Dependently Typed Programming with Finite Sets

Denis Firsov Tarmo Uustalu

Institute of Cybernetics at Tallinn University of Technology
Akadeemia tee 21, 12618 Tallinn, Estonia
{denis,tarmo}@cs.ioc.ee

Abstract

Definitions of many mathematical structures used in computer science are parametrized by finite sets. To work with such structures in proof assistants, we need to be able to explain what a finite set is. In constructive mathematics, a widely used definition is listability: a set is considered to be finite, if its elements can be listed completely. In this paper, we formalize different variations of this definition in the Agda programming language. We develop a toolbox for boilerplate-free programming with finite sets that arise as subsets of some base set with decidable equality. Among other things we implement combinators for defining functions from finite sets and a prover for quantified formulas over decidable properties on finite sets.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.2.4 [*Software Engineering*]: Software/Program Verification—correctness proofs; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords certified programming; finite sets; dependently typed programming; Agda; Kuratowski finiteness; Bishop finiteness

1. Introduction

Many definitions of structures used in computer science are parametrized by finite sets. For example, in the theory of formal languages, a deterministic finite automaton is defined as a 5-tuple

$$\mathtt{M = (Q, \Sigma, \delta, q_0, F)},$$

where `Q` is a finite set of states, `Σ` is a finite set of letters (alphabet), `δ` is a transition function from `Q × Σ` to `Q`, $\mathtt{q_0}$ is an initial state and `F` is a set of accepting states. To work with such concepts in proof assistants like Agda [13], which is the language we use in this paper, we need to be able to say what a finite set is.

One standard way to state that some set `X` is finite is to provide a list containing all elements of `X`. In our example, if the alphabet is binary (`Σ :=` $\mathbb{B}$,), then the list `false :: true :: []` together with a proof that every truth value is contained in this list establish finiteness of `Σ`. Another (equivalent) option is to provide a surjection from the set $[0..n)$ for some $n \in \mathbb{N}$. In our case, we can do with the function from $[0..2)$ that sends 0 to `false` and 1 to `true` together with a proof that this function is surjective.

In what follows, we define an example taken from quantum computing [12], the Pauli group on 1 qubit (with the global phase quotiented out), as a datatype with 4 nullary constructors. We also implement equality decision and the group operation to highlight the weak points of this straightforward approach and show the boilerplate code that we would like to reduce.

1.1 Extended Example

A finite set like the Pauli group can be defined as a datatype with a nullary constructor for each element:

```
data Pauli : Set where
  X : Pauli
  Y : Pauli
  Z : Pauli
  I : Pauli
```

The constructors `X`, `Y`, `Z`, and `I` denote the four distinct elements of the set `Pauli`.

To show that `Pauli` is finite (so that one can, for example, iterate through all elements), we can provide a list:

```
listPauli : List Pauli
listPauli = X :: Y :: Z :: I :: []
```

We can prove that the list is complete:

```
allPauli : (x : Pauli) → x ∈ listPauli
allPauli X = here
allPauli Y = there here
allPauli Z = there (there here)
allPauli I = there (there (there here))
```

(Here, `here` is a proof of `x ∈ x :: xs`; and `there p` is a proof of `x ∈ y :: xs`, if `p` is a proof that `x ∈ xs`.)

We could also prove that this list does not contain duplicates, but this is not mandatory.

We continue our example by implementing equality decision for elements of `Pauli`:

```
_≡P?_ : (x₁ x₂ : Pauli) → x₁ ≡ x₂ ⊎ ¬ (x₁ ≡ x₂)
X ≡P? X = inj₁ refl
X ≡P? Y = inj₂ λ()
X ≡P? Z = inj₂ λ()
X ≡P? I = inj₂ λ()
Y ≡P? X = inj₂ λ()
Y ≡P? Y = inj₁ refl
Y ≡P? Z = inj₂ λ()
Y ≡P? I = inj₂ λ()
Z ≡P? X = inj₂ λ()
Z ≡P? Y = inj₂ λ()
Z ≡P? Z = inj₁ refl
Z ≡P? I = inj₂ λ()
```

```
I ≡P? X = inj₂ λ()
I ≡P? Y = inj₂ λ()
I ≡P? Z = inj₂ λ()
I ≡P? I = inj₁ refl
```

(`¬ P = P → ⊥`, where ⊥ is the empty set; `X ⊎ Y` denotes the disjoint sum of `X` and `Y`. `()` is called the absurd pattern and denotes impossibility of a pattern.)

To conclude our example, we define the group operation:

```
_·_ : Pauli → Pauli → Pauli
X · X = I
X · Y = Z
X · Z = Y
Y · X = Z
Y · Y = I
Y · Z = X
Z · X = Y
Z · Y = X
Z · Z = I
x · I = x
I · x = x
```

And we prove that it is commutative:

```
·-comm : (x₁ x₂ : Pauli) → x₁ · x₂ ≡ x₂ · x₁
·-comm X X = refl
·-comm X Y = refl
·-comm X Z = refl
·-comm X I = refl
·-comm Y X = refl
·-comm Y Y = refl
·-comm Y Z = refl
·-comm Y I = refl
·-comm Z X = refl
·-comm Z Y = refl
·-comm Z Z = refl
·-comm Z I = refl
·-comm I X = refl
·-comm I Y = refl
·-comm I Z = refl
·-comm I I = refl
```

It is important to realize that `refl` takes different implicit arguments in different lines of the code above, so it cannot be shortened to just one line `·-comm _ _ = refl`. Actually, the code shown is the shortest "direct" proof and requires full pattern matching. It is easy to see that an associativity proof requires 64 lines of code.

