, we apply the function part of the *ilist* to all the elements of this list.

Definition 8 (*makeListFin*)

$$makeListFin\ 0 :\ list\ (Fin\ 0) := []$$
$$makeListFin\ (n+1) :\ list\ (Fin\ (n+1)) := (first\ n) :: (map\ succ\ (makeListFin\ n))$$

It is easy to prove the following lemma on *makeListFin*:

Lemma 3 $\forall\ n, length\ (makeListFin\ n) = n.$

Definition 9 (*ilist2list*)

$$ilist2list\ T\ l :\ list\ T\ :=\ map\ (fcti\ l)\ (makeListFin\ (lgti\ l))$$

To define *list2ilist*, we also proceed in two steps. First we must define a method that gives us the "i^{th}" element of the list. We do not detail it here because the problems that arise are more Coq-related (technical) than theoretical. We call this function *list2FinT*. It takes a list l and an element i of $Fin\ (length\ l)$ as parameters and returns the i^{th} element of l. It is characterized by the following assertions:

$$\forall\ t\ q,\ list2FinT\ (t :: q)\ (first\ (length\ q)) = t$$
$$\forall\ t\ q\ i,\ list2FinT\ (t :: q)\ (succ\ i) = list2FinT\ q\ i$$

and it is such that the following lemmas are true:

Lemma 4

$$\forall\ (l : list\ T)\ (f : T \to U)\ (i : Fin\ (length\ (map\ f\ l))),\\ list2FinT\ (map\ f\ l)\ i = f\ (list2FinT\ l\ i'_h)$$

where h is a proof that $length\ (map\ f\ l) = length\ l$*, which is trivial.*

Lemma 5

$$\forall (l :\ ilist\ T)\ (i :\ Fin\ (length\ (makeListFin\ (lgti\ l)))),\ list2FinT\ (makeListFin\ (lgti\ l))\ i = i'_h$$

where h is a proof that $length\ (makeListFin\ (lgti\ l)) = lgti\ l$*, which is an instance of Lemma 3.*

Actually, *list2FinT* is the function part of the *ilist* we want to create in *list2ilist*. *list2ilist* is defined as follows:

Definition 10 (*list2ilist*)

$$list2ilist\ T\ l :\ ilist\ T\ :=\ \langle length\ l,\ list2FinT\ l\rangle$$

To show that there is a bijection between *ilist* and lists, we need to prove that the compositions $ilist2list \circ list2ilist$ and $list2ilist \circ ilist2list$ are both extensionally equal to the identity, i. e., only pointwise and only with respect to *ilist_rel* when comparing elements of *ilist*.
We first define the following two lemmas to help us with the proofs mentioned above. The proofs of those lemmas are straightforward and not detailed here.

Lemma 6 $\forall\ T\ l,\ lgti\ (list2ilist\ (ilist2list\ l))\ =\ lgti\ l.$

Lemma 7 $\forall\ T\ l,\ length\ (ilist2list\ (list2ilist\ l))\ =\ length\ l.$

And now we can prove the bijection between *ilist* and lists.

Theorem 1 ($list2ilist \circ ilist2list = id$) $\forall\ T\ l, ilist\_rel_{eq}\ l\ (list2ilist\ (ilist2list\ l)).$

Proof. Using Definition 6 we have that

$$ilist\_rel_{eq}\ l\ (list2ilist\ (ilist2list\ l)) \Leftrightarrow \exists h : lgti\ l = lgti\ (list2ilist\ (ilist2list\ l)),\\ \forall i : Fin\ (lgti\ l),\ fcti\ l\ i = fcti\ (list2ilist\ (ilist2list\ l))\ i'_h$$

We obtain h thanks to Lemma 6. Therefore, we now only have to prove that:

$$\forall i : Fin\ (lgti\ l),\ fcti\ l\ i = fcti\ (\ \underbrace{list2ilist\ (ilist2list\ l)}_{\langle length\ (ilist2list\ l),\ list2FinT\ (\ \underbrace{ilist2list\ l}_{map\ (fcti\ l)\ (makeListFin\ (lgti\ l))}\ )\rangle}\ )\ i'_h\\ \underbrace{list2FinT\ (map\ (fcti\ l)\ (makeListFin\ (lgti\ l)))\ i'_h}_{fcti\ l\ (\underbrace{list2FinT\ (makeListFin\ (lgti\ l))\ (i'_h)'_{h'}}_{i})}$$

where h' comes from the application of Lemma 4, and the last step involves Lemma 5. □

Remark 16 *Obviously, the statement of Theorem 1 is then also valid for any other reflexive relation R than eq since ilist_rel is monotone in its relation argument.*

Theorem 2 $(ilist2list \circ list2ilist = id) \quad \forall\, T\ l, l = ilist2list\ (list2ilist\ l)$.

Proof (by induction on l).

[Case []] Applying the definitions of *ilist2list* and *list2ilist*, we get as goal $[] = []$, which is true.

