Acyclicity and Finite Linear Extendability: a Formal and Constructive Equivalence.

Stéphane Le Roux

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

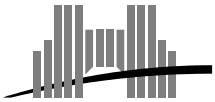
Acyclicity and Finite Linear Extendability: a Formal and Constructive Equivalence

Stéphane Le Roux

Mar 2007

Research Report N° 2007-14

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`



Acyclicity and Finite Linear Extendability: a Formal and Constructive Equivalence

Stéphane Le Roux

Mar 2007

Abstract

Linear extension of partial orders emerged in the late 1920's. Its computer-oriented version, *i.e.*, topological sorting of finite partial orders, arose in the late 1950's. However, those issues have not yet been considered from a viewpoint that is both formal and constructive; this paper discusses a few related claims formally proved with the constructive proof assistant Coq. For instance, it states that a given decidable binary relation is acyclic and equality is decidable on its domain *iff* an irreflexive linear extension can be computed uniformly for any of its finite restriction. A detailed introduction and proofs written in plain English shall help readers who are not familiar with constructive issues or Coq formalism.

Keywords: Binary relation, finite restriction, linear extension, (non-)uniform computability, topological sorting, constructivism, induction, proof assistant.

Résumé

Ceci est le resumé en français

Mots-clés: Relation binaire, restriction finie, extension linéaire, calculabilité (non-)uniforme, tri topologique, constructisme, récurrence, assistant à la preuve.

1 Introduction

This section adopts an approach technical and historical. It presents three issues: the main ingredients of the proof assistant Coq, namely inductive methods, constructivism, the Curry-De Bruijn-Howard correspondence, and constructive proof assistant in general; the notions of computability and linear extension, both involved in the results proved in Coq and discussed in this paper; the main results and the contents of the paper.

1.1 A Historical View on Inductive Methods

Acerbi [2] identifies the following three stages in the history of proof by induction. First, an early intuition can be found in Plato's Parmenides. Second, in 1575, Maurolico [11] showed by an inductive argument that the sum of the first n odd natural numbers equals n^2. Third, Pascal seems to have performed fully conscious inductive proofs. Historically, definitions by induction came long after proofs by induction. In 1889, even though the Peano's axiomatization of the natural numbers [14] referred to the successor of a natural, it was not yet an inductive definitionbut merely a property that had to hold on *pre-existing* naturals. Early XXth century, axiomatic set theory enabled inductive definitions of the naturals, like von Neumann [19], starting from the empty set representing zero. Beside the natural numbers, other objects also can be inductively/recursively defined. According to Gochet and Gribomont [7], primitive recursive functions were introduced by Dedekind and general recursive functions followed works of Herbrand and Gödel; since then, it has been also possible to define sets by induction, as subsets of known supersets. However, the inductive definition of objects from scratch, *i.e.*, not as part of a greater collection, was mainly developed through recursive types (*e.g.*, lists or trees).

1.2 Constructivism in Proof Theory

Traditional mathematical reasoning is ruled by *classical logic*. First attempts to formalize this logic can be traced back to ancient Greeks like Aristotle [4] who discussed the principle of *proof by contradiction* among others: to prove a proposition by contradiction, one first derives an absurdity from the denial of the proposition, which means that the proposition can *not not* hold. From this, one eventually concludes that the proposition must hold. This principle is correct with respect to classical logic and it yields elegant and economical proof arguments. For example, a proof by contradiction may show the existence of objects complying with a given predicate without exhibiting a constructed witness: if such an object can *not not* exist then it must exist. At the beginning of the XXth century, many mathematicians started to think that providing an actual witness was a stronger proof argument. Some of them, like Brouwer, would even consider the proof by contradiction as a wrong principle. This mindset led to *intuitionistic logic* and, more generally, to *constructivist logics* formalized by Heyting, Gentzen, and Kleene among others. Instead of the principle of proof by contradiction, intuitionists use a stricter version stating only that an absurdity implies anything. Intuitionistic logic is smaller than classical logic in the sense that any intuitionistic theorem is also a classical theorem, but the converse does not hold. In [18], a counter-example shows that the intermediate value theorem is only classical, which implies the same for the Brouwer fixed point theorem. The principle of excluded middle states that any proposition is either "true" or "false". It is also controversial and it is actually equivalent, with respect to intuitionistic

logic, to the principle of proof by contradiction. Adding any of those two principles to the intuitionistic logic yields the classical logic. In this sense, each of those principles captures the difference between the two logics.

1.3 The Curry-De Bruijn-Howard Correspondence

Nowadays, intuitionistic logic is also of interest due to practical reasons: the *Curry-De Bruijn-Howard* correspondence identifies intuitionistic proofs with functional computer programs and propositions with types. For example a program f of type $A \to B$ is an object requiring an input of type A and returning an output of type B. By *inhabiting* the type $A \to B$, the function f is also a proof of "A implies B". This vision results from many breakthroughs in proof and type theories: type theory was first developed by Russell and Whitehead in [20] in order to cope with paradoxes in naive set theory. People like Brouwer, Heyting, and Kolmogorov had the intuition that a proof was a method (or an algorithm, or a function), but could not formally state it at that time. In 1958, Curry saw a connection between his combinators and Hilbert's axioms. Later, Howard [16] made a connection between proofs and lambda terms. Eventually, De Bruijn [13] stated that the type of a proof was the proven proposition.

1.4 Constructive Proof Assistants

The Curry-De Bruijn-Howard Correspondence led to rather powerful *proof assistants*. Those pieces of software verify a proof by checking whether the program encoding the proof is well-typed. Accordingly, proving a given proposition amounts to providing a program of a given type. Some basic proof-writing steps are automated but users have to code the "interesting" parts of the proofs themselves. Each single step is verified, which gives an additional guarantee of the correctness of a mathematical proof. Of course this guarantee is not absolute: technology problems (such as software or hardware bugs) may yield validation of a wrong proof and human interpretations may also distort a formal result. Beside level of guarantee, another advantage is that a well-structured formal proof can be translated into natural language by mentioning all and only the key points from which a full formal proof can be easily retrieved. Such a reliable summary is usually different from the sketch of a "proof" that has not been actually written. An advantage of intuitionistic logic over classical logic is that intuitionistic proofs of existence correspond to search algorithms and some proof assistants, like *Coq*, are able to automatically extract an effective search program from an encoded proof, and the program is certified for free. See the Coq website [1] and the book by Bertot and Casteran [5].

1.5 Decidability and Computability

In the middle of the 1930's, Church introduced the lambda calculus, and Turing and Post independently designed their very similar machines. Those three notions are by some means equivalent models for computer programs. A question is said to be decidable if there exists a computer program (equivalently lambda term or Post-Turing machine) requiring the parameters of the questions as input, returning (within finite time) "yes" or "no" as output, and thus correctly answering the question. In this way, a binary relation over a set is said to be decidable if there exists a program expecting two elements in that set and returning "yes" if they are related or "no" if they are not. A sister notion is that of computable (or recursive) function, *i.e.*, mathematical function the images of which are computable within finite time

by the same computer program although the domain and the codomain of the function may be infinite. Note that computability with two-element-codomain functions amounts to decidability. If several computable functions can be all computed by the same program, then these functions are said to be uniformly computable.

