Variations on Noetherianness

Denis Firsov Tarmo Uustalu Niccolò Veltri
Institute of Cybernetics at Tallinn University of Technology, Estonia
{denis,tarmo,niccolo}@cs.ioc.ee

In constructive mathematics, several nonequivalent notions of finiteness exist. In this paper, we continue the study of Noetherian sets in the dependently typed setting of the Agda programming language. We want to say that a set is Noetherian, if, when we are shown elements from it one after another, we will sooner or later have seen some element twice. This idea can be made precise in a number of ways. We explore the properties and connections of some of the possible encodings. In particular, we show that certain implementations imply decidable equality while others do not, and we construct counterexamples in the latter case. Additionally, we explore the relation between Noetherianness and other notions of finiteness.

1 Introduction

To work with finite sets in the constructive setting of proof assistants like Agda [1], which is the language we use in this paper, we need to be able to say what a finite set is. The straightforward way of saying that a set is finite is to ask for an enumeration of its elements together with the proof that the enumeration is complete [7]. This idea can be formalized as follows:

```
Listable X = Σ[ xs ∈ List X ] ((x : X) → x ∈ xs)
```

(In Agda, `Σ[ a ∈ A ] B a` is the type of dependent pairs of an element `a` of type `A` and an element of type `B a`. Note the unfortunate and confusing use of ∈ instead of : for typing the bound variable in this notation.)

An alternative notion of finiteness found in the literature is Noetherianness. Intuitively, a set `X` is Noetherian if, whenever we are shown enough elements of `X`, eventually we will have seen some element twice. Following Coquand and Spiwack [4], this idea can be formalized as follows:

```
data NoethAcc' (X : Set) (acc : List X) : Set where
  stop : Dup acc → NoethAcc' X acc
  ask  : ((x : X) → NoethAcc' X (x :: acc)) → NoethAcc' X acc
```

The auxiliary definition `NoethAcc'` is parametrized by a set and an accumulator list over this set. It has two constructors. The constructor `ask` says that, by constructing a proof of `NoethAcc' X (x :: acc)` for all `x : X`, one has constructed a proof of `NoethAcc' X acc`. The constructor `stop` says that, if `acc` already contains a duplicate, then one gets a proof of `NoethAcc' X acc`. Therefore, to construct a proof of `NoethAcc' X acc`, we must show that, if we ask for elements long enough, then, independently of which elements we are presented, we can eventually stop. Finally, we say that the set is Noetherian, if we can prove `NoethAcc'` starting with the empty accumulator:

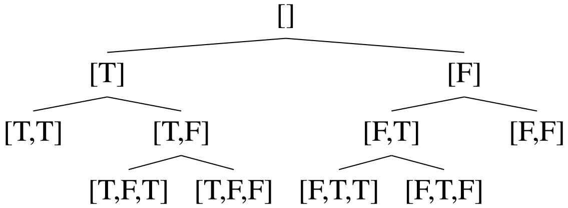
```
NoethAcc X = NoethAcc' X []
```

Let us prove that the set of Booleans is Noetherian. The proof is done by case analysis and pattern matching under lambdas. Every time we ask for an element, the proof tree branches according to whether the element provided is `true` or `false`. Clearly, after asking for elements at least three times, all branches can stop.

Atkey & Krishnaswami (Eds.): MSFP 2016
EPTCS 207, 2016, pp. 76–88, doi:10.4204/EPTCS.207.4

```
BoolNoetherian : NoethAcc Bool
BoolNoetherian =
    ask (λ { true   → ask (λ { true  → stop dup0
                             ; false → ask (λ { true  → stop dup1
                                              ; false → stop dup2 }) })
           ; false → ask (λ  { true → ask (λ { true  → stop dup2
                                             ; false → stop dup1 })
                             ; false → stop dup0 })  })
  where
    -- the implementation of dup* is left out for brevity
    dup0 : {x : Bool} → Dup (x :: x :: [])
    dup1 : {x y : Bool} → Dup (x :: y :: x :: [])
    dup2 : {x y : Bool} → Dup (x :: x :: y :: [])
```

The proof corresponds to the tree of the following shape:



`NoethAcc Bool`

Note that the definition of `NoethAcc` does not require us to terminate immediately after discovering the first duplicate in the accumulator. Therefore, all branches could be continued for some finite number of iterations.

We can already observe that Noetherianness has a number of interesting properties in comparison with listability. A proof that a certain set is Noetherian does not provide an access to elements of the set. This also implies that there is no information about the size of the set, or even about inhabitedness of the set. Moreover, the proof objects of Noetherianness are lightweight and do not induce excessive computations. For example, consider the finite set `Factors n`, which contains all the factors of a natural number `n`. It is straightforward to prove that this set is both listable and Noetherian. However, the proof of listability can cause the search of factors of `n`, if pattern matching is performed on the witness list. This causes problems for big values of `n`.