[Case $t :: q$] The induction hypothesis *IH* is $q = ilist2list\ (list2ilist\ q)$.
$ilist2list\ (list2ilist\ (t :: q))$ reduces in the following way:

$$\underbrace{(ilist2list\ (\underbrace{list2ilist\ (t :: q)}_{\langle length\ q+1,\ list2FinT\ (t :: q)\rangle}))}_{list2FinT(t :: q)\ (first\ (length\ q))\ ::\ map\ ((list2FinT\ (t :: q))\circ succ)\ (makeListFin\ (length\ q))}$$
$$map\ (list2FinT\ (t :: q))\ (\underbrace{makeListFin\ (length\ q+1)}_{first\ (length\ q)\ ::\ map\ succ\ (makeListFin\ (length\ q))})$$

For this last simplification, we have used the following property of *map*:

$$\forall\, l\ f\ g,\ map\ f\ (map\ g\ l) = map\ (f \circ g)\ l$$

According to the characterization of *list2FinT*, the first expression before :: reduces to t, and the last expression after :: finally reduces to $map\ (list2FinT\ q)\ (makeListFin\ (length\ q))$ because the result of applying $map\ f$ only depends on the extension of f (the pointwise behaviour).

We actually only need to prove that:

$$t :: q = t :: \underbrace{map\ (list2FinT\ q)\ (makeListFin\ (length\ q))}_{\underbrace{ilist2list\ \underbrace{\langle length\ q,\ list2FinT\ q\rangle}_{list2ilist\ q}}_{q\quad \text{(according to } IH\text{)}}}$$

□

This proves that there is a bijection between lists and *ilist*, and it validates our definition of *ilist*.

2.3.5 Functions on *ilist*

As we have a bijection between lists and *ilist*, we can redefine any function f that has lists as parameters and/or lists as result type. In particular that means that all the usual functions (and higher order functions) on lists have their counterpart on *ilist*. For example, the well-known *filter* function on lists has an analogue on *ilist* (for P a predicate on the type of elements):

Definition 11 $ifilter\ P\ l := list2ilist\ (filter\ P\ (ilist2list\ l))$

And in general, any function $f : list\ T \to list\ T$ can be translated to *ilist* as a function f' of type $ilist\ T \to ilist\ T$. The function f' is defined as follows :

$$f' := list2ilist \circ f \circ ilist2list$$

However, this is only anecdotal as we embed f into another function and therefore we do not solve the guardedness issue. For example, if we defined an analogue of the *map* function (let's call it *imap*) with this method, we would have:

Definition 12 (*imap*, First Try)

$$imap : (T \to U) \to ilist\ T \to ilist\ U$$
$$imap\ f\ l := list2ilist\ (map\ f\ (ilist2list\ l))$$

But this does not solve our problem since the function f (which in our example is the co-recursive call) would still be embedded into the *map* function, which as we saw does not work.

2.3.6 *imap*

We have to redefine the *map* function directly. This is actually quite easy since the part of the *ilist* that is affected by the *map* is the function part (*ilistn*). So in fact, the *imap* function is little more than a composition of functions. What we have to do is to compose the function part of the *ilist* with the function we have to apply and then recreate the *ilist*. The result has the same natural numbers part ($lgti\ l$) and a new function part $f \circ (fcti\ l)$:

Definition 13 (*imap*, Suitable for Guarded Definitions) $\quad imap\ f\ l := \langle lgti\ l,\ f \circ (fcti\ l) \rangle$

Here, the function f (and therefore in our example the co-recursive call) is directly under the constructor $\langle \ldots, \ldots \rangle$. This satisfies the guardedness restriction and solves our problem. So we see that the use of function spaces is considered less critical than the use of inductive types because they are more primitive. They are even part of the logical framework. This could not have been done on lists since they are defined inductively and so should be the functions that manipulate them. There is no other way than recursion to define *map* on lists. All such higher-order functions add a layer between the constructor and the function given as a parameter. In the case this function is a co-recursive call, it can create, as we saw, a conflict with the guardedness conditions. As the *imap* function is not defined recursively, there is no layer added and as we said, in case of a co-recursive call the guardedness condition is satisfied.

2.3.7 Universal Quantification

For further definitions (see Section 3) we need to define a property on *ilist* that expresses that all the elements of an *ilist* satisfy a predicate P. We call it *iall* (it is the counterpart to the *for_all* function in Caml). It is defined as follows:

Definition 14 (*iall*) $\quad iall\ T\ P\ l : Prop := \forall i, P\ (fcti\ l\ i)$

2.3.8 Manipulation of *ilist* in List Fashion

What we wanted to do when we created *ilist* was to have functions that would mimic list behaviour. Thus, we want to be able to manipulate *ilist* in a similar way as we manipulate lists.
There are two constructors of list: *nil* ([]) that allows to create an empty list, and *cons* (infix ::) that allows to insert an element at the head of a list. Thus, we have written the two following functions that allow, respectively, to create an empty *ilist* and to append an element at the head of an *ilist*.
To define *inil*, we need an element of *ilistn* T 0, i. e., a function of type *Fin* $0 \to T$. As *Fin* 0 is empty, all the *ilistn* T 0 are equivalent (and are inhabited for all T). Let's call *iniln* one of those. We define *inil* the following way:

Definition 15 (*inil*) $\quad inil\ T := \langle 0,\ iniln\ T \rangle$

For the sake of clarity, we will also define the function part of *icons* separately.