We can see that the straightforward way of defining a finite set as an enumeration type has a number of shortcomings:

1. When defining `Pauli` and `listPauli`, we effectively listed all elements twice.
2. The proof of `allPauli` is verbose and dependent on the order of elements in the list `listPauli`. All three definitions (`Pauli`, `listPauli`, `allPauli`) must be kept consistent at all times, when modifying the code.
3. The equality decider is not derived automatically and the manual definition is verbose. The same would apply to duplicate-freeness decision, if we wanted to implement it.
4. The proof of commutativity of the `_·_` operation is dull, but cannot be compressed.

Alternatively, to show that `Pauli` is finite, we can provide a surjection from an initial segment of natural numbers. Let us first introduce a family of sets for initial segments of the set of all natural numbers. `Fin n` represents the set of first `n` natural numbers, i.e., the set of all numbers smaller than `n`.

```
data Fin : ℕ → Set where
  fzero : {n : ℕ} → Fin (suc n)
  fsuc  : {n : ℕ} → Fin n → Fin (suc n)
```

(In Agda, an argument enclosed in curly braces is implicit. The Agda type-checker will try to figure it. If an argument cannot be inferred, it must be provided explicitly.) `fzero` is smaller than `suc n` for any `n` and, if `i` is smaller than `n`, then `fsuc i` is smaller than `suc n`. As there is no number smaller than zero, `Fin zero` is empty. (When there are no possible constructor patterns for a given argument, one can pattern match on it with the absurd pattern `()`.)

Now, to show that `Pauli` is finite, we can define a function from `Fin 4` to `Pauli`:

```
f2p : Fin 4 → Pauli
f2p fzero = X
f2p (fsuc fzero) = Y
f2p (fsuc (fsuc fzero)) = Z
f2p (fsuc (fsuc (fsuc fzero))) = I
f2p (fsuc (fsuc (fsuc (fsuc ()))))
```

We can also define a function in the converse direction:

```
p2f : Pauli → Fin 4
p2f X = fzero
p2f Y = fsuc fzero
p2f Z = fsuc (fsuc fzero)
p2f I = fsuc (fsuc (fsuc fzero))
```

This allows us show that `f2p` is surjective by establishing

```
f2p-surj : (x : Pauli) → f2p (p2f x) ≡ x
f2p-surj X = refl
f2p-surj Y = refl
f2p-surj Z = refl
f2p-surj I = refl
```

In fact, `f2p` is not only a surjection, but even a bijection, but this is not mandatory for finiteness. We have once more established that `Pauli` is finite, however we introduced even more dependencies than when defining `Pauli`, `listPauli`, and `allPauli`. Namely, now the definitions of `Pauli`, `f2p`, `p2f`, and `f2p-surj` must all be kept in agreement with each other.

In this paper, we set out to explore finite sets in the constructive and dependently typed setting of Agda and develop an infrastructure for clean programming with finite sets.

In Section 2, we give some basic definitions regarding decidable propositions and also introduce effective squashing for such propositions.

In Section 3, we introduce some notions of constructive finiteness of a set and of a subset of a set and explore how they are related to each other. We present some conditions under which equality on a finite set is decidable. Furthermore, we show that there are finite subsets for which equality is undecidable.

Section 4 is devoted to a pragmatic approach for programming with finite sets that arise as subsets of a base set with decidable equality. In this approach, we are able to define a finite subset by listing its elements once and automatically derive completeness and decidable equality.

In Section 5, we show that the union, intersection, product and disjoint sum of finite sets are finite.

In Section 6.1, we show that functions from finite sets can be defined via tables. After that in Section 6.2, we introduce the notion of predicate matching and show how it can be used for defining functions on finite sets.

In Section 7, we implement a prover for quantified formulas over decidable properties on finite sets.

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/finset/`.

2. Basic Definitions

The predicate `All X` states that a given list `xs` contains all elements of a set `X` (duplicates being allowed):

```
All : (X : Set) → List X → Set
All X xs = (x : X) → x ∈ xs
```

A proposition `P` is called decidable, if there is a proof of either `P` or not `P`:

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

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

Now, we say that a set `X` has decidable equality, if there is a function sending any elements x_1 and x_2 of `X` to a proof of `Dec (x₁ ≡ x₂)`:

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

With these notations, the type of `_≡P?_` from Section 1 can be abbreviated to `DecEq Pauli`.