In this paper, we continue the study of Noetherian sets and make the following main contributions:

- In Section 2, we provide a number of implementations of the idea of Noetherianness and compare them to each other.
- In Section 3, we show that the original definition of Noetherian sets by Coquand and Spiwack [4] (`NoethAcc`) implies decidable equality on the set.
- In Section 3.1, we construct a class of finite sets with generally undecidable equality and prove that they are Noetherian for a particular variation of Noetherianness (`NoethAccS`). This implies that some encodings are logically weaker than others.
- In Section 4, we prove that the variations of Noetherianness introduced are nonequivalent by providing counterexamples. In addition, we establish connections with notions of listability, streamlessness, and almost-fullness. The relation between all the notions of finiteness presented in the paper is summarized in Figure 1.

Section 5 is devoted to related work, conclusions, and further work. In Appendix A, we give a detailed explanation of the basic Agda definitions we use (e.g., membership in a list, duplicates, decidable equality, propositional types, etc).

We used Agda 2.4.2.2 and Agda Standard Library 0.9 for this development. The full Agda code of this paper can be found at `http://cs.ioc.ee/~denis/noeth/`.

2 Different Encodings of Noetherianness

In the introduction, we have presented an encoding of Noetherianness, called `NoethAcc`. In order to construct a proof of `NoethAcc X`, we repeatedly ask for elements of `X` until we end up with an element that we have seen twice. A couple of notes on this encoding:

1. When we ask for an element of `X`, we are given an *arbitrary* one. In particular, we may receive an element that we have already seen and conclude using the constructor `stop`. This may happen before having seen all the elements of `X`.
2. A proof of `NoethAcc X` does not necessarily detect the first element that appears twice. We can keep asking for elements of `X` after we have already seen an element twice.

The above observations expose the fact that there are some degrees of freedom in the encoding of Noetherianness. This section is devoted to the description of implementations of Noetherianness alternative to `NoethAcc`.

The first variation we present is called `NoethAccS`. The extra “S” stands for “strict”. In this implementation, we do not ask for arbitrary elements, but only for *fresh* ones. The idea can be formalized as follows:

```
data NoethAccS' (X : Set) (acc : List X) : Set where
  ask : ((x : X) → ¬ x ∈ acc → NoethAccS' X (x :: acc)) → NoethAccS' X acc
```

The auxiliary definition `NoethAccS'` has only one constructor `ask`. It says that, by constructing a proof of `NoethAccS' X (x :: acc)` for all `x : X` that are not in the accumulator `acc`, one has constructed a proof of `NoethAccS' X acc`. Then we say that the set is Noetherian, if we can prove `NoethAccS'` starting with empty accumulator:

```
NoethAccS X = NoethAccS' X []
```

The base case is reached when, for all `x : X`, we have that `¬ x ∈ acc` is false, i.e., can produce a proof of `¬ ¬ x ∈ acc`. Therefore, in order to construct a proof of `NoethAccS`, we repeatedly ask for fresh elements of `X` until no element can fail to be in the accumulator.

The predicate `NoethAcc` is stronger than `NoethAccS`, since every set that satisfies `NoethAcc` also satisfies `NoethAccS`. More generally, given a duplicate-free accumulator `acc : List X`, we have that `NoethAcc' X acc` implies `NoethAccS' X acc`. This fact is proved as follows:

```
NoethAcc'→NoethAccS' : {X : Set} → (acc : List X) → ¬ Dup acc
  → NoethAcc' X acc → NoethAccS' X acc
NoethAcc'→NoethAccS' acc ¬d (stop d) = ⊥-elim (¬d d)
NoethAcc'→NoethAccS' acc ¬d (ask n) =
  ask (λ x ¬m → NoethAcc'→NoethAccS' (x :: acc) (¬DupCons ¬d ¬m) (n x))
```

where `⊥-elim` is the elimination principle of ⊥ and ¬DupCons constructs a proof of `¬ Dup (x :: acc)` from a proof of `¬ Dup acc` and a proof of `¬ x ∈ acc`. The proof proceeds by induction on the proof of `NoethAcc' X acc`: if `acc` contains a duplicate, then we can derive ⊥, since by hypothesis `acc` is duplicate-free; otherwise, we have a proof `n` of `(x : X) → NoethAcc' X (x :: acc)`. We ask for a fresh element `x`. This element can be added to the accumulator, which remains duplicate-free, and we can invoke the induction hypothesis on `n x : NoethAcc' X (x :: acc)`.