Definition 16 (*iconsn*)

$$\begin{aligned} &iconsn : \forall\ T\ n, T \to ilistn\ T\ n \to ilistn\ T\ (n+1) \\ &iconsn\ T\ n\ t\ ln\ (first\ n) := t \\ &iconsn\ T\ n\ t\ ln\ (succ\ n\ i') := ln\ i' \end{aligned}$$

Definition 17 (*icons*) $\quad icons\ T\ t\ l := \langle lgti\ l + 1,\ iconsn\ t\ (fcti\ l)) \rangle$

The basic notions on lists are the head and the tail. Therefore, to be able to manipulate elements of *ilist* as lists we need functions that allow us to get the head and the tail of an element of *ilist*. These functions are defined as follows:

Definition 18 (*ihead*) In this definition, the parameter t of type T represents the default element returned by *ihead* in case the *ilist* parameter is empty.

$$\begin{aligned} &ihead : \forall T, ilist\ T \to T \to T \\ &ihead\ T\ \langle 0,\ ln \rangle\ t := t \\ &ihead\ T\ \langle n+1,\ ln \rangle\ t := ln\ (first\ n) \end{aligned}$$

Definition 19 (*itail*) In case the *ilist* parameter is empty, *itail* returns an empty *ilist* (*inil*).

$$\begin{aligned} &itail : \forall T, ilist\ T \to ilist\ T \\ &itail\ T\ \langle 0,\ ln \rangle := inil\ T \\ &itail\ T\ \langle n+1,\ ln \rangle := \langle n,\ ln \circ succ \rangle \end{aligned}$$

It is easy to show, in order to validate our definitions, that:

$$\forall\ T\ l\ t, lgti\ l > 0 \to ilist\_rel_{eq}\ l\ (icons\ (ihead\ l\ t)\ (itail\ l))$$

Remark 17 (Manipulation of lists vs. manipulation of *ilist*) *The functions defined previously allow to manipulate elements of ilist in a list way. However, this manipulation is not really well*

suited for ilist, although one is usually more used to it. Indeed, the notions of head and tail, basic for list, are not well adapted for ilist. Elements of ilist are basically functions and it is not easier to get the first element than any other. Actually, the basic notion on ilist is the notion of i^{th} element while it is not a basic notion on lists. In the same fashion, adding an element to an ilist (icons) is a quite complex operation, while on lists it is done by a constructor.
The conclusion of this is that even though lists and ilist are equivalent, the respective natural ways to manipulate them are quite different.

Remark 18 (Notation) *In the rest of this paper, we will use list-like notations for elements of ilist . In particular, we will write* $[\![]\!]$ *for inil and* $[\![x;y;z;...]\!]$ *for successive applications of icons to inil.*

3 The Refined Definition of Graph Representation

Now, we can redefine the type *Graph* using *ilist* and define various functions and properties on it.

3.1 Definitions of *Graph* and *applyF2G*

The definition of *Graph* is identical to the previous one, except that lists are replaced by *ilist*. We define it through the following constructor:

Definition 20 (*Graph*, Viewed Coinductively) $mk\_Graph : T \to ilist\ (Graph\ T) \to Graph\ T$

Now we can define the function *applyF2G* so that it respects the guardedness condition:

Definition 21 $applyF2G\ f\ (mk\_Graph\ t\ l) := mk\_Graph\ (f\ t)\ \big(imap\ (applyF2G\ f)\ l\big)$

We call *label* and *sons* the two functions on *Graph* that return respectively the element of T and the *ilist* part of a *Graph*. They are such that the following lemma is correct:

Lemma 8 $\forall g,\ g = mk\_Graph\ (label\ g)\ (sons\ g)$.

Remark 19 *Here we have the right to use Leibniz equality to compare two elements of Graph as they are definitionally equal for any g of the form mk_Graph t l (and not only bisimulated).*

We can redefine Example 2 and Example 3 with our new definition of *Graph* using the notations introduced in Remark 18.

Example 5 (Redefinition of *Finite_Graph*)

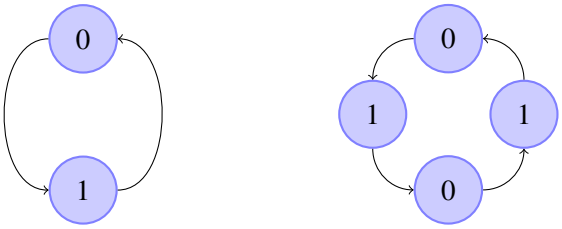
$$Finite\_Graph := mk\_Graph\ 0\ [\![mk\_Graph\ 1\ [\![Finite\_Graph]\!]]\!]$$

Example 6 (Redefinition of $Infinite\_Graph_n$)

$$Infinite\_Graph_n := mk\_Graph\ n\ [\![Infinite\_Graph_{n+1}]\!]$$

3.2 An Equivalence on *Graph*

We can also define all the other tools we need. In particular, we can define an equivalence relation on *Graph*. Indeed, as elements of *Graph* are coinductive, Leibniz equality cannot be used here (it is just too fine-grained and cannot be established by coinduction), we need bisimulation. To illustrate this, Figure 4 shows a situation where two graphs are not Leibniz equal while we want them to be equivalent. Indeed, those two graphs are different (graphically this is evident) but they unfold into the same infinite tree. So they actually represent the same element but have different (syntactic) representations. If one wanted to differentiate the 0 (resp. 1) nodes, one would have to use a richer type than only natural numbers.