1.6 Transitive Closure, Linear Extension, and Topological Sorting

The calculus of binary relations was developed by De Morgan around 1860. The notion of transitive closure of a binary relation (smallest transitive binary relation including a given binary relation) was defined in different manners by different people about 1890. See Pratt [15] for a historical account. In 1930, Szpilrajn [17] proved that, assuming the axiom of choice, any partial order has a linear extension, *i.e.*, is included in some total order. The proof invokes a notion close to transitive closure. Szpilrajn acknowledged that Banach, Kuratowsky, and Tarski had found unpublished proofs of the same result. In the late 1950's, The US Navy [3] designed PERT (Program Evaluation Research Task or Project Evaluation Review Techniques) for management and scheduling purposes. This tool partly consists in splitting a big project into small jobs on a chart and expressing with arrows when one job has to be done before another one can start up. In order to study the resulting directed graph, Jarnagin [12] introduced a finite and algorithmic version of Szpilrajn's result. This gave birth to the widely studied topological sorting issue, which spread to the industry in the early 1960's (see [10] and [8]). Some technical details and computer-oriented examples can be found in Knuth's book [9].

1.7 Contribution

This paper revisits a few folklore results involving transitive closure, excluded middle, computability, linear extension, and topological sorting. Most of the properties are logical equivalences instead of one-way implications, which suggests maximal generality. Claims have been fully formalized (and proved) in Coq and then slightly modified in order to fit in the Coq-related CoLoR library [6]. This paper follows the structure of the underlying Coq development but some straightforward results are omitted. Arguments are constructive (therefore also classical) and usually simple. This paper is meant to be read by a mathematician who is not familiar with constructivism. Concepts specific to Coq are introduced before they are used. Formal definitions and results are stated in a light Coq formalism that is very close to traditional mathematics and slightly different from the actual Coq code in order to ease the reading. Proofs are mostly written in plain English. The main result in this paper relies on an intermediate one, and is itself invoked in a game theoretic proof (in Coq) not published yet.

In this paper, a binary relation over an arbitrary set is said to be middle-excluding if for any two elements in the set, either they are related or they are not. The intermediate result of this paper implies that in an arbitrary set with decidable (resp. middle-excluding) equality, a binary relation is decidable (resp. middle-excluding) *iff* the transitive closures of its finite restrictions are uniformly decidable (resp. middle-excluding). The main result splits into two parts, one on excluded middle and one on computability: First, consider a middle-excluding relation. It is acyclic and equality on its domain is middle-excluding *iff* its restriction to any finite set has a middle-excluding irreflexive linear extension. Second, consider R a decidable binary relation over A. The following three propositions are equivalent.

Note that computability of linear extensions is non-uniform in the second proposition but uniform in the third one.

- Equality on A is decidable and R is acyclic.
- Equality on A is decidable and every finite restriction of R has a decidable linear extension.
- There exists a computable function that expects finite restrictions of R and returns (decidable) linear extensions of them.

1.8 Contents

Section 2 gives a quick look at the Coq versions of types, binary relations, excluded middle, and computability. Through the example of lists, section 3 explains the principle of definition by induction in Coq, as well as the associated inductive proof principle and definition by recursion. In particular, subsection 3.2 details a simple proof by induction on list. Subsection 4.1 explains the inductive notion of transitive closure and the associated inductive proof principle. Subsections 4.2 and 4.3 discuss irreflexivity, representation of "finite sets" by lists, define finite restrictions of a binary relation, and detail a simple proof by induction on transitive closure. Section 5 defines paths with respect to a binary relation and proves their correspondence with transitive closure. It also defines bounded paths that are proved to preserve decidability and middle-exclusion properties of the original relation. Since bounded paths and paths are by some means equivalent on finite sets, subsection 5.4 states the intermediate result. Subsection 6.1 defines relation totality over finite sets. Subsections 6.2 to 6.5 define an acyclicity-preserving conditional single-arc addition (to a relation), and an acyclicity-preserving multi-stage arc addition over finite sets, which consists in repeating in turn single-arc addition and transitive closure. This procedure helps state linear extension equivalence in 6.6 and topological sorting equivalence in 6.7.

1.9 Convention

Let A be a *Set*. Throughout this paper x, y, z, and t implicitly refer to objects of type A. In the same way R, R', and R'' refer to binary relations over A; l, l', and l'' to lists over A, and n to natural numbers. For the sake of readability, types will sometimes be omitted according to the above convention, even in formal statements where Coq could not infer them. The notation $\neg P$ stands for $P \to \mathit{False}$, $x \neq y$ for $x = y \to \mathit{False}$, and $\nexists x,\, P$ for $(\exists x,\, P) \to \mathit{False}$.

2 Preliminaries

2.1 Types and Relations

Any Coq object has a type, which informs of the usage of the object and its possible interactions with other Coq objects. The Coq syntax $\mathit{Obj} : T$ means that Obj has type T. For example, $f : A \to B$ means that f requires an argument in the domain A and returns an output in the codomain B. If x has type A then f and x can be combined and yield $(f\ x)$, also written $f(x)$ or $f\ x$, of type B. A type is also a Coq object so it has a type too. The only types of types mentioned in this paper are *Prop* and *Set*. The two propositions *True* and *False* are in *Prop* but in a constructive setting there are propositions, *i.e.*, objects in *Prop*, neither

equivalent to *True* nor to *False*. Both the collection of all natural numbers and the collection of the two booleans *true* and *false* have type *Set*. Intuitively, proving propositions in *Prop* amounts to traditional (and intuitionistic) mathematical reasoning as proving objects in *Set* is computationally stronger since effective programs can be extracted from theorems in *Set*. Now consider $g : A \to (B \to C)$ where the parentheses are usually omitted by convention. The function g expects an argument in A and returns a function expecting an argument in B and returning an output in C. Therefore g can be seen as a function requiring a first argument in A, a second one in B, and returning an object in C. Binary relations over A can be represented by functions typed in $A \to A \to Prop$, *i.e.* requiring two arguments in A and returning a proposition (that may be interpreted as "the two arguments are related"). The returned proposition may be *True*, *False*, or something else that may or may not be equivalent to either *True* or *False*. For example if it returns always something absurd, *i.e.*, implying *False*, then it is "the" empty relation over A. The object *Identity_relation*, defined below in the light Coq formalism using this paper's convention, can be interpreted as the identity relation over A. Indeed, it requires two arguments in A and returns a proposition asserting that those arguments are equal.

Definition *Identity_relation* $x\ y : Prop := x{=}y$.

The actual Coq code would need to make it clear that x and y are in A.

Definition *Identity_relation* $(x\ y : A) : Prop := x{=}y$.

2.2 Excluded Middle and Decidability

The following two objects define middle-excluding equality on A and middle-excluding binary relations over A, respectively.

Definition *eq_midex* $:= \forall\ x\ y,\ x{=}y \vee x{\neq}y$.

Definition *rel_midex* $R := \forall\ x\ y,\ R\ x\ y \vee \neg R\ x\ y$.

Note that the proposition *eq_midex* is only a definition but not a theorem in Coq, *i.e.*, there is no proof of which the conclusion is the proposition $\forall\ x\ y,\ x{=}y \vee x{\neq}y$. Same remark for *rel_midex*.

This paper widely uses the syntax $\forall\ v,\ \{B\}{+}\{C\}$ which is a $\forall\ v,\ B \vee C$ with a computational content. It means that for all v either B holds or C holds, and that, in addition, there exists a computable function expecting a v and pointing to one that holds. The next two definitions respectively say that equality on A is decidable and that a given binary relation over A is decidable.