The main statement follows by instantiating `acc` with the empty list and noting that the empty list is duplicate-free.

```
NoethAcc→NoethAccS : {X : Set} → NoethAcc X → NoethAccS X
NoethAcc→NoethAccS n = NoethAcc'→NoethAccS' [] (λ ()) n
```

The converse implication does not generally hold, as we will demonstrate in Section 3.1.

Alternatively, we can implement the idea behind `NoethAccS` without an explicit accumulator. In this variation, which we call `NoethSet`, we also ask for fresh elements of a given type `X`, but instead of storing the already seen elements in a list, we directly remove them from the set `X`. The predicate `NoethSet` can be formalized as follows:

```
data NoethSet (X : Set) : Set where
  ask : ((x : X) → NoethSet (X \ x)) → NoethSet X
```

where the type `X \ x` of elements of `X` different from `x` can be defined as follows:

```
_\_ : (X : Set) → X → Set
X \ x = Σ[ x' ∈ X ] ¬ x' ≡ x
```

The type `NoethSet` has only one constructor `ask`. It says that, to construct a proof of `NoethSet X`, one has to construct a proof of `NoethSet (X \ x)` for all `x : X`. This encoding of Noetherianness was mentioned by Bezem et al.[3]. Here the base case is reached when the type `X` becomes empty, i.e., from an inhabitant `x : X`, we can derive ⊥. Therefore, in order to construct a proof of `NoethSet`, we repeatedly ask for fresh elements of `X` and remove from `X` the elements presented to us, until the set `X` becomes empty.

The encodings `NoethAccS` and `NoethSet` are logically equivalent (`NoethAccS→NoethSet` requires the assumption of the principle of function extensionality).

```
NoethAccS→NoethSet : {X : Set} → NoethAccS X → NoethSet X

NoethSet→NoethAccS : {X : Set} → NoethSet X → NoethAccS X
```

So `NoethSet` does not utilize an accumulator, but it still remembers what elements of `X` have already been seen. These elements correspond exactly to the ones that have been removed from the set `X`.

2.1 The Noetherianness Game

In this subsection, we give a game-theoretic description of Noetherianness. This will help us develop other different variations on the theme. The Noetherianness game, based on the encoding `NoethAccS`, works as follows. Let `X` be a set. Two players participate in the game, the prover and the opponent. The prover cannot see what is inside `X` and repeatedly asks the opponent for fresh elements of `X`. The opponent knows exactly which elements are in `X` and answers the prover's queries by supplying a fresh element. The game terminates, when the opponent cannot provide any fresh element to the prover. The prover wins, if the game terminates in a finite number of steps, no matter what the strategy of the opponent is. The opponent wins, if she can come up with a strategy that makes the game non-terminating.

We present another variant on Noetherianness, also explainable along the lines of the game-theoretical presentation given above. We call it `NoethGame`. At each turn the prover can not only ask for a fresh element of X but also provide an element of X. Whenever the prover asks for a fresh element, the opponent provides one. The game terminates only when the opponent cannot satisfy the prover's request. The winning conditions are the same as in the Noetherianness game. This idea can be formalized as follows:

```
data NoethGame' (X : Set) (acc : List X) : Set where
  tell : (x : X) → NoethGame' X (x :: acc) → NoethGame' X acc
  ask  : ((x : X) → ¬ x ∈ acc → NoethGame' X (x :: acc)) → NoethGame' X acc
```

The constructor `tell` says that having an inhabitant `x : X` and a proof of `NoethGame' X (x :: acc)` makes a proof of `NoethGame' X acc`. Finally, we define `NoethGame X` as `NoethGame' X []`.

```
NoethGame X = NoethGame' X []
```

The predicate `NoethGame` is different in flavor from the Noetherianness predicates introduced before. When constructing a proof of `NoethGame X`, we can not only ask for elements of `X`, but also choose to provide elements of `X`, if we happen to know some. The predicate `NoethGame` is particularly useful, when we have some kind of partial knowledge of the set `X`. Clearly, `NoethAccS X` implies `NoethGame X` by construction.

```
NoethAccS→NoethGame : {X : Set} → NoethAccS X → NoethGame X
```

We present another variant of Noetherianness, also based on the game-theoretic intuition. We modify the rules of `NoethGame`. At each step, the prover can either ask for *any* element of `X`, provide an element of `X`, or win the game by giving a proof that the accumulator is exhaustive. Putting it formally:

```
data NoethExpose' (X : Set) (acc : List X) : Set where
  stop : ((x : X) → x ∈ acc) → NoethExpose' X acc
  tell : (x : X) → NoethExpose' X (x :: acc) → NoethExpose' X acc
  ask  : ((x : X) → NoethExpose' X (x :: acc)) → NoethExpose' X acc

NoethExpose X = NoethExpose' X []
```

A set satisfying `NoethExpose` has an interesting property. Once we know an inhabitant of it, we know that the set is listable.

```
NoethExpose→Listable : {X : Set} → X → NoethExpose X → Listable X
```

Given an element `x : X`. A proof of `NoethExpose X` is a tree that we can walk up to a leaf by choosing the `x`-th branch at each `ask` node. The `tell` nodes construct a list and the leaf node of the path contains a proof that the accumulator has all the elements of `X`.