(a) Example of Figure 1

(b) Example of Figure 1 unfolded once

Figure 4: Example graphs that are equivalent but not equal

To relate two elements of *Graph*, we need (as we did for *ilist*) to relate their two parts. The label part is compared through an equivalence relation R on type T. For the sons part, that is represented by an *ilist*, we will use the equivalence relation defined on *ilist*: *ilist_rel* (see Subsubsection 2.3.3). As the type parameter for the *ilist* is itself *Graph*, *ilist_rel* needs the equivalence relation on *Graph* as argument. So this relation must be defined coinductively. Finally, we can define the equivalence relation on *Graph* (let's call it *Geq*) as follows (defined co-inductively):

Definition 22 (*Geq*)

$$\forall\ T\ R\ g_1\ g_2,\ Geq_R\ g_1\ g_2 \Leftrightarrow R\ (label\ g_1)\ (label\ g_2)\ \wedge\ ilist\_rel_{Geq_R}\ (sons\ g_1)\ (sons\ g_2)$$

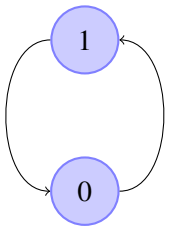
It is possible to show that *Geq* is an equivalence relation using the same style of reasoning as for *ilist_rel*.

Example 7 (Equivalence of the graphs of Figure 4) *The graph of Figure 4(b) can be defined in our definition of Graph as follows:*

$$Finite\_Graph\_Unfolded :=$$
$$mk\_Graph\ 0\ [\![mk\_Graph\ 1\ [\![mk\_Graph\ 0\ [\![mk\_Graph\ 1\ [\![Finite\_Graph\_Unfolded]\!]]\!]]\!]\!]$$

It is easy to show Geq_{eq} *Finite_Graph Finite_Graph_Unfolded. The proof is a simple coinduction. The definition of Finite_Graph only has to be unfolded once.*

Remark 20 *To be equivalent, two elements of Graph have to be constructed in the same way. Therefore, the two graphs of Figure 1 and Figure 5 are not equivalent even though we might wish them to be, if we disregard roots. We are working on the design of a coarser relation for this purpose.*



This graph is represented by the following expression using Definition 20:
$Finite\_Graph' := mk\_Graph\ 1\ [\![mk\_Graph\ 0\ [\![Finite\_Graph']\!]]\!]$

Figure 5: Other representation of the graph of Figure 1, disregarding roots

3.3 Universal Quantification on *Graph*

As we did with *ilist* (see Subsubsection 2.3.7), we define a property of universal quantification on *Graph*. It will be useful, in particular in Subsection 3.4. This property expresses that a predicate $P : Graph\ T \to Prop$ on *Graph* is satisfied by an element g of *Graph* and all its descendants (sons, sons of its sons, and so on). As *Graph* is co-inductive, this property must be defined co-inductively too. We call it *G_all* and it is defined as follows:

Definition 23 (*G_all*) $\forall\ P\ g,\ G\_all\ P\ g \Leftrightarrow P\ g\ \wedge\ iall\ (G\_all\ P)\ (sons\ g)$

3.4 Finiteness of *Graph*

Another interesting property on *Graph* is finiteness. It would be interesting for example to prove that Example 2 and Example 3 indeed are respectively finite and infinite. Saying that an element g of *Graph* is finite means that it contains a finite number of elements of *Graph*, up to bisimilarity. This can be expressed by the fact that all the elements of *Graph* contained in g can fit into a finite list. This is the way we choose to define the finiteness of a *Graph*. We call the finiteness property *G_finite*. To define it, we need a predicate (let's call it *element_of*) to check whether an element g of *Graph* is included in a list of graphs. By included we mean that there is an element of the list that is related through bisimulation (Geq_R for the chosen R) with g. We use $\in$ to say that an element is in a list.

Definition 24 $element\_of_R\ l\ g := \exists y,\ y \in l\ \wedge\ Geq_R\ g\ y$

Thanks to it we can define *G_finite*.

Definition 25 (*G_finite*) $\forall g,\ G\_finite_R\ g :\Leftrightarrow \exists l,\ G\_all\ (element\_of_R\ l)\ g$

3.5 Proofs of Finiteness and Infiniteness

We want here to prove that Example 2 and Example 3 are respectively finite and infinite. The equivalence relation on labels used here is Leibniz equality as the labels are natural numbers. First we prove that Example 2 is finite.

Lemma 9 (*Finite_Graph* Is Finite) $\quad G\_finite_{eq}\ Finite\_Graph.$

Proof (by co-induction). The proof here is quite easy. We must give a list l with all the elements of *Graph* contained in *Finite_Graph* and show that they are actually all included in l. There are only two elements of *Graph* contained in *Finite_Graph*: *Finite_Graph* itself and $mk\_Graph\ 1\ [\![Finite\_Graph]\!]$. The provided list is: $[Finite\_Graph\ ;\ mk\_Graph\ 1\ [\![Finite\_Graph]\!]]$. Now, we only have to prove that *Finite_Graph* is contained in the list (but it was designed for it!); that its sons are (it only has one son: $mk\_Graph\ 1\ [\![Finite\_Graph]\!]$, so it is in the list) and that the sons of its son are in the list too (this is *Finite_Graph* itself, so we use the co-inductive hypothesis). □

Similarly, we want to prove that $Infinite\_Graph_n$ is not finite.

Lemma 10 ($Infinite\_Graph_n$ Is Infinite) $\quad \forall n,\ \neg\ G\_finite_{eq}\ Infinite\_Graph_n.$

To prove this we prove a general lemma that we will then instantiate to say that if the *Graph* is finite then its labels are bounded (Lemma 12). The general lemma says that for any function f of type $Graph\ T \to nat$ and for any element g of *Graph*, if g is finite then the image of the set of nodes of g by f is bounded. Actually, to prove this lemma we need a property that says, basically, that f is a morphism, i. e., that $\forall\ g_1\ g_2,\ Geq_R\ g_1\ g_2 \Rightarrow f\ g_1 = f\ g_2$. We abbreviate this property $Morph_R(f)$.