Definition *eq_dec* $:= \forall\ x\ y,\ \{x{=}y\}{+}\{x{\neq}y\}$.

Definition *rel_dec* $R := \forall\ x\ y,\ \{R\ x\ y\}{+}\{\neg R\ x\ y\}$.

The remainder of this subsection 2.2 justifies further the syntax $\forall\ v,\ \{B\}{+}\{C\}$ as correct representation for decidability. Roughly speaking, $\{Obj : T \mid P\}$ means the existence, with computational content, of an object of type T satisfying the predicate P, whereas $\exists\ Obj : T$, P does the same without computational content. As shown by the two lemmas below, *rel_dec* R is equivalent to "computable" existence of a function requiring two arguments in A and returning the boolean *true* if the two arguments are related or *false* if they are not, which amounts to decidability as discussed in subsection 1.5.

Lemma *rel_dec_bool* : $\forall$ *R*,
rel_dec R $\rightarrow$ $\{f : A \rightarrow A \rightarrow bool \mid \forall\ x\ y : A,\ if\ f\ x\ y\ then\ R\ x\ y\ else\ \neg R\ x\ y\}$.

Lemma *bool_rel_dec* : $\forall$ *R*,
$\{f : A \rightarrow A \rightarrow bool \mid \forall\ x\ y : A,\ if\ f\ x\ y\ then\ R\ x\ y\ else\ \neg R\ x\ y\} \rightarrow$ *rel_dec R*.

Therefore the syntax $\{P\}+\{\neg P\}$ is usually a convenient way to represent decidability in Coq. In terms of usage: while proving a proposition, both $x{=}y \vee x{\neq}y$ and $\{x{=}y\}+\{x{\neq}y\}$ allow to case split (first case $x{=}y$ and second case $x{\neq}y$). When building a function, however, only $\{x{=}y\}+\{x{\neq}y\}$ allows to case split, *e.g.*, to perform a traditional "if-then-else". It means that the computational content of $\{x{=}y\}+\{x{\neq}y\}$ is stronger, as suggested below. The same kind of result is available for binary relations.

Lemma *eq_dec_midex* : *eq_dec* $\rightarrow$ *eq_midex*.

Lemma *rel_dec_midex* : *rel_dec* $\rightarrow$ *rel_midex*.

3 On Lists

3.1 Lists in the Coq Standard Library

All the underlying Coq code of this subsection can be found in the Coq Standard Library [1]. Semi-formally, lists are defined by *induction* as follows. Let B be any set. Consider all L complying with the following two conditions:

- *nil* is in L and is called the empty list over B.
- If x is in B and l is in L then (*cons x l*) is in L.

The lists over B are defined as the (unique) least such an L. Formally in Coq, lists over A can be defined as follows:

Inductive *list* : *Set* :=
| *nil* : *list*
| *cons* : $A \rightarrow$ *list* $\rightarrow$ *list*.

The first line defines lists over A by induction and considers the collection of all lists over A as a *Set*. The next line says that *nil* is a list over A and the last line says that applying an element of A and list over A to the constructor *cons* yields a new list over A. The notation $x{::}l$ stands for *cons x l*. Call x the head of $x{::}l$ and l its tail.

Example 1 *If x and y are in A then cons x (cons y nil) is a list over A represented by x::y::nil.*

In Coq, any definition by induction comes with an inductive proof principle, aka induction principle. For lists, it states that if a predicate holds for the empty list and is preserved by list construction, then it holds for all lists. Formally:

$\forall\ (P : list \rightarrow Prop),\ P\ nil \rightarrow (\forall\ a\ l,\ P\ l \rightarrow P\ (a :: l)) \rightarrow \forall\ l,\ P\ l$.

There is a way to define functions over inductively defined objects, just along the inductive definition of those objects. This is called a definition by *recursion*. For example, appending two lists l and l' is defined by recursion on the first list argument l.

Fixpoint *app l l'* {*struct l*} : *list* :=
match l with
| *nil* ⇒ *l'*
| *x* :: *l"* ⇒ *x* :: *app l" l'*
end.

The function *app* requires two lists over A and returns a list over A. The command Fixpoint means that *app* is defined by recursion and {*struct l*} means that the recursion involves l, the first list argument. As lists are built using two constructors, the body of the function case splits on the structure of the list argument. If it is *nil* then the appending function returns the second list argument. If not then the appending function returns a list involving the same appending function with a strictly smaller first list argument, which ensures process termination. The notation *l*++*l'* stands for *app l l'*.

Example 2 *Let x, y, z, and t be in A. Computation steps are shown below.*
(x::y::nil)++(z::t::nil) ⇝ *x::((y::nil)++(z::t::nil))* ⇝
x::y::(nil++(z::t::nil)) ⇝ *x::y::z::t::nil.*

The function *length* of a list is defined by recursion, as well as the predicate *In* saying that a given element occurs in a given list. The predicate *incl* says that all the elements of a first list occur in a second list. It is defined by recursion on the first list, using *In*.

3.2 Decomposition of a List

If equality is middle-excluding on A and if an element occurs in a list built over A, then the list can be decomposed into three parts: a list, one occurrence of the element, and a second list where the element does not occur.

Lemma *In_elim_right* : *eq_midex* → ∀ *x l*,
In x l → ∃ *l'*, ∃ *l"*, *l*=*l'*++(*x*::*l"*) ∧ ¬*In x l"*.

Proof Assume that equality is middle-excluding on A and let x be in A. Next, prove by induction on l the proposition ∀ *l*, *In x l* → ∃ *l'*, ∃ *l"*, *l*=*l'*++(*x*::*l"*) ∧ ¬*In x l"*. The base case, *l*=*nil*, is straightforward since x cannot occur in the empty list. For the inductive case, *l*=*y*::*l1*, the induction hypothesis is *In x l1* → ∃ *l'*, ∃ *l"*, *l1*=*l'*++(*x*::*l"*) ∧ ¬*In x l"*. Assume that x occurs in l and prove ∃ *l'*, ∃ *l"*, *y*::*l1*=*l'*++(*x*::*l"*) ∧ ¬*In x l"* as follows: case split on x occurring in *l1*. If x occurs in *l1* then get *l'* and *l"* from the induction hypothesis, and show that *y*::*l'* and *l"* are witnesses. If x does not occur in *l1* then x equals y, so *nil* and *l1* are witnesses. □

3.3 Repeat-Free Lists

The predicate *repeat_free* says that no element occurs more than once in a given list. It is defined by recursion on its sole argument.

Fixpoint *repeat_free l* : *Prop* :=
match l with
| *nil* ⇒ *True*
| *x*::*l'* ⇒ ¬*In x l'* ∧ *repeat_free l'*
end.

If equality is middle-excluding on A then a *repeat_free* list included in another list is not longer than the other list. This is proved by induction on the *repeat_free* list. For the inductive step, invoke *In_elim_right* to decompose the other list along the head of the *repeat_free* list.

Lemma *repeat_free_incl_length* : *eq_midex* → ∀ *l l'*,
repeat_free l → *incl l l'* → *length l*≤*length l'*.