It is easy to see that every listable set satisfies `NoethExpose`. Moreover, `NoethExpose X` implies `NoethAcc X`. In fact, just ask for an element of `X`. At this point, the set `X` becomes listable. Hence we are done, since listable sets satisfy `NoethAcc`. The converse implication does not generally hold, as we will demonstrate in Section 4.1.

3 Decidable Equality

It has been shown [5] that every listable set has decidable equality. In this section, we prove that the same holds for Noetherian sets in the sense of `NoethAcc`.

```
NoethAcc→DecEq : {X : Set} → NoethAcc X → DecEq X
```

We give an informal account of the proof of `NoethAcc→DecEq`. To decide whether two elements `x` and `y` are equal, we proceed as follows.

- First, we repeatedly feed `x` into the Noetherianness proof, until we get some list `xs` with the proof that it contains duplicates, say `d : Dup xs`. By construction, all elements of `xs` are equal to `x`. The proof `d` points to two different positions `p1` and `p2` in `xs`.
- Next, we repeat the procedure described above, this time feeding `y` instead of `x` at the `p1`-th iteration. The procedure returns a proof `d' : Dup xs'`. The list `xs'` contains `x` at all positions except for the `p1`-th, where `y` has been inserted. The proof `d'` points to two different positions `p1'` and `p2'` in `xs'`.
- If `p1` is equal to `p1'`, then clearly `x` is equal to `y`. In the other case, if `p1` is different from `p1'`, then also `x` differs from `y`.

Notice that the same procedure cannot be replayed for `NoethAccS`, since we cannot feed the same element twice into a proof of `NoethAccS X`.

As a corollary we obtain that, if a set satisfies `NoethExpose`, then it has decidable equality, since `NoethExpose` is stronger than `NoethAcc`.

3.1 A Counterexample to Decidable Equality for `NoethAccS`

In this subsection, we assume the principle of function extensionality.

```
funext : {X Y : Set} {f g : X → Y} → (∀ x → f x ≡ g x) → f ≡ g
```

The principle says that two functions are propositionally equal, if they deliver the same result for all arguments. Assuming `funext`, we show that it is not the case that every set `X` such that `NoethAccS X` has decidable equality. Note that this proves that not every set `X` satisfying `NoethAccS X` satisfies `NoethAcc X`. Let us define a family `NotNotIn` of sets parametrized by a type `X` and a list over `X`.

```
NotNotIn : {X : Set} → List X → Set
NotNotIn {X} xs = Σ[ x ∈ X ] ¬ ¬ x ∈ xs
```

Since propositional equality is not decidable for a general type `X` and the functions of type `¬ ¬ x ∈ xs` are all equal thanks to function extensionality, we have that equality is not generally decidable for `NotNotIn xs`. Moreover, from the general decidable equality on the type `NotNotIn xs`, we can derive decidable equality for every type.

```
EqNotNotIn→Eq : ({X : Set} (xs : List X) → DecEq (NotNotIn xs))
  → {X : Set} → DecEq X
```

The last step is to show that `NotNotIn xs` satisfies `NoethAccS`.

```
NoethAccSNotNotIn : {X : Set} → (xs : List X) → NoethAccS (NotNotIn xs)
```

The proof proceeds as follows. Apply `length xs + 1` times the constructor `ask`. By doing so, we arrive at a list `acc` containing `length xs + 1` different elements. However, the type `NotNotIn xs` has at most as many elements as there are positions in the list `xs`. From the two previous observations, we get a contradiction, which corresponds to the base case of `NoethAccS`.

If we could derive `NoethAcc` from `NoethAccS`, then `NotNotIn xs` would have decidable equality by `NoethAcc→DecEq`. However, then `EqNotNotIn→Eq` would allow us to derive the decidable equality for every type, which is not plausible in a constructive setting.

4 Connections between Noetherianness and Other Notions of Finiteness

In the previous section, we have shown that the notions `NoethAcc` and `NoethAccS` are different by constructing a "separating" class of sets, i.e., a class whose every member satisfies `NoethAccS`, but not `NoethAcc` in general. Given two notions of finiteness `F` and `F'`, we say that `F` is separated from `F'` by a class of sets, if `F' X` holds for all of its members, while `F X` holding for all members implies some nonconstructive principle. In this section, we show which other variations of Noetherianness are separated. We also discuss the connection of Noetherian sets with streamless sets and almost-full relations.