Lemma 11 $\quad \forall\ f\ g,\ Morph_R(f) \wedge G\_finite_R\ g \Rightarrow \exists m,\ G\_all\ (\lambda g'.\ f\ g' \leq m)\ g.$

Proof. The proof here is based on a simple idea: as g is finite, there exists a list l containing all the nodes of g (see Definition 25). Therefore, the image of the set of nodes of g by f can actually be represented by $map\ f\ l$ (possibly containing duplicates). As l is finite, $map\ f\ l$ also is. What is more, $map\ f\ l$ has type $list\ nat$, therefore, the values contained in it are bounded. We call max the maximum. As all the nodes contained in g are also in l, we can show (by co-induction on g) that $G\_all\ (\lambda g'.\ f\ g' \leq max)\ g$. □

Now, we can prove the following lemma (for $T = nat$):

Lemma 12 $\quad \forall g, G\_finite_{eq}\ g \Rightarrow \exists m,\ G\_all\ (\lambda x.\ label\ x \leq m)\ g.$

Proof. This lemma is actually just an instantiation of Lemma 11 with $f = label$. We directly have the property that $\forall\ g_1\ g_2,\ Geq_{eq}\ g_1\ g_2 \Rightarrow f\ g_1 = f\ g_2$ thanks to the definition of *Geq*. □

Now, to prove Lemma 10, we only have to prove that the labels of $Infinite\_Graph_n$ are unbounded and we will have the result simply using Lemma 12. To prove that the labels of $Infinite\_Graph_n$

are unbounded, we show that $\forall m, m \geq n \Rightarrow \mathit{Infinite\_Graph}_m \subseteq \mathit{Infinite\_Graph}_n$.

Remark 21 *We informally use the notation* $\subseteq$ *to say that a Graph is included in another.*

With this, it is easy to show that the labels are unbounded (since the first label of $\mathit{Infinite\_Graph}_n$ is n).

Remark 22 *In a similar way, we have that if the* number of sons *in a Graph is unbounded, then the Graph is infinite. However, it is also possible to construct elements of Graph in which the out-degree of a node is bounded and so are the labels and that are still infinite, see Figure 6 for an example. Here, the proof of infiniteness is much more difficult (it is part of [PM11]).*

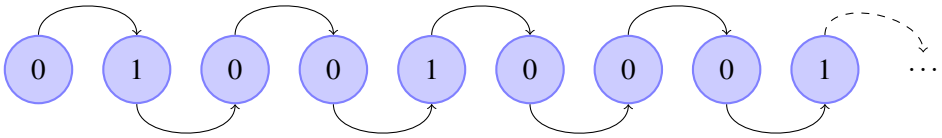


Figure 6: Example of an infinite graph with bounded number of sons and bounded labels

3.6 *Graph* in *Graph*

We will need to represent the property asserting that an element g_{in} of *Graph* is (strictly) included in another element g_{out} of *Graph* (see Subsection 3.7). We split the situation into two different cases: g_{in} is part of *sons* g_{out} or g_{in} is included in one of g_{out}'s sons. In Coq, the following definition is represented as an inductive property with two constructors.

Definition 26 (GinG)

$$\forall\ T\ R\ g_{in}\ g_{out},\ \mathit{GinG}_R\ g_{in}\ g_{out} \Leftrightarrow \begin{cases} \exists i,\ \mathit{Geq}_R\ g_{in}\ (\mathit{fcti}\ (\mathit{sons}\ g_{out})\ i) & \mathit{or} \\ \exists i,\ \mathit{GinG}_R\ g_{in}\ (\mathit{fcti}\ (\mathit{sons}\ g_{out})\ i) \end{cases}$$

We can prove that GinG_R is transitive if R is transitive.

3.7 Cycles in *Graph*

It may also be useful to define a property about the existence of a cycle in an element of *Graph*. To do so, we use the property *GinG* defined above.
First of all, we define a property saying that an element g of *Graph* is itself a cycle (i. e., there is a non-empty path from the root to the root). This means that g is itself included in g. Therefore the definition of *isCycle* is straightforward.

Definition 27 (*isCycle*) $\forall\ T\ R\ g,\ \mathit{isCycle}_R\ g \Leftrightarrow \mathit{GinG}_R\ g\ g$

Using this definition, it is easy to define the property of existence of a cycle in an element g of *Graph*. Just as we did for *GinG*, we divide the property into two cases. Either g is a cycle or one

element of *sons g* has a cycle. As before, in Coq this is defined through two constructors of an inductive definition.

Definition 28 (*hasCycle*) $\forall\ T\ R\ g,\ hasCycle_R\ g \Leftrightarrow \begin{cases} isCycle_R\ g & or \\ \exists i,\ hasCycle_R\ (fcti\ (sons\ g)\ i) \end{cases}$

For a finite element of *Graph*, it is quite easy to prove the existence or non-existence of a cycle (for example, it is straightforward to prove that Example 5 has a cycle). However, if there are many nodes, the proof might be long. Indeed, the proofs are constructive, that is, one will have to exhibit the cycle to prove that it exists or to look into each different path to show that there is none. This last operation may be tedious.