4 On Relations

4.1 Transitive Closure in the Coq Standard Library

Traditionally, the transitive closure of a binary relation is *the* smallest transitive binary relation including the original relation. The notion of transitive closure can be formally defined by induction, in the Coq Standard Library. The following function *clos_trans* expects a relation over A and yields its transitive closure, which is also a relation over A.

```
Inductive clos_trans R : A → A → Prop :=
  | t_step : ∀ x y, R x y → clos_trans R x y
  | t_trans :
      ∀ x y z, clos_trans R x y → clos_trans R y z → clos_trans R x z.
```

Informally, the *t_step* constructor guarantees that *clos_trans R* contains R and the *t_trans* constructor adds all "arcs" the absence of which would contradict transitivity.

Intuitively, two elements are related by the transitive closure of a binary relation if one can start at the first element and reach the second one in finitely many steps of the original relation. Therefore replacing *clos_trans R x y* → *clos_trans R y z* → *clos_trans R x z* by *R x y* → *clos_trans R y z* → *clos_trans R x z* or *clos_trans R x y* → *R y z* → *clos_trans R x z* in the definition of *clos_trans* would yield two relations coinciding with *clos_trans*. Those three relations are yet different in intension: only *clos_trans* captures the meaning of the terminology "transitive closure".

The Coq Standard Library also defines what a *transitive* relation is. In addition, this paper needs the notion of subrelation.

Definition *sub_rel R R'* : *Prop* := ∀ *x y*, *R x y* → *R' x y*.

The notion of subrelation helps express the induction principle for *clos_trans*. It states that if a relation contains R and satisfies the following "weak transitivity" property then it also contains *clos_trans R*.

∀ *R'*, *sub_rel R R'* →
(∀ *x y z*, *clos_trans R x y* → *R' x y* → *clos_trans R y z* →*R' y z* → *R' x z*) → *sub_rel* (*clos_trans R*) *R'*

The next lemma asserts that a transitive relation contains its own transitive closure (they actually coincide).

Lemma *transitive_sub_rel_clos_trans* : ∀ *R*,
transitive R → *sub_rel* (*clos_trans R*) *R*.

Proof Let R be a transitive relation over A. Prove the subrelation property by the induction principle of *clos_trans*. The base case is trivial and the inductive case is derived from the transitivity of R. □

4.2 Irreflexivity

A relation is irreflexive if no element is related to itself. Therefore irreflexivity of a relation implies irreflexivity of any subrelation.

Definition *irreflexive* R : *Prop* := $\forall\ x, \neg R\ x\ x$.

Lemma *irreflexive_preserved* : $\forall\ R\ R'$,
sub_rel $R\ R' \rightarrow$ *irreflexive* $R' \rightarrow$ *irreflexive* R.

4.3 Restrictions

Throughout this paper, finite "subsets" of A are represented by lists over A. For that specific use of lists, the number and the order of occurrences of elements in a list are irrelevant. Let R be a binary relation over A and l be a list over A. The binary relation *restriction* $R\ l$ relates elements that are both occurring in l and related by R. The predicate *is_restricted* says that "the support of the given binary relation R is included in the list l". And the next lemma shows that transitive closure preserves restriction to a given finite set.

Definition *restriction* $R\ l\ x\ y$: *Prop* := *In* $x\ l \wedge$ *In* $y\ l \wedge R\ x\ y$.

Definition *is_restricted* $R\ l$: *Prop* := $\forall\ x\ y, R\ x\ y \rightarrow$ *In* $x\ l \wedge$ *In* $y\ l$.

Lemma *restricted_clos_trans* : $\forall\ R\ l$,
is_restricted $R\ l \rightarrow$ *is_restricted* (*clos_trans* R) l.

Proof Assume that R is restricted to l. Let x and y in A be such that *clos_trans* $R\ x\ y$, and prove by induction on that last hypothesis that x and y are in l. The base case, where "*clos_trans* $R\ x\ y$ comes from $R\ x\ y$", follows by definition of restriction. For the inductive case, where "*clos_trans* $R\ x\ y$ comes from *clos_trans* $R\ x\ z$ and *clos_trans* $R\ z\ y$ for some z in A", induction hypotheses are *In* $x\ l \wedge$ *In* $z\ l$ and *In* $z\ l \wedge$ *In* $y\ l$, which allows to conclude. □

If the support of a relation involves only two (possibly equal) elements, and if those two elements are related by the transitive closure, then they are also related by the original relation. By the induction principle for *clos_trans* and lemma *restricted_clos_trans*.

Lemma *clos_trans_restricted_pair* : $\forall\ R\ x\ y$,
is_restricted R (x::y::*nil*) $\rightarrow$ *clos_trans* $R\ x\ y \rightarrow R\ x\ y$.

5 On Paths and Transitive Closure

5.1 Paths

The notion of path relates to one interpretation of transitive closure. Informally, a path is a list recording consecutive steps of a given relation. The following predicate says that a given list is a path between two given elements with respect to a given relation.

Fixpoint *is_path* $R\ x\ y\ l$ {*struct* l} : *Prop* :=
match l *with*
| *nil* $\Rightarrow R\ x\ y$
| z::l' $\Rightarrow R\ x\ z \wedge$ *is_path* $R\ z\ y\ l'$
end.

The following two lemmas show the correspondence between paths and transitive closure. The first is proved by the induction principle of *clos_trans* and an appending property on paths proved by induction on lists. For the second, let y be in A and prove $\forall\ l\ x$, *is_path* $R\ x\ y\ l \rightarrow$ *clos_trans* $R\ x\ y$ by induction on l. Now consider the variable appearance order $\forall\ y\ l\ x$ in this lemma. Changing the order would yield a correct lemma as well, but the proof would be less workable. Indeed y can be fixed once for all but x needs to be universally quantified in the induction hypothesis, so x must appear after l on which the induction is performed. Also note that the two lemmas imply $\forall\ x\ y$, *clos_trans* $R\ x\ y \leftrightarrow \exists\ l$, *is_path* $R\ x\ y\ l$.

Lemma *clos_trans_path* : $\forall\ x\ y$, *clos_trans* $R\ x\ y \rightarrow \exists\ l$, *is_path* $R\ x\ y\ l$.

Lemma *path_clos_trans* : $\forall\ y\ l\ x$, *is_path* $R\ x\ y\ l \rightarrow$ *clos_trans* $R\ x\ y$.

Assume that equality is middle-excluding on A and consider a path between two points. Between those two points there is a *repeat_free* path avoiding them and (point-wise) included in the first path. The inclusion is also arc-wise by construction, but it is not needed in this paper.

Lemma *path_repeat_free_length* : *eq_midex* $\rightarrow \forall\ y\ l\ x$,
is_path $R\ x\ y\ l \rightarrow$
$\exists\ l'$, $\neg$*In* $x\ l' \wedge \neg$*In* $y\ l' \wedge$ *repeat_free* $l' \wedge$
length $l' \leq$ *length* $l \wedge$ *incl* $l'\ l \wedge$ *is_path* $R\ x\ y\ l'$.

Proof Assume that equality is middle-excluding on A, let y be in A, and perform an induction on l. For the inductive step, call a the head of l. If a equals y then the empty list is a witness for the existential quantifier. Now assume that a and y are distinct. Use the induction hypothesis with a and get a list l'. Case split on x occurring in l'. If x occurs in l' then invoke lemma *In_elim_right* and decompose l' along x, and get two lists. In order to prove that the second list, where x does not occur, is a witness for the existential quantifier, notice that splitting a path yields two paths (*a priori* between different elements) and that appending reflects the *repeat_free* predicate (if the appending of two lists is *repeat_free* then the original lists also are). Next, assume that x does not occur in l'. If x equals a then l' is a witness for the existential quantifier. If x and a are distinct then $a::l'$ is a witness. □

5.2 Bounded Paths

Given a relation and a natural number, the function *bounded_path* returns a relation saying that there exists a path of length at most the given natural number between two given elements.