4.1 Separating Listability from `NoethExpose`

There exists a class of sets whose every member satisfies `NoethExpose` but is not listable. Consider any set `X` such that every two elements of `X` are equal, i.e., a proposition. Then `NoethExpose X` holds, just ask for one element of `X` and the element presented has already made the accumulator complete.

```
NoethExposeProp : (X : Set) → isProp X → NoethExpose X
```

On the other hand, if we could construct a proof that any proposition `X` is listable, then, by checking whether the given list is empty or not, we could decide the inhabitedness of `X` (i.e., we could prove the law of excluded middle for propositions).

```
ListableProp→LEM : ((X : Set) → isProp X → Listable X)
  → (X : Set) → isProp X → X + ¬ X
```

4.2 Separation from Bounded Sets

A set `X` is called bounded, if there exists a natural number (bound) `n` such that every list over `X` with more than `n` elements contains duplicates.

```
Bounded X = Σ[ n ∈ ℕ ] (xs : List X) → n ≤ length xs → Dup xs
```

Coquand and Spiwack [4] showed that `Listable X` implies `Bounded X`, and also `Bounded X` implies `NoethAcc X`. Moreover, they proved that `Listable` is separated from `Bounded` and most notably `Bounded` is separated from `NoethAcc`. In fact, they proved that, if every Noetherian set were bounded, then one could derive the limited principle of omniscience (LPO). The same class of sets separating `Bounded` from `NoethAcc` also separates `Bounded` from `NoethExpose`.

We present a class of sets that separates `NoethExpose` from `Bounded`. Consider a proposition `X`. The set `⊤ + X` (where `⊤` is the unit type) is bounded, since it contains at most two elements.

```
MaybePropBounded : (X : Set) → isProp X → Bounded (⊤ + X)
```

On the other hand, if we could construct a proof that `⊤ + X` satisfies `NoethExpose` for every proposition `X`, then we could derive the law of excluded middle for propositions.

```
MaybePropNoethExpose→LEM : (∀ X → isProp X → NoethExpose (⊤ + X))
  → ∀ X → isProp X → X + ¬ X
```

In fact, from the theorem `NoethExpose→Listable`, we have that `NoethExpose (⊤ + X)` implies that the set `⊤ + X` is listable, since `⊤ + X` is inhabited for all `X`. In turn, this implies that `X` is listable. But we already showed that, if every proposition is listable, then we can derive the law of excluded middle for propositions (`ListableProp→LEM`).

Therefore `NoethExpose` and `Bounded` are separated from each other. As a consequence, we have that `NoethExpose` is separated from `NoethAcc`.

4.3 Streamless Sets

A notion of finiteness similar to Noetherianness is the that of streamless set. A set `X` is streamless, if every stream (infinite list) over `X` contains duplicates.

```
Streamless X = (xs : Stream X) → DupS xs
```

The formal definition of `Stream` and `DupS` can be found in Appendix A. Coquand and Spiwack [4] showed that every Noetherian set (in the sense of `NoethAcc`) is streamless.

```
NoethAcc→Streamless : {X : Set} → NoethAcc X → Streamless X
```

Bezem et al. [2] conjectured that it is unprovable that every streamless set is Noetherian. Parmann [9] also proved that every streamless set has decidable equality under the hypothesis of function extensionality (or stream extensionality, depending on the chosen representation of streams).

The encoding of streamless sets admits variations similar to those for Noetherianness introduced earlier. For example, we can define a set `X` to be streamless, if all duplicate-free colists (possibly infinite lists) have finite length.

```
StreamlessS X = (xs : Colist X) → ¬ DupC xs → xs ⇓
```

The formal definition of `Colist`, `DupC` and `_⇓` can be found in Appendix A. Note that this strict variation is similar to `NoethAccS`. Moreover, `NoethAccS` is stronger than `StreamlessS`.

```
NoethAccS→StreamlessS : {X : Set} → NoethAccS X → StreamlessS X
```

As a corollary of `NoethAccS→StreamlessS`, we have that in general `StreamlessS X` does not imply decidability of equality on `X`. Additionally, this shows that `Streamless` and `StreamlessS` are separated similarly to `NoethAcc` and `NoethAccS`.

4.4 Noetherian Sets and Almost-Full Relations

Almost-full relations were introduced by Veldman and Bezem [10] for developing an intuitionistic version of Ramsey theory. Vytiniotis et al. [11] analyzed almost-full relations in connection to program termination and defined this concept as follows:

```
data AF (X : Set) (R : X → X → Set) : Set where
  afzt  : ((x y : X) → R x y) → AF X R
  afsup : ((x : X) → AF X (λ y z → R y z + R x y)) → AF X R
```

A proof terminates, if the relation `R` is total. Otherwise, we ask an element `x` from the opponent and we construct a bigger relation `R'` such that, for all `y z : X`, `R' y z` if and only if `R y z` or `R x y`. Then, by providing a proof of `AF X R'`, we conclude the proof of `AF X R`.