4 Towards Metamodel Representation

As we have explained in Section 1, our final goal is to represent metamodels and then perform transformations on these models. Until now, we only have shown graph representation. It is a first step towards metamodel representation, but metamodels have other properties that we can not represent with *Graph* as it is. We present here two extensions of our development that bring us closer to metamodel representation. The first one is a representation of non-connected graphs and the second one a representation of multiplicities. Another problem that arises and that is not treated here is the representation of inheritance.

4.1 A Representation of a Wider Class of Graphs

As we have said in Remark 3, the type *Graph* only represents single-rooted connected graphs. However, there might be some cases where a more general graph representation could be necessary. For instance, a metamodel may not be connected or not single rooted. We present here a graph representation that allows to represent non-connected and non-single-rooted graphs.

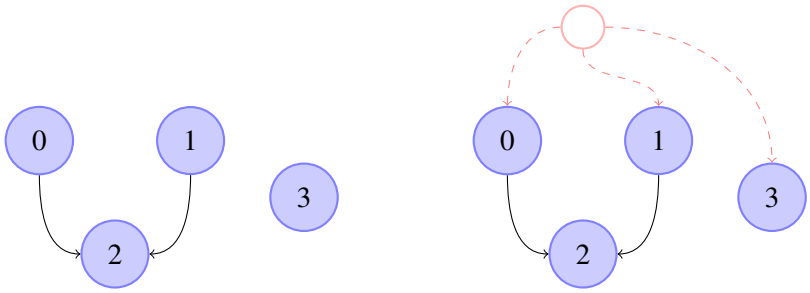
The idea is to add fictitious nodes to our previous definition of *Graph* in order to be able to represent in only one structure non-connected graphs.

Example 8 *To illustrate this, and before giving the formal definition, here is an example of what we might want to represent (Figure 7(a)) and an example of its representation with fictitious nodes (Figure 7(b)).*

We are going to represent those fictitious nodes with the usual *option* type of functional languages. When the label associated to a node is of the form *Some t*, the node is a "real" node (and its label is *t*) and when it is of the form *None*, it is a fictitious node. Therefore, we see that we can actually represent this extension of *Graph* as an instance of *Graph*.

Definition 29 (*AllGraph*) $AllGraph\ T := Graph\ (option\ T)$

Example 9 (Representation of Example 8) *The graph of Figure 7(b) is represented as follows*



(a) Graph to represent (b) Symbolic representation

Figure 7: Representation of a non-connected graph

with Definition 29:

$$\begin{array}{l} \mathit{let}\ g_2 : \mathit{AllGraph}\ \mathit{nat} := \mathit{mk\_Graph}\ (\mathit{Some}\ 2)\ [\![]\!]\ \mathit{in} \\ \quad \mathit{mk\_Graph}\ \mathit{None}\ [\![\mathit{mk\_Graph}\ (\mathit{Some}\ 0)\ [\![g_2]\!]\ ;\ \mathit{mk\_Graph}\ (\mathit{Some}\ 1)\ [\![g_2]\!]\ ; \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{mk\_Graph}\ (\mathit{Some}\ 3)\ [\![]\!]]\!] \end{array}$$

As we use the previous definition of *Graph* to define *AllGraph*, all the definitions and lemmas on *Graph* are still valid on *AllGraph*. For instance, we can use *Geq* to compare two elements of *AllGraph*, for which we typically lift the relation R on T canonically to a relation on $option\ T$.

4.2 Multiplicity

In this section, we present an extension of *ilist* to represent multiplicities in metamodels representation. For this, we have extended the concept of *ilist* to take into account multiplicity, i. e., an interval constraint on the out-degree.

Remark 23 *Here we deviate a little from the graph representation of this article. Indeed, in graphs (at least as we have represented them here), the labels of all the nodes have the same type. But in metamodel representation, this is not true. That is why we cannot use Graph in the rest of this section.*

4.2.1 Presentation of the Problem on an Example

To illustrate the usefulness of multiplicities, consider the following example of Figure 8.
To represent this using the tools we currently have at our disposal, we would define A, B and C (simultaneously) as follows (these definitions are to be interpreted co-inductively):

$$\begin{array}{l} \mathit{mk\_A} : \mathit{ilist}\ B \to \mathit{ilist}\ C \to A \\ \mathit{mk\_B} : C \to B \\ \mathit{mk\_C} : B \to B \to C \end{array}$$

Here, various problems arise:

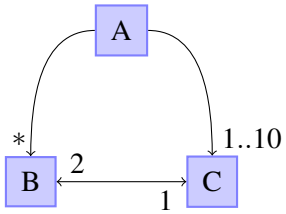


Figure 8: Example of a metamodel with multiplicities

- to represent the 1..10 multiplicity of the edge A to C, we might explicitly enumerate all possibilities with increasing numbers of arguments but this is heavy for large numbers and even impossible for indeterminate bounds. Since list is not an option, we see no other good choice than using *ilist*. However, we lose the bounds (taking *ilist C* is equivalent to a multiplicity $*$) and therefore, information,
- the representation is not homogeneous. Here, there are two different ways to represent an edge with multiplicity:
  - an *ilist* for a variable multiplicity (e. g., $*$ or 1..10)
  - a sequence of $T \to T \to \ldots \to T$ for a fixed multiplicity (e. g., 2)

Hence, an extension of *ilist* that allows to take into account multiplicities would solve these two issues. This is what we present now.