Inductive *bounded_path* $R\ n : A \rightarrow A \rightarrow$ *Prop* :=
| *bp_intro* : $\forall\ x\ y\ l$, *length* $l \leq n \rightarrow$ *is_path* $R\ x\ y\ l \rightarrow$ *bounded_path* $R\ n\ x\ y$.

Below, two lemmas relate *bounded_path* and *clos_trans*. The first one is derived from *path_clos_trans*; the second one from *clos_trans_path*, *path_repeat_free_length*, *repeat_free_incl_length*, and a path of a restricted relation being included in the support of the relation. Especially, the second lemma says that in order to know whether two elements are related by the transitive closure of a restricted relation, it suffices to check whether there is, between those two elements, a path of length at most the "cardinal" of the support of the relation.

Lemma *bounded_path_clos_trans* : $\forall\ R\ n$,

sub_rel (*bounded_path R n*) (*clos_trans R*).

Lemma *clos_trans_bounded_path* : *eq_midex* → ∀ *R l*,
is_restricted R l → *sub_rel* (*clos_trans R*) (*bounded_path R* (*length l*)) .

5.3 Restriction, Decidability, and Transitive Closure

The following lemma says that it is decidable whether or not one step of a given decidable relation from a given starting point to some point z in a given finite set and one step of another given decidable relation from the same point z can lead to another given ending point. Moreover such an intermediate point z is computable when it exists, hence the syntax $\{z : A \mid \dots\}$.

Lemma *dec_lem* : ∀ *R' R" x y l*, *rel_dec R'* → *rel_dec R"* →
{*z* : *A* | *In z l* ∧ *R' x z* ∧ *R" z y*}+{∄ *z* : *A*, *In z l* ∧ *R' x z* ∧ *R" z y*}.

The following lemma is the middle-excluding version of the previous lemma.

Lemma *midex_lem* : ∀ *R' R" x y l*, *rel_midex R'* → *rel_midex R"* →
(∃ *z* : *A*, *In z l* ∧ *R' x z* ∧ *R" z y*) ∨ (∄ *z* : *A*, *In z l* ∧ *R' x z* ∧ *R" z y*).

Proof By induction on *l*. For the inductive step, call *a* the head of *l*. Then case split on the induction hypothesis. In the case of existence, any witness for the induction hypothesis is also a witness for the wanted property. In the case of non-existence, case split on *R' x a* and *R" a y*. □

By unfolding the definition *rel_midex*, the next result implies that given a restricted and middle-excluding relation, a given natural number and two given points, either there is a path of length at most that number between those points or there is no such path. Replacing *midex* by *dec* in the lemma yields a correct lemma about decidability.

Lemma *bounded_path_midex* : ∀ *R l n*,
is_restricted R l → *rel_midex R* → *rel_midex* (*bounded_path R n*).

Proof First prove three simple lemmas relating *bounded_path*, *n*, and *S n*. Then let *R* be a middle-excluding relation restricted to *l* and *x* and *y* be in *A*. Perform an induction on *n*. For the inductive step, case split on the induction hypothesis with *x* and *y*. If *bounded_path R n x y* holds then it is straightforward. If its negation holds then case split on *em_lem* with *R*, *bounded_path R n*, *x*, *y*, and *l*. In the existence case, just notice that a path of length less than *n* is of length less than *S n*. In the non-existence case, show the negation of *bounded_path* in the wanted property. □

Let equality and a restricted relation be middle-excluding over *A*, then the transitive closure of the relation is also middle-excluding. The proof invokes *bounded_path_midex*, *bounded_path_clos_trans*, and *clos_trans_bounded_path*. The decidability version of it is also correct.

Lemma *restricted_midex_clos_trans_midex* : *eq_midex* → ∀ *R l*,
rel_midex R → *is_restricted R l* → *rel_midex* (*clos_trans R*).

5.4 Intermediate Results

The following theorems state the equivalence between decidability of a relation and uniform decidability of the transitive closures of its finite restrictions. The first result invokes *clos_trans_restricted_pair* and the second implication uses *restricted_dec_clos_trans_dec*. Note that decidable equality is required only for the second implication. These results remain correct when considering excluded middle instead of decidability.

Theorem *clos_trans_restriction_dec_R_dec* : $\forall$ *R*
($\forall$ *l*, *rel_dec* (*clos_trans* (*restriction R l*))) $\rightarrow$ *rel_dec R*.

Theorem *R_dec_clos_trans_restriction_dec* : *eq_dec* $\rightarrow \forall$ *R*
rel_dec R $\rightarrow \forall$ *l*, *rel_dec* (*clos_trans* (*restriction R l*)).

6 Linear Extension and Topological Sorting

Consider R a binary relation over A and l a list over A. This section presents a way of preserving acyclicity of R while "adding arcs" to the restriction of R to l in order to build a total and transitive relation over l. In particular, if R is acyclic, then its image by the relation completion procedure must be a strict total order. The basic idea is to compute the transitive closure of the restriction of R to l, add an arc *iff* it can be done without creating any cycle, taking the transitive closure, adding an arc if possible, etc. All those steps preserve existing arcs. Since l is finite, there are finitely many eligible arcs, which ensures termination of the process. This is not the fastest topological sort algorithm but its fairly simple expression leads to a simple proof of correctness.

6.1 Total

R is said to be total on l if any two distinct elements in l are related either way. Such a trichotomy property for a relation implies trichotomy for any bigger relation.

Definition *trichotomy R x y* : *Prop* := *R x y* $\vee$ *x*=*y* $\vee$ *R y x*.

Definition *total R l* : *Prop* := $\forall$ *x y*, *In x l* $\rightarrow$ *In y l* $\rightarrow$ *trichotomy R x y*.

Lemma *trichotomy_preserved* : $\forall$ *R R' x y*,
sub_rel R R' $\rightarrow$ *trichotomy R x y* $\rightarrow$ *trichotomy R' x y*.

6.2 Try Add Arc

If x and y are equal or related either way then define the relation *try_add_arc R x y* as R, otherwise define it as the disjoint union of R and the arc (x,y).

Inductive *try_add_arc R x y* : *A* $\rightarrow$ *A* $\rightarrow$ *Prop* :=
| *keep* : $\forall$ *z t*, *R z t* $\rightarrow$ *try_add_arc R x y z t*
| *try_add* : *x*$\neq$*y* $\rightarrow \neg$*R y x* $\rightarrow$ *try_add_arc R x y x y*.

Prove by induction on l and a few case splittings that, under some conditions, a path with respect to an image of *try_add_arc* is also a path with respect to the original relation.

Lemma *path_try_add_arc_path* : $\forall$ *R t x y l z*,
$\neg$(*x*=*z* $\vee$ *In x l*) $\vee \neg$(*y*=*t* $\vee$ *In y l*) $\rightarrow$

is_path R (try_add_arc R x y) z t l → is_path R z t l.

The next three lemmas lead to the conclusion that the function *try_add_arc* does not create cycles. The first one is derived from a few case splittings and the last one highly relies on the second one but also invokes *clos_trans_path*.