Vytiniotis et al. [11] remarked that the type `AF X _≡_` states that `X` has finitely many inhabitants. We denote `AF X _≡_` by `AFEq X`, and show that, for our hierarchy of encodings, `AFEq` is equivalent to `NoethAcc`.

```
AFEq X = AF X _≡_

AFEq→NoethAcc : {X : Set} → AFEq X → NoethAcc X

NoethAcc→AFEq : {X : Set} → NoethAcc X → AFEq X
```

Notice that the two above results cannot be proved by induction directly, since the second constructor of `AF` proceeds by growing the relation. Therefore, we have to introduce a notion of Noetherianness for binary relations.

```
data NoethAccR' (X : Set)(R : X → X → Set) (acc : List X) : Set where
  stop : DupR R acc → NoethAccR' X R acc
  ask  : ((x : X) → NoethAccR' X R (x :: acc)) → NoethAccR' X R acc

NoethAccR X R = NoethAccR' X R []
```

The notion `NoethAccR'` is different from `NoethAcc'` in the first constructor, where instead of looking for duplicates in the accumulator we search for related elements. This generalized notion `NoethAccR` of Noetherianness for relations is equivalent to `AF`.

```
AF→NoethAccR : {X : Set}{R : X → X → Set} → AF X R → NoethAccR X R

NoethAccR→AF : {X : Set}{R : X → X → Set} → NoethAccR X R → AF X R
```

This equivalence can serve as an explanation of the rather unintuitive notion of almost-fullness.

5 Related Work and Conclusions

Finiteness in constructive mathematics has been studied by various authors recently. Coquand and Spiwack [4] introduced four constructively nonequivalent notions of finite sets in set theory à la Bishop: enumerated sets (that we call listable sets), bounded size sets, Noetherian sets and streamless sets. They showed how these different notions are connected and proved several closure properties. Parmann [9] studied streamless sets in the setting of Martin-Löf type theory. He showed that streamless sets are closed under Cartesian product, if at least one of the sets has decidable equality. Firsov and Uustalu [5] developed a practical toolbox for programming with listable subsets of base sets with decidable equality in Agda. Bezem et al. [3] investigated a number of notions of finiteness of decidable subsets of natural numbers.

In this paper, we introduced several variations on the notion of Noetherian set. Our current knowledge about the relations between different encodings is summed up in Figure 1.

Different encodings of Noetherianness all share the distinctive property of hiding the elements of the set. Nonetheless some implementations "reveal" more information about the set than others. We showed that `NoethExpose`, and most importantly `NoethAcc`, allow one to construct a decider of equality for the set, while such a decider cannot generally be built for sets satisfying `NoethAccS` or `NoethGame`.

It remains open whether `NoethAccS` and `NoethGame` are equivalent notions or not. A class of sets separating `NoethAccS` from `NoethGame` must have the following properties: its members cannot have a computable bound on their size; they cannot have decidable equality.

Coquand and Spiwack [4] analyzed some closure properties of `NoethAcc`, such as closure under subsets, binary products and coproducts. In this paper, we did not extend the study of such properties for the variations on Noetherianness discussed. This is a possible direction for future work. Nevertheless, it is worth mentioning that `NoethAccS` is closed under quotients (implemented as inductive-like types à la Hofmann [6]) while `NoethAcc` is not.

In constructive mathematics, one encounters several further standard ways of expressing finiteness, e.g., Dedekind finiteness. Clearly, one can express finiteness in also in many exotic ways. We wonder whether some form of unifying theory of useful notions finiteness in constructive mathematics is possible.