4.2.2 Implementation of Multiplicities

First we need a property (let's call it *PropMult*) to say whether a number is between the two specified bounds of the multiplicity condition. The inferior bound (let's call it *inf*) always exists (it can be 0 but always has a value). Therefore it has type *nat*. On the opposite, the superior bound (let's call it *sup*) may not exist (multiplicity "$*$"). Therefore it has type *option nat* (constructor *Some* if it exists, constructor *None* if not). The property is expressed as follows:

Definition 30 (*PropMult*) $\forall\ inf\ sup\ k$,

[Case $sup = Some\ s$**]** $k \geq inf \wedge k \leq s$ **[Case** $sup = None$**]** $k \geq inf$

Thanks to this property we can refine our *ilistn* (that was the set of functions of type $Fin\ n \to T$) to keep only the ones whose n satisfies *PropMult*.

Definition 31 $ilistnMult\ T\ inf\ sup\ n := \{ln : ilistn\ T\ n \mid PropMult\ inf\ sup\ n\}$

Remark 24 *Elements of ilistnMult are pairs formed by an element of ilistn and a proof of PropMult inf sup n, hence the type is empty if PropMult inf sup n does not hold.*

Using *ilistnMult*, the definition of *ilistMult* (the counterpart of *ilist* with multiplicity) is straightforward. It is the same as the definition of *ilist* (see Subsubsection 2.3.2) but using *ilistnMult*.

Definition 32 $ilistMult\ T\ inf\ sup := \Sigma n : nat.\, ilistnMult\ T\ inf\ sup\ n$

We can define a relation and functions on *ilistMult* very much the same way as we did on *ilist*. Therefore we do not present them here again.
We can also show that there is a bijection between $ilistMult\ T\ 0\ None$ and list (we do it the same manner we did for *ilist*, defining *ilistMult2list* and *list2ilistMult* and showing that their compositions are extensionally equal to the identity).

Remark 25 *The multiplicities* 0 *and None are explained by the fact that a list may have no element (empty list, so* $inf = 0$*) or a finite but unbounded number of elements (i. e., multiplicity "*$*$*", so* $sup = None$*).*

Combining the lemmas about bijection between lists, *ilist* and *ilistMult*, we obtain that there is a bijection between $ilist\ T$ and $ilistMult\ T\ 0\ None$.

The important result is that all definitions written with $ilist\ T$ can be written equivalently with $ilistMult\ T\ 0\ None$. In particular, the following definition of *GraphMult* is equivalent to *Graph*:

Definition 33 (*GraphMult*) $mk\_GraphMult : T \to ilistMult\ Graph\ 0\ None \to Graph$

With this definition of *ilistMult*, the example of Figure 8 would be represented as follows (*A*, *B* and *C* are still defined simultaneously and co-inductively):

$$mk\_A : ilistMult\ B\ 0\ None \to ilistMult\ C\ 1\ (Some\ 10) \to A$$
$$mk\_B : ilistMult\ C\ 1\ (Some\ 1) \to B$$
$$mk\_C : ilistMult\ B\ 2\ (Some\ 2) \to C$$

These definitions are homogenous and complete (no information is lost).

5 Related Work and Conclusion

The work presented here shares concerns with other work. Among them, we can cite the work by Bertot and Komendantskaya in [BK08]. In their paper they treat the problem of representing streams as functions, to overcome guardedness issues in Coq. The main difference is that we need a finite definition set (*Fin n*) whereas they can just use *nat*. Recall that our problem was with the embedded *inductive* type of lists and not the co-inductive streams. In [Dam10], Dams proposes an alternative solution to our problem in Coq. He defines everything co-inductively (so instead of lists, he has streams of sons) and then restricts what needs to be finite by a property of finiteness. In that approach, programming is done with a bigger datatype and the proofs have to be carried out for the "good" elements. In [Niq08], Niqui describes a general solution for the representation of bisimulation in Coq using category theory. However, as we tried to apply his theory, it seemed that only co-inductive embedded types could be treated (streams but not lists) with the given solution. Moreover, it did not seem possible to parameterize the bisimulation by an equivalence relation over the types of the elements.
Coq is not the only proof assistant to have guardedness issues. For example, they are present

in Agda[3], another proof assistant based on predicative type theory. We studied the way guardedness issues are addressed in Agda. Danielsson describes it in [Dan10] (see also the extended case study with Altenkirch in [DA10]). The solution used is to redefine the types (for example the types of lists) adding a constructor for each problematic function (for example, *map*). However, this is based on a mixture of inductive and co-inductive constructors for a single datatype definition, which is not admissible in Coq and of experimental status even in Agda.
About graph representation in functional languages, we can mention the work by Erwig. [Erw01] proposes a way to represent directed graphs using inductive types, where, in the inductive step, a new node is added, together with all its edges to and from previously introduced nodes. Being "new" or "previously introduced" is not part of the inductive specification but only of a more refined implementation. Moreover, there is no certification of these invariants for graph algorithms, although this might be interesting future work in expressive systems such as Coq. However, the main conceptual difference to our work is that in his representation, all nodes are represented at the same level (they are more or less elements of a list) while we actually wanted, for our own needs, to build into the construction navigability through the graph, including its loops.