Lemma *trans_try_add_arc_sym* : ∀ *R x y z t*,
transitive R → try_add_arc R x y z t → try_add_arc R x y t z → R z z.

Lemma *trans_bounded_path_try_add_arc* : *eq_midex* → ∀ *R x y z n*,
transitive R → bounded_path (try_add_arc R x y) n z z → R z z.

Proof By induction on *n*. The base case requires only *trans_try_add_arc_sym*. For the inductive case, consider a path of length less than or equal to *n+1* and build one of length less than *n+1* as follows. By *path_repeat_free_length* the path may be *repeat_free*, *i.e.*, without circuit. Proceed by case splitting on the construction of the path: when the path is *nil*, it is straightforward. If the length of the path is one then invoke *sub_rel_try_add_arc trans_try_add_arc_sym*; otherwise perform a 4-case splitting (induced by the disjunctive definition of *try_add_arc*) on the first two (*try_add_arc R x y*)-steps of the path. Two cases out of the four need lemmas *transitive_sub_rel_clos_trans*, *path_clos_trans*, and *path_try_add_arc_path*. □

Lemma *try_add_arc_irrefl* : *eq_midex* → ∀ *R x y*,
transitive R → irreflexive R → irreflexive (*clos_trans* (*try_add_arc R x y*)).

6.3 Try Add Arc (One to Many)

The function *try_add_arc_one_to_many* recursively tries to (by preserving acyclicity) add all arcs starting at a given point and ending in a given list.

Fixpoint *try_add_arc_one_to_many R x l* {*struct l*} : *A → A → Prop* :=
match l with
| *nil* ⇒ *R*
| *y*::*l'* ⇒ *clos_trans* (*try_add_arc* (*try_add_arc_one_to_many R x l'*) *x y*)
end.

The following three lemmas prove preservation properties about the function *try_add_arc_one_to_many*: namely, arc preservation, restriction preservation, and middle-exclusion preservation. Decidability preservation is also correct, although not formally stated here.

Lemma *sub_rel_try_add_arc_one_to_many* : ∀ *R x l*,
sub_rel R (*try_add_arc_one_to_many R x l*).

Proof By induction on *l*. For the inductive step, call *a* the head of *l* and *l'* its tail. Use transitivity of *sub_rel* with *try_add_arc_one_to_many x l'* and *try_add_arc* (*try_add_arc_one_to_many x l'*) *x a*. Also invoke *clos_trans* and a similar arc preservation property for *try_add_arc*. □

Lemma *restricted_try_add_arc_one_to_many* : ∀ *R l x l', In x l → incl l' l →*
is_restricted R l → is_restricted (*try_add_arc_one_to_many R x l'*) *l.*

Proof By induction on *l'*, *restricted_clos_trans*, and a similar restriction preservation property for *try_add_arc*. □

Lemma *try_add_arc_one_to_many_midex* :
eq_midex → ∀ *R x l l'*, *In x l* → *incl l' l* → *is_restricted R l* →
rel_midex R → *rel_midex* (*try_add_arc_one_to_many R x l'*).

Proof By induction on *l'*. Also invoke *restricted_try_add_arc_one_to_many*, lemma *restricted_midex_clos_trans_midex* with *l*, and a similar middle-exclusion preservation property for *try_add_arc*. □

Next, a step towards totality.

Lemma *try_add_arc_one_to_many_trichotomy* : *eq_midex* → ∀ *R x y l l'*,
In y l' → *In x l* → *incl l' l* → *is_restricted R l* → *rel_midex R* →
trichotomy (*try_add_arc_one_to_many R x l'*) *x y*.

Proof By induction on *l'*. For the inductive step, invoke *trichotomy_preserved*, case split on *y* being the head of *l'* or *y* occurring in the tail of *l'*. Also refer to a similar trichotomy property for *try_add_arc*. □

6.4 Try Add Arc (Many to Many)

The function *try_add_arc_many_to_many* requires a relation and two lists. Then, using *try_add_arc_one_to_many*, it recursively tries to safely add all arcs starting in first list argument and ending in the second one.

Fixpoint *try_add_arc_many_to_many R l' l* {*struct l'*} : *A* → *A* → *Prop* :=
match l' with
| *nil* ⇒ *R*
| *x*::*l"* ⇒ *try_add_arc_one_to_many* (*try_add_arc_many_to_many R l" l*) *x l*
end.

The following three results proved by induction on the list *l'* state arc, restriction, and decidability preservation properties of *try_add_arc_many_to_many*. For the inductive case of the first lemma, call *l"*the tail of *l'*, apply the transitivity of *sub_rel* with (*try_add_arc_many_to_many R l" l*), and invoke lemma *sub_rel_try_add_arc_one_to_many*. Use *restricted_try_add_arc_one_to_many* for the second lemma. For the third one invoke*try_add_arc_one_to_many_dec* and also *restricted_try_add_arc_many_to_many*. Middle-exclusion preservation is also correct, although not formally stated here.

Lemma *sub_rel_try_add_arc_many_to_many* : ∀ *R l l'*,
sub_rel R (*try_add_arc_many_to_many R l' l*).

Lemma *restricted_try_add_arc_many_to_many* : ∀ *R l l'*, *incl l' l* →
is_restricted R l → *is_restricted* (*try_add_arc_many_to_many R l' l*) *l*.

Lemma *try_add_arc_many_to_many_dec* : → ∀ *R l l'*, *incl l' l* →
is_restricted R l → *rel_dec R* → *rel_dec* (*try_add_arc_many_to_many R l' l*).

The next two results state a trichotomy property and that the function *try_add_arc_many_to_many* does not create any cycle.

Lemma *try_add_arc_many_to_many_trichotomy* : *eq_midex* → ∀ *R l x y l'*,
incl l' l → *In y l* → *In x l'* → *restricted R l* → *rel_midex R* →
trichotomy (*try_add_arc_many_to_many R l' l*) *x y*.

Proof By induction on *l'*. Start the inductive step by case splitting on x being the head of *l'* or occurring in its tail *l"*. Conclude the first case by *try_add_arc_one_to_many_trichotomy*, *try_add_arc_many_to_many_midex*, and *restricted_try_add_arc_many_to_many*. Use *trichotomy_preserved*, the induction hypothesis, and *sub_rel_try_add_arc_one_to_many* for the second case. □

Lemma *try_add_arc_many_to_many_irrefl* : *eq_midex* → ∀ *R l l'*,
incl l' l → *is_restricted R l* → *transitive A R* →
irreflexive R → *irreflexive* (*try_add_arc_many_to_many R l' l*).

Proof By induction on *l'*. For the inductive step, first prove a similar irreflexivity property for *try_add_arc_one_to_many* by induction on lists and *try_add_arc_irrefl*. Then invoke *restricted_try_add_arc_many_to_many*. Both this proof and the one for *try_add_arc_one_to_many* also require transitivity of the transitive closure and an additional case splitting on *l'* being *nil* or not. □

6.5 Linear Extension/Topological Sort Function

Consider the restriction of a given relation to a given list. The following function tries to add all arcs both starting and ending in that list to that restriction while still preserving acyclicity.

Definition *LETS R l* : *A* → *A* → *Prop* :=
try_add_arc_many_to_many (*clos_trans* (*restriction R l*)) *l l*.