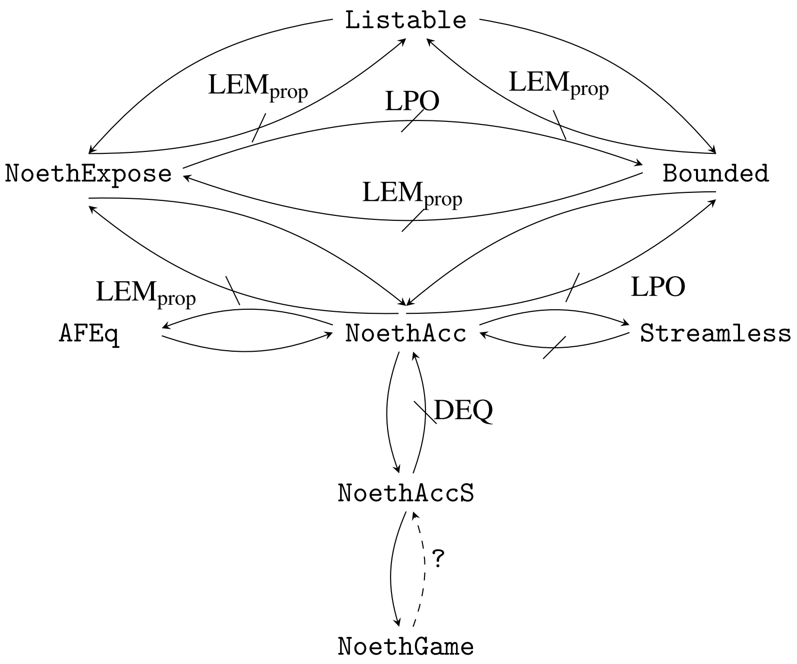


Figure 1: Variations on Noetherianness. LEM$_{\text{prop}}$, LPO, and DEQ denote the law of excluded middle for propositions, the limited principle of omniscience, and the decidable equality for all types.

Acknowledgement This research was supported by the Estonian Ministry of Education and Research institutional research grant no. IUT33-13, the Estonian Science Council personal research grant no. PUT763 and the Estonian Science Foundation grant no. 9475.

References

[1] The Agda Team (2015): *The Agda Wiki*. Available at `http://wiki.portal.chalmers.se/agda/`

[2] M. Bezem, T. Coquand & K. Nakata (2013): *Are streamless sets Noetherian?* In: *Abstracts of 19th Int. Conf. on Types for Proofs and Programs*, Inst. de Rech. en Inform. de Toulouse, pp. 32–33. `http://www.irit.fr/TYPES2013/TYPES2013BookOfAbstracts.pdf`

[3] M. Bezem, K. Nakata & T. Uustalu (2012): *On streams that are finitely red*. *Log. Meth. in Comput. Sci.* 8(4), article 4. doi: 10.2168/lmcs-8(4:4)2012

[4] T. Coquand & A. Spiwack (2010): *Constructively finite?* In L. Lambán, A. Romero & J. Rubio, eds.: *Contribuciones científicas en honor de Mirian Andrés Gómez*, Universidad de La Rioja, pp. 217–230. Available at `http://dialnet.unirioja.es/servlet/fichero_articulo?codigo=3217816`

[5] D. Firsov & T. Uustalu (2015): *Dependently typed programming with finite sets*. In: *Proc. of 11th ACM SIGPLAN Wksh. on Generic Programming, WGP '15*, ACM Press, pp. 33–44. doi: 10.1145/2808098.2808102

[6] M. Hofmann (1997): *Extensional Constructs in Intensional Type Theory*. *CPHS/BCS Distinguished Dissertations*. Springer, London. doi: 10.1007/978-1-4471-0963-1

[7] K. Kuratowski (1920): *Sur la notion d'ensemble fini*. *Fund. Math.* 1(1), pp. 129–131. Available at `https://eudml.org/doc/212596`

[8] U. Norell (2009): *Dependently typed programming in Agda*. In P. Koopman, R. Plasmeijer & S. D. Swierstra, eds.: *Revised Lectures from 6th Int. School on Advanced Functional Programming, AFP 2008*, *Lect. Notes in Comput. Sci.* 5832, Springer, pp. 230–266. doi: 10.1007/978-3-642-04652-0_5

[9] E. Parmann (2015): *Investigating streamless sets*. In H. Herbelin, P. Letouzey & M. Sozeau, eds.: *20th International Conference on Types for Proofs and Programs*, *Leibniz Int. Proc. in Informatics* 39, Dagstuhl Publishing, pp. 187–201. doi: 10.4230/lipics.types.2014.187

[10] W. Veldman & M. Bezem (1993): *Ramsey's theorem and the pigeonhole principle in intuitionistic mathematics*. *J. of London Math. Soc.* s2-47(2), pp. 193–211. doi: 10.1112/jlms/s2-47.2.193

[11] D. Vytiniotis, T. Coquand & D. Wahlstedt (2012): *Stop when you are almost-full*. In L. Beringer & A. Felty, eds.: *Proc. of 6th Int. Conf. on Interactive Theorem Proving, ITP 2012*, *Lect. Notes in Comput. Sci.* 7406, Springer, pp. 250–265. doi: 10.1007/978-3-642-32347-8_17

A Basic Definitions in Agda

Membership in a list We define the inductive type of proofs that an element `x` is in a list `xs`.

```
data  _∈_ {X : Set} (x : X) : List X → Set where
  here  : {xs : List X} → x ∈ x :: xs
  there : {y : X} {xs : List X} → x ∈ xs → x ∈ y :: xs
```

The constructor `here` says that the head of a list is a member of the list, and the constructor `there` says that any element of the tail of a list is also an element of the entire list.