To conclude, in this paper, we have developed a complete solution to overcome Coq's guardedness condition when mixing the inductive type of lists with co-inductive types. The Coq development corresponding to this work is available in [PM11]. This framework can be extended with new features as needed. For the results we wanted to obtain, it worked well. Clearly, it would have been easier if a more refined guardedness criterion had been available in Coq but the last ten years have shown that getting the criterion right is a quite subtle issue. Another solution would have been to use another proof assistant instead of Coq. Nevertheless, for further developments, we needed a system based on type theory. Furthermore, despite the guardedness restrictions, coinductive types in Coq are quite practical to use. They are an addition to the original CIC, hence a part of the kernel. As such, they are subject to discussion about justification and optimization, and there is quite some on-going scientific work around them in the Coq community.

However, we would now be interested in a more general solution to overcome the guardedness condition with any embedded inductive type (not only lists). But we realized that to do so, we needed to be more abstract. In particular, we think that we could draw inspiration from category theory. The work by Niqui in [Niq08] might be a good start.

Moreover, the work we present has to be seen as part of a larger project where we are interested in a co-inductive representation of metamodels (see Subsection 4.2). We have solved the problem of multiplicity and non-connected graphs but the problem of inheritance/subtyping remains. This is difficult since we look for an extensible way to represent metamodels (they may vary over time). Poernomo's work on type theory for metamodels in [Poe08] is relevant here. The work by Boulmé on FOCAL [Bou00] that has been realized with Coq, will probably help in treating the inheritance problem. Finally, and still within the aim of representing metamodels, we are working on the design of a more liberal equivalence relation on *Graph*.

Acknowledgements: This development was initiated by the original idea of Jean-Paul Bodeveix to use *ilist* to overcome the guardedness condition. We are grateful for several interesting suggestions by Silvano Dal Zilio and for the careful reading of a preliminary version by Martin

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

Strecker. We are grateful for the feedback we got for the preliminary version presented at the workshop GCM'10 [PM10].

Bibliography

[Alt93] T. Altenkirch. A Formalization of the Strong Normalization Proof for System F in LEGO. In Bezem and Groote (eds.), *Typed Lambda Calculi and Applications, International Conference, TLCA 1993*. Lecture Notes in Computer Science 664, pp. 13–28. Springer, 1993.

[BDd09] S. Berardi, F. Damiani, U. de'Liguoro (eds.). *Types for Proofs and Programs, International Conference, TYPES 2008, Torino, Italy, March 26-29, 2008, Revised Selected Papers*. Lecture Notes in Computer Science 5497. Springer, 2009.

[Bir01] R. S. Bird. Maximum marking problems. *J. Funct. Program.* 11(4):411–424, 2001.

[BK08] Y. Bertot, E. Komendantskaya. Using Structural Recursion for Corecursion. Pp. 220–236 in [BDd09].

[Bou00] S. Boulmé. Specifying in Coq inheritance used in Computer Algebra. Research report, LIP6, 2000. Available on www.lip6.fr/reports/lip6.2000.013.html.

[Chl10] A. Chlipala. Posting to Coq club in the thread "Is Coq being too conservative?". January 2010.
http://logical.saclay.inria.fr/coq-puma/messages/d71fd3954d860d42#msg-285229ea3f28adef

[Coq93] T. Coquand. Infinite Objects in Type Theory. In Barendregt and Nipkow (eds.), *Types for Proofs and Programs, International Conference, TYPES 1993*. Lecture Notes in Computer Science 806, pp. 62–78. Springer, 1993.

[DA10] N. A. Danielsson, T. Altenkirch. Subtyping, Declaratively. In Bolduc et al. (eds.), *Mathematics of Program Construction (MPC'10)*. Lecture Notes in Computer Science 6120, pp. 100–118. Springer, 2010.

[Dam10] C. Dams. Posting to Coq club in the thread "Is Coq being too conservative?". January 2010.
http://logical.saclay.inria.fr/coq-puma/messages/d71fd3954d860d42#msg-7946fd74eb4de604

[Dan10] N. A. Danielsson. Beating the Productivity Checker Using Embedded Languages. In Bove et al. (eds.), *Proceedings Workshop on Partiality and Recursion in Interactive Theorem Provers*. EPTCS 43, pp. 29–48. 2010.

[Erw01] M. Erwig. Inductive graphs and functional graph algorithms. *J. Funct. Program.* 11(5):467–492, 2001.

[GC07] E. Giménez, P. Castéran. A Tutorial on [Co-]Inductive Types in Coq. 2007. www.labri.fr/perso/casteran/RecTutorial.pdf

[MM04] C. McBride, J. McKinna. The view from the left. *J. Funct. Program.* 14(1):69–111, 2004.

[Niq08] M. Niqui. Coalgebraic Reasoning in Coq: Bisimulation and the lambda-Coiteration Scheme. Pp. 272–288 in [BDd09].

[PM10] C. Picard, R. Matthes. Coinductive graph representation: the problem of embedded lists. In Echahed et al. (eds.), *GCM 2010, The Third International Workshop on Graph Computation Models*. Pp. 133–147. 2010. Online available at the workshop's website http://gcm2010.imag.fr/.

[PM11] C. Picard, R. Matthes. Formalization in Coq for this article. 2011. www.irit.fr/~Celia.Picard/Coq/Coind_Graph/.

[Poe08] I. Poernomo. Proofs-as-Model-Transformations. In Vallecillo et al. (eds.), *International Conference on Model Transformation, ICMT 2008*. Lecture Notes in Computer Science 5063, pp. 214–228. Springer, 2008.

[TCDT] The Coq Development Team. The Coq Proof Assistant Reference Manual. http://coq.inria.fr