The next three lemmas are proved by *sub_rel_try_add_arc_many_to_many*, *transitive_clos_trans*, and *restricted_try_add_arc_many_to_many* respectively.

Lemma *LETS_sub_rel* : ∀ *R l*,
sub_rel (*clos_trans* (*restriction R l*)) (*LETS R l*).

Lemma *LETS_transitive* : ∀ *R l*, *transitive* (*LETS R l*).

Lemma *LETS_restricted* : ∀ *R l*, *is_restricted* (*LETS R l*) *l*.

Under middle-excluding equality, the finite restriction of R to l has no cycle *iff* *LETS R l* is irreflexive. Prove left to right by *try_add_arc_many_to_many_irrefl*, and right to left by *irreflexive_preserved* and *LETS_sub_rel*.

Lemma *LETS_irrefl* : *eq_midex* → ∀ *R l*,
(*irreflexive* (*clos_trans* (*restriction R l*)) ↔ *irreflexive* (*LETS R l*)).

If R and equality on A are middle-excluding then *LETS R l* is total on *l*. This is proved by *R_midex_clos_trans_restriction_midex* (in 5.4) and *try_add_arc_many_to_many_trichotomy*.

Lemma *LETS_total* : *eq_midex* → ∀ *R l*, *rel_midex R* → *total* (*LETS R l*) *l*.

The next two lemmas show that if R and equality on A are middle-excluding (resp. decidable) then so is *LETS R l*: by *try_add_arc_many_to_many_midex* (resp. *try_add_arc_many_to_many_dec*) and *R_midex_clos_trans_restriction_midex* (resp. *R_dec_clos_trans_restriction_dec*).

Lemma *LETS_midex* : *eq_midex* → ∀ *R l*,
rel_midex R → *rel_midex* (*LETS R l*).

Lemma *LETS_dec* : *eq_dec* → ∀ *R*, *rel_dec R* → ∀ *l*, *rel_dec* (*LETS R l*).

6.6 Linear Extension

Traditionally, a linear extension of a partial order is a total order including the partial order. Below, a linear extension (over a list) of a binary relation is a strict total order (over the list) that is bigger than the original relation (restricted to the list).

Definition *linear_extension R l R'* := *is_restricted R' l* ∧
sub_rel (*restriction R l*) *R'* ∧ *transitive A R'* ∧ *irreflexive R'* ∧ *total R' l*.

The next two lemmas say that a relation "locally" contained in some acyclic relation is "globally" acyclic and that if for any list over A there is a middle-excluding total order over that list, then equality is middle-excluding on A.

Lemma *local_global_acyclic* : ∀ *R*,
(∀ *l*, ∃ *R'*, *sub_rel* (*restriction R l*) *R'* ∧ *transitive R'* ∧ *irreflexive R'*) →
irreflexive (*clos_trans R*).

Proof Let R be a relation over A. Assume that any finite restriction of R is included in some strict partial order. Let x be in A such that *clos_trans R x x*. Then derive *False* as follows. Invoke *clos_trans_path* and get a path. It is still a path for the restriction of R to the path itself (the path is a list seen as a subset of A). Use *path_clos_trans*, then the main assumption, *transitive_sub_rel_clos_trans*, and the monotonicity of *clos_trans* with respect to *sub_rel*. □

Lemma *total_order_eq_midex* :
(∀ *l*, ∃ *R*, *transitive R* ∧ *irreflexive R* ∧ *total R l* ∧ *rel_midex R*) → *eq_midex*.

Proof Assume the left conjunct, let x and y be in A, use the assumption with $x::y::nil$, get a relation, and double case split on x and y being related either way. □

Consider a middle-excluding relation on A. It is acyclic and equality is middle-excluding on A *iff* for any list over A there exists, on the given list, a decidable strict total order containing the original relation.

Theorem *linearly_extendable* : ∀ *R*, *rel_midex R* →
(*eq_midex* ∧ *irreflexive* (*clos_trans R*) ↔
∀ *l*, ∃ *R'*, *linear_extension R l R'* ∧ *rel_midex R'*).

Proof Left to right: by the relevant lemmas of subsection 6.5, (*LETS R l*) is a witness for the existential quantifier. Right to left by *local_global_acyclic* and *total_order_eq_midex*. □

6.7 Topological Sorting

In this subsection, excluded-middle results of subsection 6.6 are translated into decidability results and augmented: as there is only one concept of linear extension in subsection 6.6, this section presents three slightly different concepts of topological sort. Instead of the equivalence of theorem *linearly_extendable*, those three definitions yield a quadruple equivalence.

From now on a decidable relation may be represented by a function to booleans instead of a function to *Prop* satisfying the definition *rel_dec*. However, those two representations are "equivalent" thanks to lemmas *rel_dec_bool* and *bool_rel_dec* in subsection 2.2.

In this article, a given relation over A is said to be non-uniformly (topologically) sortable if the restriction of the relation to any list has a decidable linear extension.

Definition *non_uni_topo_sortable R* :=
$\forall\ l, \exists\ R' : A \to A \to bool$, *linear_extension R l* (*fun x y* $\Rightarrow$ *R' x y*=*true*).

In the definition above, *R'* represents a decidable binary relation that intends to be a linear extension of R over the list l. But *R'* has type $A \to A \to bool$ so it cannot be used with the predicate *linear_extension R l* that expects an object of type $A \to A \to Prop$, which is the usual type for representing binary relations in Coq. The function *fun x y* $\Rightarrow$ (*R' x y*)=*true* above is the translation of *R'* in the suitable type/representation. It expects two elements x and y in A and returns the proposition *R' x y*=*true*, in *Prop*.

In this article, a given relation over A is said to be uniformly sortable if there exists a computable function expecting a list over A and producing, over the list argument, a (decidable) linear extension of the original relation.

Definition *uni_topo_sortable R* := {*F* : *list A* $\to$ *A* $\to$ *A* $\to$ *bool* |
$\forall\ l$, *linear_extension R l* (*fun x y* $\Rightarrow$ (*F l x y*)=*true*)}.

The third definition of topological sort uses the concept of *asymmetry*, which is now informally introduced; from an algorithmic viewpoint: given a way of representing binary relations, different objects may represent the same binary relation; from a logical viewpoint: two binary relations different in intension, *i.e.* their definitions intend different things, may still coincide, *i.e.* may be logically equivalent. In an arbitrary topological sort algorithm, the returned linear extension may depend on which object has been chosen to represent the original binary relation. For example, applying the empty relation on a given two-element set to a topological sort algorithm may produce the two possible linear extensions depending on the order in which the two elements constituting the set are given. This remark leads to the following definition.

Definition *asym R G* := $\forall\ x\ y : A$,
$x \neq y \to \neg R\ x\ y \to \neg R\ y\ x \to \neg(G\ (x::y::nil)\ x\ y \wedge G\ (y::x::nil)\ x\ y)$.

Next comes the definition of asymmetry for a topological sort of a binary relation. The syntax *let variable*:= *formula in formula'* avoids writing *formula* several times in *formula'*.

Definition *asym_topo_sortable R* := {*F* : *list A* $\to$ *A* $\to$ *A* $\to$ *bool* |
let G:= (*fun l x y* $\Rightarrow$ *F l x y*=*true*) *in*
asym R G $\wedge\ \forall\ l$, *linear_extension R l* (*G l*)}.