Duplicates in a list We define the type of proofs that the list contains duplicates:

```
data Dup {X : Set} : List X → Set where
  duphere  : {x : X} {xs : List X} → x ∈ xs → Dup (x :: xs)
  dupthere : {x : X} {xs : List X} → Dup xs → Dup (x :: xs)
```

The list `x :: xs` contains duplicates, if the element `x` appears in the tail `xs` or if `xs` contains duplicates.

Generalized types for membership and duplicates in a list An inhabitant of the type `x ∈ xs` is a proof that the list `xs` contains an element *equal* to `x`. This can generalized by replacing equality by an arbitary binary relation `R`:

```
data  MemR {X : Set}(R : X → X → Set) (x : X) : List X → Set where
  here  : {y : X} {xs : List X} → R y x → MemR R x (y :: xs)
  there : {y : X} {xs : List X} → MemR R x xs → MemR R x (y :: xs)
```

The type `MemR R x xs` contains proofs that the element `x` is related by `R` to some element in the list `xs`. A similar generalization is possible for duplicates:

```
data DupR {X : Set}(R : X → X → Set) : List X → Set where
  duphere  : {x : X} {xs : List X} → MemR R x xs → DupR R (x :: xs)
  dupthere : {x : X} {xs : List X} → DupR R xs → DupR R (x :: xs)
```

The type `DupR R xs` contains proofs that the list `xs` contains a pair of elements related by `R`.

Decidable equality If `P` is a set, then `Dec P` is a type of proofs of `P` or not `P`:

```
data Dec (P : Set) : Set where
  yes : (prf :   P)  → Dec P
  no  : (prf : ¬ P) → Dec P
```

Here `yes` and `no` are two constructors of `Dec P`. The former takes a proof of `P` as its argument while the latter takes a proof of `¬ P` (i.e., `P → ⊥`).

Now, we say that `X` has decidable equality, if, for any `x₁` and `x₂` of type `X`, we have `Dec (x₁ ≡ x₂)`:

```
DecEq : Set → Set
DecEq X = (x₁ x₂ : X) → Dec (x₁ ≡ x₂)
```

Propositional types We say that the type `X` is a proposition, if any two elements of `X` are equal, i.e., if the type has at most one element:

```
isProp : Set → Set
isProp X = (x₁ x₂ : X) → x ≡ x₂
```

For example, the empty and unit types are propositions.

Function extensionality Two functions are extensionally equal, if they return the same value when applied to the same input. The principle of function extensionality asserts that two functions are equal, if they are extensionally equal.

```
funext : {X Y : Set} {f g : X → Y} → ((x : X) → f x ≡ g x) → f ≡ g
```

The principle of function extensionality is assumed in the proof of `NoethAccS→NoethSet` and in Section 3.1.

Membership and duplicates in a stream Streams are "infinite lists", defined coinductively as follows:

```
data Stream {X : Set} : Set where
  _::_ : X → ∞ (Stream X) → Stream X
```

The membership relation and the predicate of duplicates are defined similarly to those for lists.

```
data _∈S_ {X : Set} (x : X) : Stream X → Set where
  here  : {xs : ∞ (Stream X)} → x ∈S x :: xs
  there : {y : X} {xs : ∞ (Stream X)} → x ∈S ♭ xs → x ∈S y :: xs

data DupS {X : Set} : Stream X → Set where
  duphere  : {x : X} {xs : ∞ (Stream X)} → x ∈S ♭ xs → DupS (x :: xs)
  dupthere : {x : X} {xs : ∞ (Stream X)} → DupS (♭ xs) → DupS (x :: xs)
```

Membership, duplicates and finite length for colists Colists are "possibly infinite lists", defined coinductively as follows:

```
data Colist {X : Set} : Set where
  []   : Colist X
  _::_ : X → ∞ (Colist X) → Colist X
```

The membership relation and the predicate of duplicates are defined similarly to those for lists and streams.

```
data _∈C_ {X : Set} (x : X) : Colist X → Set where
  here  : {xs : ∞ (Colist X)} → x ∈C x :: xs
  there : {y : X} {xs : ∞ (Colist X)} → x ∈C ♭ xs → x ∈C y :: xs
```

```
data DupC {X : Set} : Colist X → Set where
  duphere  : {x : X} {xs : ∞ (Colist X)} → x ∈C ♭ xs → DupC (x :: xs)
  dupthere : {x : X} {xs : ∞ (Colist X)} → DupC (♭ xs) → DupC (x :: xs)
```

A colist has finite length, if it is a list after all. Formally, a colist has finite length, if it satisfies the following inductively defined predicate:

```
data _⇓ {X : Set} : Colist X → Set where
  []   : [] ⇓
  _::_ : (x : X) {xs : ∞ (Colist X)} → (♭ xs) ⇓ → (x :: xs) ⇓
```