Given a binary relation R over A, the remainder of this subsection proves that the four following assertions are equivalent:

1. Equality on A is decidable, and R is decidable and acyclic.
2. R is middle-excluding and asymmetrically sortable.
3. R is decidable and uniformly sortable.
4. Equality on A is decidable, and R is decidable and non-uniformly sortable.

The following lemma says that if there exists a computable function expecting a list over A and producing a (decidable) strict total order over A, then equality on A is decidable. The proof is similar to the one for *total_order_eq_midex*.

Lemma *total_order_eq_dec* :

$\{F : list\ A \to A \to A \to bool \mid \forall\ l,\ let\ G := fun\ x\ y \Rightarrow F\ l\ x\ y{=}true\ in$
$transitive\ A\ G \wedge irreflexive\ G \wedge total\ G\ l\} \to eq\_dec\ A.$

Next lemma shows that *LETS* yields asymmetric topological sort.

Lemma $LETS\_asym : \forall\ R,\ asym\ R\ (LETS\ R).$

Proof Assume all possible premises, especially let x and y be in A. As a preliminary: the hypotheses involve one relation image of *restriction* and four relations images of *try_add_arc*. Prove that all of them are restricted to $x{::}y{::}nil$. Then perform a few cases splittings and apply *clos_trans_restricted_pair* seven times. □

The quadruple equivalence claimed above is derived from *rel_dec_midex* and the six theorems below. The proofs are rather similar to the middle-excluding case in subsection 6.6. The first theorem proves 1 → 2 by the relevant lemmas of subsection 6.5 and *LETS* producing a witness for the computational existence. The second (straightforward) and the third show 2 → 3. The fourth (straightforward) and the fifth, proved by *total_order_eq_dec*, yield 3 → 4. The last shows 4 → 1 by invoking *local_global_acyclic*.

Theorem $possible\_asym\_topo\_sorting : \forall\ R,$
$eq\_dec\ A \to rel\_dec\ R \to irreflexive\ (clos\_trans\ A\ R) \to asym\_topo\_sortable\ R.$

Theorem $asym\_topo\_sortable\_uni\_topo\_sortable : \forall\ R,$
$asym\_topo\_sortable\ R \to uni\_topo\_sortable\ R.$

Theorem $asym\_topo\_sortable\_rel\_dec : \forall\ R,$
$rel\_midex\ R \to asym\_topo\_sortable\ R \to rel\_dec\ R.$

Proof First notice that R is acyclic by *local_global_acyclic* and that equality on A is decidable by *total_order_eq_dec*. Then let x and y by in A. By decidable equality, case split on x and y being equal. If they are equal then they are not related by acyclicity. Now consider that they are distinct. Thanks to the assumption, get TS an asymmetric topological sort of R. Case split on x and y being related by TS $(x{::}y{::}nil)$. If they are not then they cannot be related by R by subrelation property. If they are related then case split again on x and y being related by TS $(y{::}x{::}nil)$. If they are not then they cannot be related by R by subrelation property. If they are then they also are by R by the asymmetry property. □

Theorem $uni\_topo\_sortable\_non\_uni\_topo\_sortable : \forall\ R,$
$uni\_topo\_sortable\ R \to non\_uni\_topo\_sortable\ R.$

Theorem $rel\_dec\_uni\_topo\_sortable\_eq\_dec : \forall\ R,$
$rel\_dec\ R \to uni\_topo\_sortable\ R \to eq\_dec\ A.$

Theorem $rel\_dec\_non\_uni\_topo\_sortable\_acyclic : \forall\ R,$
$rel\_dec\ R \to non\_uni\_topo\_sortable\ R \to irreflexive\ (clos\_trans\ A\ R).$

7 Conclusion

This paper has given a detailed account on a few facts related to linear extensions of acyclic binary relations. The discussion is based on a formal proof developed with the proof assistant

Coq. Since arguments are constructive, they are also correct with respect to traditional mathematical reasoning. The paper aimes to be understandable to mathematicians new to Coq or more generally by readers unfamiliar with constructive issues. The three main results are stated again below. First, a binary relation over a set with decidable/middle-excluding equality is decidable/middle-excluding *iff* transitive closures of its finite restrictions are also decidable/middle-excluding. This theorem is involved in the proof of the second and third main results. Second, consider a middle-excluding relation over an arbitrary domain. It is acyclic and equality on its domain is middle-excluding *iff* any of its finite restriction has a middle-excluding linear extension. Third, consider R a decidable binary relation over A. The following three propositions are equivalent:

- Equality on A is decidable and R is acyclic.
- Equality on A is decidable and R is *non-uniformly* sortable.
- R is *uniformly* sortable.

The proofs of the last two main results rely on the constructive function *LETS* that is actually (similar to) a basic topological sort algorithm. An effective program could therefore be extracted from the Coq development (related to computability). The original proof would in turn serve as a formal proof of correctness for the program.

8 Acknowledgement

I thank Pierre Lescanne for his careful reading and helpful comments, as well as Philippe Audebaud, Guillaume Melquiond, and Victor Poupet for discussions, and Jingdi Zeng for proofreading my English.

References

[1] The Coq proof assistant, version 8.1, http://coq.inria.fr/.

[2] Fabio Acerbi. Plato: Parmenides 149a7-c3. a proof by complete induction? *Archive for History of Exact Sciences*, 55(1):57–76, 2000.

[3] Anonymous. Program evaluation research task. Summary report Phase 1 and 2, U.S. Government Printing Office, Washington, D.C., 1958.

[4] Aristotle. Organon, Prior Analytics, 350 BC.

[5] Yves Berthot and Pierre Castéran. *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[6] F. Blanqui, S. Coupet-Grimal, W. Delobel, S. Hinderer, and A. Koprowski. CoLoR, a Coq Library on rewriting and termination. In *Workshop on Termination*, 2006. http://color.loria.fr/.

[7] Paul Gochet and Pascal Gribomont. *Logique, méthode pour l'informatique fondamentale*. Hermes, 1991.

[8] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, 1962.

[9] Donald E. Knuth. *The Art of Computer Programming*, volume 1, second edition. Addison Wesley, 1973.

[10] Daniel J. Lasser. Topological ordering of a list of randomly-numbered elements of a network. *Commun. ACM*, 4(4):167–168, 1961.

[11] Francesco Maurolico. Arithmeticorum libri duo, 1575.

[12] M.P.Jarnagin. Automatic machine methods of testing pert networks for consistency. Technical Memorandum K-24/60, U. S. Naval Weapons Laboratory, Dahlgren, Va, 1960.

[13] R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer, editors. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1994.

[14] Giuseppe Peano. *Arithmetices principia, nova methodo exposita*. 1889.

[15] Vaughan Pratt. Origins of the calculus of binary relations. In *Logic in Computer Science*, 1992.

[16] J.P. Seldin and J.R. Hindley, editors. *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, London, 1980.

[17] Edward Szpilrajn. Sur l'extension de l'ordre partiel. *Fund. Math*, 1930.

[18] A.S. Troelstra and D. Van Dalen. *Constructivism in mathematics. An introduction.*, volume 1, chapter 6, section 1.2. North-Holland, 1988.

[19] John von Neumann. Zur Einfürung der transfiniten Zahlen. *Acta Litterarum ac Scientiarum Regiae Universitatis Hungaricae Franscisco-Josephinae Section Scientiarum Mathematicarum*, 1:199–208, 1923.

[20] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge Mathematical Library, 1910.