Similarly, we can define decidable list membership:

```
DecIn : (X : Set) → Set
DecIn X = (x : X) → (xs : List X) → Dec (x ∈ xs)
```

A proof of `DecIn X` is a function that, for any element `x : X` and a list `xs : List X`, returns a proof of either `x ∈ xs` or its negation. It is easy to verify that `DecEq X` and `DecIn X` are equivalent, namely:

```
deq2din : {X : Set} → DecEq X → DecIn X

din2deq : {X : Set} → DecIn X → DecEq X
```

We also define a notion of a proposition `P` being a mere proposition:

```
Prop : Set → Set
Prop P = (p₁ p₂ : P) → p₁ ≡ p₂
```

It says that `P` can have at most one proof.

Another basic predicate is `NoDup`, which expresses that a given list `xs` is duplicate-free:

```
NoDup : {X : Set} → List X → Set
NoDup {X} xs = (x : X) → Prop (x ∈ xs)
```

Duplication-freeness of `xs` is the same as there being at most one proof of membership in `xs` for every `x : X`.

If `X` has decidable equality, then `All X` and `NoDup` are decidable:

```
deq2dall : {X : Set} → DecEq X
  → (xs : List X) → Dec (All X xs)

deq2dnd : {X : Set} → DecEq X
  → (xs : List X) → Dec (NoDup xs)
```

If `P` is decidable, we can effectively define a squashed version of `P` (i.e., quotient `P` by the total equivalence relation):

```
‖_‖ : {P : Set} → Dec P → Set
‖ yes _ ‖  = ⊤
‖ no  _ ‖  = ⊥
```

(Here ⊤ is the unit type: a singleton type with a unique element `tt`.) Note that we are squashing `P`, not `Dec P`, however, we make use of a proof that `P` is decidable. For example, we can observe that the type

```
X ∈ X :: Y :: X :: []
```

is decidable. Moreover, there are two different proofs:

```
prf₁ : Dec (X ∈ X :: Y :: X :: [])
prf₁ = yes here

prf₂ : Dec (X ∈ X :: Y :: X :: [])
prf₂ = yes (there (there here))
```

So we can squash the type `X ∈ X :: Y :: X :: []` in two different ways: `‖ prf₁ ‖` or `‖ prf₂ ‖`, but both evaluate to ⊤.

It is easy to see that any two elements of a squashed type are equal:

```
propSq : {P : Set} → (d : Dec P) → Prop ‖ d ‖
```

It is also important to note that one can always get a proof of `P`, if the squashed version is inhabited:

```
fromSq : {P : Set} → (d : Dec P) → {‖ d ‖} → P
```

We have made the third argument (of type `‖ d ‖`) implicit, since if `d` proves `Dec P`, then the only possible value is the unique element `tt : ⊤` and the type-checker can derive it automatically.

3. Finiteness Constructively

3.1 Listable Sets

The best known and most used constructive notion of finiteness of a set is listability (also sometimes called Kuratowski finiteness, although Kuratowski [11] phrased his definition in different terms): a set is finite, if its elements can be completely listed:

```
Listable : (X : Set) → Set
Listable X = Σ[ xs ∈ List X ]
               All 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.)

A close alternative idea is to require a surjection from an initial segment of the set of natural numbers:

```
FinSurj : (X : Set) → Set
FinSurj X = Σ[ n ∈ ℕ ]
              Σ[ fromFin ∈ (Fin n → X) ]
              Σ[ toFin   ∈ (X → Fin n) ]
              ((x : X) → fromFin (toFin x) ≡ x)
```

The two notions are equivalent:

```
surj2lstbl : {X : Set}
  → FinSurj X → Listable X

lstbl2surj : {X : Set}
  → Listable X → FinSurj X
```

It is clear that, from listability of a set, one can learn an upper bound on the number of its elements. (But in fact one can learn also the actual cardinality, just wait a little.)

An a priori trimmer version of listability (sometimes called Bishop-finiteness [6]) forbids duplicates:

```
ListableNoDup : (X : Set) → Set
ListableNoDup X = Σ[ xs ∈ List X ]
                    All X xs ×
                    NoDup xs
```

Alternatively, one may require a bijection from an initial segment of the set of natural numbers:

```
FinBij : (X : Set) → Set
FinBij X = Σ[ n ∈ ℕ ]
           Σ[ fromFin ∈ (Fin n → X) ]
           Σ[ toFin ∈ (X → Fin n) ]
           ((x : X) → fromFin (toFin x) ≡ x) ×
           ((i : Fin n) → toFin (fromFin i) ≡ i)
```

These two notions of finiteness are also equivalent:

```
bij2lstblnd : {X : Set}
  → FinBij X → ListableNoDup X

lstblnd2bij : {X : Set}
  → ListableNoDup X → FinBij X
```

Quite clearly, from duplicate-free listability of a set, one can extract its exact cardinality.

It is less obvious that all four notions of finiteness are equivalent. The reason is that equality on a listable set is decidable:

```
lstbl2deq : {X : Set} → Listable X → DecEq X
```

We give the main idea behind the implementation of `lstbl2deq`. If a set `X` is listable, then there exist a list `xs` and a function `cmplt` that, for any element `x : X`, returns a proof that `x ∈ xs`. You can think of this proof as a position of `x` in `xs`. Therefore, for any value `x`, there is a split of `xs` such that $\mathtt{xs} \equiv \mathtt{xs}_1$ `++ x :: ` $\mathtt{xs}_2$. Now, if we need to check whether $\mathtt{x}_1$ `: X` and $\mathtt{x}_2$ `: X` are equal, we can proceed as follows. We ask `cmplt` for two splits of `xs`: `cmplt` $\mathtt{x}_1$ gives a split $\mathtt{xs} \equiv \mathtt{xs}_1$ `++` $\mathtt{x}_1$ `::` $\mathtt{xs}_2$ and `cmplt` $\mathtt{x}_2$ gives a split $\mathtt{xs} \equiv \mathtt{xs}'$ `++` $\mathtt{x}_2$ `::` $\mathtt{xs}''$. Next, it is clear that if

$$\mathtt{length}\ \mathtt{xs}_1 \equiv \mathtt{length}\ \mathtt{xs}'$$

then $\mathtt{x}_1 \equiv \mathtt{x}_2$. But what if $\mathtt{length}\ \mathtt{xs}_1 \not\equiv \mathtt{length}\ \mathtt{xs}'$? If the list `xs` were guaranteed to be duplicate-free, then this would immediately imply that $\mathtt{x}_1 \not\equiv \mathtt{x}_2$, since there would then be a bijection between positions of `xs` and elements of `X`. However, in the presence of duplicates, this argument does not work. Instead, we observe that `cmplt` is a function of type `(x : X) → x ∈ xs`. Therefore, for equal elements of `X`, it must deliver equal results. With this observation, we can argue that if `length` $\mathtt{xs}_1$ is not equal to `length` $\mathtt{xs}'$ then $\mathtt{x}_1 \not\equiv \mathtt{x}_2$. Indeed, if $\mathtt{x}_1 \equiv \mathtt{x}_2$, then `cmplt` $\mathtt{x}_1$, and `cmplt` $\mathtt{x}_2$ must give the same split contradicting $\mathtt{length}\ \mathtt{xs}_1 \not\equiv \mathtt{length}\ \mathtt{xs}'$.

As soon as list membership is decidable, we can implement removal of duplicates:

```
remDup : {X : Set} → DecIn X → List X → List X
```

We show that `remDup` is complete. Namely, if some element belongs to the list, then at least one copy of that element is preserved by `remDup`:

```
remDupComplete : {X : Set} → (_∈?_ : DecIn X)
  → (x : X) → (xs : List X)
  → x ∈ xs → x ∈ remDup ∈? xs
```

`remDup` is also sound—the resulting list does not contain any new elements:

```
remDupSound : {X : Set} → (_∈?_ : DecIn X)
  → (x : X) → (xs : List X)
  → x ∈ remDup ∈? xs → x ∈ xs
```

And most importantly, the resulting list is free of duplicates:

```
remDupProgress : {X : Set} → (_∈?_ : DecIn X)
  → (xs : List X) → NoDup (remDup ∈? xs)
```

Now, with `lstbl2deq` and `deq2din`, we can prove that `Listable X` implies `ListableNoDup X`, and the converse is a triviality:

```
lstbl2lstblnd : {X : Set}
  → Listable X → ListableNoDup X

lstblnd2lstbl : {X : Set}
  → ListableNoDup X → Listable X
```

It is worth noticing that the proof `lstbl2deq` also provides an alternative definition of an equality decider for listable types like `Pauli` from Section 1:

```
listablePauli : Listable Pauli
listablePauli = listPauli , allPauli

deqPauli : DecEq Pauli
deqPauli = lstbl2deq listablePauli
```

Remember that the direct approach for defining decidable equality on `Pauli` required us 4^2 lines of code.

3.2 Listable Subsets

A special case of sets are those defined as a subset of a larger set. Here we have more variations of finiteness.[1].

A subset of a base set `U` carved out by a predicate `P : U → Set` is called subfinite, if there is a list containing all elements of `U` that satisfy `P` (we call this property completeness):

```
ListableJunkSub : (U : Set) → (U → Set) → Set
ListableJunkSub U P = Σ[ xs ∈ List U ]
                        ((x : U) → P x → x ∈ xs)
```

This notion of finiteness (which can only be formulated for subsets of some base set, not for general sets) allows `xs` to contain also elements not satisfying `P`. Therefore, we cannot even know whether the subset is empty. But we have an immediate upper bound on the number of elements in the subset: it is the length of the list `xs`.

A stronger notion of finiteness requires also soundness, i.e., a proof that an element of `U` belongs to `xs` only if it satisfies the predicate `P` (duplicates are still allowed):

```
ListableSub : (U : Set) → (U → Set) → Set
ListableSub U P = Σ[ xs ∈ List U ]
                    ((x : U) → P x → x ∈ xs) ×
                    ((x : U) → x ∈ xs → P x)
```

A listable subset can be checked for emptiness:

```
empty? : {U : Set}{P : U → Set}
  → (p : ListableSub U P)
  → Dec ((x : U) → ¬ x ∈? proj₁ p))
```

Listable sets are a special case of listable subsets:

```
lstbl2lsub : {U : Set}
  → Listable U → ListableSub U (λ _ → ⊤)

lsub2lstbl : {U : Set}
  → ListableSub U (λ _ → ⊤) → Listable U
```

The always true predicate `(λ _ → ⊤)` gives us the whole set `U` as the subset, i.e., the base set `U` must itself be listable. This is a special case of the situation where `P` has at most one proof for every element `x` of `U` (`P x` is a mere proposition):

```
prop2lstbl2lsub : {U : Set}{P : U → Set}
```

[1] We will generally speak of finiteness of a subset without actually constructing this subset as a set in its own right, since that would require us to be able to squash arbitary propositions, not just decidable ones.

```
  → ((x : U) → Prop (P x))
  → Listable (Σ[ x ∈ U ] P x)
  → ListableSub U P

prop2lsub2lstbl : {U : Set}{P : U → Set}
  → ((x : U) → Prop (P x))
  → ListableSub U P
  → Listable (Σ[ x ∈ U ] P x)
```

3.3 Decidability of Equality on Listable Subsets

Let us define decidability of equality on the subset of U determined by P as decidability of equality on U restricted to the elements satisfying P:

```
DecEqSub : (U : Set) → (P : U → Set) → Set
DecEqSub U P
  = (x₁ x₂ : U) → P x₁ → P x₂ → Dec (x₁ ≡ x₂)
```

In Section 3, we showed that listability of X implies decidable equality on X. Now we give a more general version of that property, namely, if, for any `x : U` there is at most one proof of `P x`, then equality on the subset given by U and P is decidable.

```
deqLstblSub1 : {U : Set}
   → (P : U → Set)
   → ListableSub U P
   → ((x : U) → Prop (P x))
   → DecEqSub U P
```

The strategy of implementing `deqLstblSub1` is similar to the strategy of implementing `lstbl2deq`. If P defines a listable subset of U, then we have a list `xs` containing all elements of U such that P. We also have a proof of completeness of `xs`:

```
cmplt : (x : U) → P x → x ∈ xs.
```

If we want to check two elements x_1 and x_2 for equality, then we are also given proofs $p_1 : P\ x_1$ and $p_2 : P\ x_2$. Clearly, if $\mathtt{cmplt}\ x_1\ p_1$ and $\mathtt{cmplt}\ x_2\ p_2$ induce the same splits of `xs`, namely, $xs \equiv xs_1 \mathbin{++} x_1 :: xs_2$, $xs \equiv xs' \mathbin{++} x_2 :: xs''$ and $\mathtt{length}\ xs_1 \equiv \mathtt{length}\ xs'$, then $x_1 \equiv x_2$. However, in the case when the splits are different, we cannot use the argument that, since `cmplt` is a function, there is only one split for each element. The reason is that, generally, `cmplt x` may deliver different splits for different proofs of `P x`. However, we have required that there is a unique proof of `P x` for any `x`. Finally, we can conclude that, if $\mathtt{length}\ xs_1 \not\equiv \mathtt{length}\ xs_2$, then $x_1 \not\equiv x_2$.

Actually, in this situation of P being a mere proposition, the intended subset can be explicitly defined as the set `Σ[ x ∈ U ] P x`, and we have decidable equality on this set:

```
deqLstblSub1' : {U : Set}
   → (P : U → Set)
   → ListableSub U P
   → ((x : U) → Prop (P x))
   → (xp₁ xp₂ : Σ[ x ∈ U ] P x)
   → Dec (xp₁ ≡ xp₂)
```

Equality on the subset is also decidable, if P is decidable:

```
deqLstblSub2 : {U : Set}
   → (P : U → Set)
   → ((x : U) → Dec (P x))
   → ListableSub U P
   → DecEqSub U P
```

A further variation says that, if we know that the list of all elements of U satisfying the predicate P is duplicate-free, then we also have decidable equality on the subset:

```
deqLstblSub3 : {U : Set}
   → (P : U → Set)
   → (p : ListableSub U P)
   → NoDup (proj₁ p)
   → DecEqSub U P
```

We conclude with a proof that there is no function turning any proof of `ListableSub U P` into a decider of equality on elements of U satisfying P:

```
deqLstblSub4 : {U : Set}
   → (P : U → Set)
   → ListableSub U P
   → DecEqSub U P
deqLstblSub4 = ???
```

Let us define the following list of functions from booleans to booleans:

```
listB2B : List (Bool → Bool)
listB2B = fun₁ :: fun₂ :: fun₃ :: []
  where
    fun₁ : Bool → Bool
    fun₁ _ = true

    fun₂ : Bool → Bool
    fun₂ _ = false

    fun₃ : Bool → Bool
    fun₃ b = if b then true else true
```

The list `listB2B` consists of three functions. The functions $\mathtt{fun}_1$ and $\mathtt{fun}_3$ always return `true`, however they are not propositionally equal, unless we assume function extensionality. The function $\mathtt{fun}_2$ always returns `false`.

Then we specify a subset of functions of type `Bool → Bool` by the following predicate B2B:

```
B2B : (Bool → Bool) → Set
B2B f = f ∈ listB2B
```

Next, we prove that the predicate B2B defines a listable subset. Clearly, it is just the set of functions from the list `listB2B`:

```
listableB2B : ListableSub (Bool → Bool) B2B
listableB2B = listB2B , (λ x p → p) , (λ x p → p)
```

So, now we could try to write a function that decides equality of elements of `listableB2B`:

```
deqB2B : (f₁ f₂ : Bool → Bool)
   → B2B f₁
   → B2B f₂
   → Dec (f₁ ≡ f₂)
deqB2B = ???
```

Given f_1 and f_2 together with $p_1 : \mathtt{B2B}\ f_1$ and $p_2 : \mathtt{B2B}\ f_2$, we can pattern-match on p_1 and p_2. Some cases are unproblematic: e.g., if p_1 = `here` and p_2 = `here`, then $f_1 = \mathtt{fun}_1$ and $f_2 = \mathtt{fun}_1$, so it is trivial that $f_1 \equiv f_2$. Similarly, if p_1 = `here` and p_2 = `there here`, then $f_1 = \mathtt{fun}_1$ and $f_2 = \mathtt{fun}_2$ and hence $\neg\ (f_1 \equiv f_2)$. But there is the critical case of p_1 = `here` and p_2 = `there (there here)`. Then $f_1 = \mathtt{fun}_1$ and $f_2 = \mathtt{fun}_3$ and there is simply no correct answer to return, as the two functions are not equal propositionally (unless function extensionality is assumed), but also not inequal.

We have argued that it is impossible to prove `deqLstblSub4`. This also implies that the notion of a listable set is stronger than the notion of a listable subset, which in turn is stronger than the notion of a subset listable with junk.

4. Pragmatic Finite Sets

In this section, we aim at a pragmatic approach to programming with finite sets. Our objective is to be able to specify a finite set by listing the intended elements just once. From specification, we want to obtain a listable set with no additional work. Our solution is to specify the finite set as a subset of some base set with decidable equality.

4.1 Motivation and Definition

In Section 1, we saw that the straightforward approach to defining the Pauli group as a datatype with nullary constructors and proving that it is finite required us to list the elements of `Pauli` multiple times and also provide verbose proofs of completeness and decidability of equality. Next, we go through a number of steps, to motivate a more pragmatic approach.

As we have seen, a predicate `P` on a base set `U` defines a subset. If there is a list of elements containing all elements of `U` satisfying `P` and no others, then we have a listable subset:

```
step1 : {U : Set} → (P : U → Set)
  → (xs : List U)
  → ((x : U) → x ∈ xs → P x)
  → ((x : U) → P x → x ∈ xs)
  → ListableSub U P
```

Next, we observe that we can create a listable subset from any list `xs` over `U` by taking `P` to be `(λ x → x ∈ xs)`:

```
step2 : {U : Set} → (xs : List U)
  → ListableSub U (λ x → x ∈ xs)
step2 = xs , (λ x i → i) , (λ x i → i)
```

A good thing is that the proofs of soundness and completeness are now trivial. But the elements of the subset are dependent pairs of an element `x` of `U` and a proof of membership (position) of `x` in `xs`.

Next we can ask for decidable list membership on `U` to be able to effectively squash sets `x ∈ xs`:

```
step3 : {U : Set} → (_∈?_ : DecIn U)
  → (xs : List U)
  → ListableSub U (λ x → ‖ x ∈? xs ‖)
```

Now an element of the subset is a pair of an element of `U` and an element of a squashed type (which, if it exists, is unique!).

By theorem `prop2lsub2lstbl` from Section 3.2, the type

```
Σ[ x ∈ U ] ‖ x ∈? xs ‖
```

must be listable.

Given these considerations, we can define a datatype of descriptions of finite sets as subsets of a base set with decidable equality:

```
data FinSubDesc (U : Set) (eq : DecEq U) :
                              Bool → Set where

  fsd-plain : List U → FinSubDesc U eq true
  fsd-nodup : (xs : List U) → {‖ nd? xs ‖}
    → FinSubDesc U eq false
      where
        nd? = deq2dnd eq
```

The datatype introduced is parametrized by a base set `U`, a decider `eq` of equality on `U`, and is also indexed by a boolean flag `b` that indicates whether the underlying list of elements is allowed to contain duplicates. There are two constructors. The constructor `fsd-plain` takes a list `xs` of elements of `U` as an argument. The constructor `fsd-nodup` accepts a list `xs` as an argument only if it is duplicate-free. It has also another, implicit argument, of a squashed type. This type is inhabited if and only if `xs` contains no duplicates. In other words, if `xs` is duplicate-free, then the type of the implicit argument evaluates to the unit type and its value can be inferred automatically. If `xs` contains duplicates, then the type of the implicit argument evaluates to $\bot$ and no value can be provided for it.

There are pragmatic reasons to have two constructors for `FinSubDesc`. If the user creates a relatively small subset of elements ($\leq$ 10000) using `fsd-nodup`, then the type-checker can feasibly check that there are no duplicates. However, if the number of elements is larger, then the price for maintaining the invariant of no duplicates becomes too high. Remember that the complexity of checking duplicate-freeness is quadratic in the length of the list.

We can now define the Pauli group as a subset of the set of all characters:

```
MyPauli : FinSubDesc Char _≡C?_ false
MyPauli
  = fsd-nodup ('X' :: 'Y' :: 'Z' :: 'I' :: [])
```

Since the list provided is without duplicates, the type of the implicit argument

```
‖ nd? ('X' :: 'Y' :: 'Z' :: 'I' :: []) ‖
```

is evaluated to $\top$ by the type-checker and the value for this argument is derived automatically.

On the other hand, the following definition is rejected by the type-checker, since `'X'` is listed twice and the type of the implicit argument is evaluates to $\bot$:

```
MyPauliBad : FinSubDesc Char _≟_ false
MyPauliBad
  = fsd-nodup ('X' :: 'Y' :: 'Z' :: 'I' :: 'X' :: [])
                                              {???}
```

The hole needs to be filled with a proof of $\bot$, which is impossible. However, we can drop the requirement of no duplicates (note the change in the type):

```
MyPauliFixed : FinSubDesc Char _≟_ true
MyPauliFixed
  = fsd-plain ('X' :: 'Y' :: 'Z' :: 'I' :: 'X' :: [])
```

Now, we can define the actual set that a finite subset description denotes:

```
toList : {U : Set}{eq : DecEq U}{b : Bool}
  → FinSubDesc U eq b → List U
toList (fsd-plain xs) = xs
toList (fsd-nodup xs) = xs

Elem : {U : Set}{eq : DecEq U}{b : Bool}
  → FinSubDesc U eq b → Set
Elem {U} {eq} D
  = Σ[ x ∈ U ] ‖ x ∈? toList D ‖
    where
      _∈?_ = deq2din eq
```

So an element of type `Elem D` for some finite subset description `D` is a dependent pair of an element `x` of `U` together with a squashed proof that `x` belongs to the list of elements defining the subset. Using the squashed membership type allows us to ignore the exact position(s) of the element in the list.

For example, we could refer to one of the elements of `MyPauli` as the `identity`:

```
identity : Elem MyPauli
identity = ('I' , _)
```

The second component of the pair (the type-checker infers that it must be `tt`) is actually a squashed proof of the fact that `I` belongs to the set `MyPauli`. Without squashing, we would need to refer to

I by its position, namely,

```
('I', there (there (there here))).
```

Clearly, we want to avoid such fragile dependencies.

On the other hand, the type-checker will accept a non-element of the list only if the user manages to provide a proof of ⊥.

```
bad : Elem MyPauli
bad = ('W' , ???)
```

4.2 Finite Subsets are Listable

Our next step is to show that, for all `D : FinSubDesc U eq b`, the corresponding subset of `U`, namely, `Elem D`, is listable.

First, we generate a list of elements of `Elem D`:

```
listElem : {U : Set}{eq : DecEq U}{b : Bool}
  → (D : FinSubDesc U eq b)
  → List (Elem D)
```

Second, we show that `listElem D` is complete, it contains all elements of `Elem D`:

```
allElem : {U : Set}{eq : DecEq U}{b : Bool}
  → (D : FinSubDesc U eq b)
  → (xp : Elem D) → xp ∈ listElem D
```

Third, we observe that `listElem D` does not introduce any duplicates:

```
ndElem : {U : Set}{eq : DecEq U}
  → (D : FinSubDesc U eq false)
  → NoDup (listElem D)
```

Finally, we show that `Elem D` is listable:

```
lstblElem : {U : Set}{eq : DecEq U}{b : Bool}
  → (D : FinSubDesc U eq b)
  → Listable (Elem D)
lstblElem D = listElem D , allElem D
```

This also implies decidable equality on `Elem D`:

```
deqElem : {U : Set}{eq : DecEq U}{b : Bool}
  → (f : FinSubDesc U eq b)
  → DecEq (Elem D)
deqElem D = lstbl2deq (lstblElem D)
```

4.3 Finite Subsets from Lists

Now, we implement a function `fromList` which is parametrized by the boolean `b`, so that user could decide if the duplicates should be removed from the resulting finite subset:

```
fromList : {U : Set} → (eq : DecEq U)
  → (b : Bool) → List U → FinSubDesc U eq b
```

Basic set operations can now be defined on the underlying lists of finite subsets. For example, the union is defined by concatenating the underlying lists of argument subsets:

```
_∪_ : {U : Set}{eq : DecEq U}
  → {b₁ b₂ : Bool}
  → FinSubDesc U eq b₁
  → FinSubDesc U eq b₂
  → FinSubDesc U eq (b₁ ∧ b₂)
_∪_ {eq = eq} D₁ D₂
  = fromList eq _ (toList D₁ ++ toList D₂)
```

Here is an example:

```
MyNats1 = fsd-nodup (1 :: 3 :: [])
MyNats2 = fsd-nodup (1 :: 6 :: [])
p : MyNats1 ∪ MyNats2 ≡ fsd-nodup (1 :: 3 :: 6 :: [])
p = refl
```

4.4 Finite Subset Monad

Finite subsets (of sets with decidable equality) are monad. We explicate this structure on the level of `FinSubDesc`:

```
return : {U : Set}{eq : DecEq U}{b : Bool}
   → U → FinSubDesc U eq b

bind : {U V : Set}{eqU : DecEq U}
  → {eqV : DecEq V}{bU bV : Bool}
  → FinSubDesc U eqU bU
  → (U → FinSubDesc V eqV bV)
  → (b : Bool)
  → FinSubDesc V eqV b
```

A peculiarity of `return` and `bind` here is that they can work in two different modes. If the boolean argument provided is `false`, then duplicates will be removed the resulting finite subset description, otherwise not. This allows the user to tune the monadic code for the efficiency.

Wadler [16] identifies the structure needed for comprehending monads. The missing bit is mzero:

```
mzero : {U : Set}{eq : DecEq U}{b : Bool}
  → FinSubDesc U eq b
mzero {b = true}  = fsd-plain []
mzero {b = false} = fsd-nodup []
```

Using `mzero`, we define a conditional `if_then_` and also some syntactic sugar for `bind`:

```
if_then_ : {U : Set}{eq : DecEq U}{b : Bool}
  → Bool → FinSubDesc U eq b
  → FinSubDesc U eq b
if b then c = if b then c else mzero

syntax bind A (λ x → B) b
  = for x ∈ A as b do B
```

As a result, we can write set comprehension code in for-loop style. Let us look at an example. Mathematically, the intersection of sets `X` and `Y` is defined as:

$$X \cap Y = \{x \mid x \in X, y \in Y, x = y\}$$

With the combinators and syntactic sugar defined above, we can write the following definition of subset intersection with comprehensions:

```
_∩_ : {U : Set}{eq : DecEq U} {b₁ b₂ : Bool}
  → FinSubDesc U eq b₁ → FinSubDesc U eq b₂
  → FinSubDesc U eq (b₁ ∧ b₂)
_∩_ {eq = _≡?_} X Y =
          for x ∈ X as _  do
            for y ∈ Y as true do
              if ⌊ x ≡? y ⌋ then return x
```

5. Combinators

In this section, we define some general combinators for listable subsets. The simplest combinator is for taking the union of two listable subsets of the same base set:

```
union : {U : Set}{P Q : U → Set}
  → ListableSub U P
  → ListableSub U Q
  → ListableSub U (λ x → P x ⊎ Q x)
```

The definition just concatenates the underlying lists of the two subsets and then adapts the proofs of completeness and soundness.

The intersection of two listable subsets is trickier, since it cannot be defined generally for two arbitrary subsets. The reason is simple, we need somehow to find the common elements. One possibility is to ask equality on U to be decidable:

```
intersection' : {U : Set}{P Q : U → Set}
  → DecEq U
  → ListableSub U P
  → ListableSub U Q
  → ListableSub U (λ x → P x × Q x)
```

But this assumption can be weakened by only asking one of the predicates to be decidable.

```
intersection : {U : Set}{P Q : U → Set}
  → ((x : U) → Dec (P x))
  → ListableSub U P
  → ListableSub U Q
  → ListableSub U (λ x → P x × Q x)
```

This a weaker condition, because, if U has decidable equality, then `ListableSub U P` implies decidability of P:

```
deq2lstbl2dp : {U : Set}{P : U → Set}
  → DecEq U
  → ListableSub U P
  → (x : U) → Dec (P x)
```

We also prove that the product and the disjoint sum of listable subsets of two base sets are listable subsets of the product/disjoint sum of the base sets:

```
product : {U : Set}{P : U → Set}
  → {V : Set}{Q : V → Set}
  → ListableSub U P
  → ListableSub V Q
  → ListableSub (U × V) < P , Q >

sum : {U : Set}{P : U → Set}
  → {V : Set}{Q : V → Set}
  → ListableSub U P
  → ListableSub V Q
  → ListableSub (U ⊎ V) [ P , Q ]
```

6. Function Definition

This section describes two different approaches for defining functions from finite sets.

We observe that, if we want to define arbitrary functions from some finite X to Y, then we must be able to compare the elements of X and also for the function to be total we need the complete list of those. Therefore, the right notion of finiteness for X is listability (`Listable X`).

6.1 Tabulation

To define a function of type `f : X → Y` for some listable X, we could explicitly provide a list of pairs `(x , y)`. For example, if `X = { ▲ , ◆ , ■ }` and `Y = ℕ` then the list

```
(▲ , 1) :: (◆ , 10) :: (■ , 100) :: []
```

could be interpreted as a function:

```
f : X → ℕ
f ▲ = 1
f ◆ = 10
f ■ = 100
```

But not any list xys of type `List (X × Y)` can be turned into a function. We need two additional properties:

1. For the function f to be total, each element of the domain must appear in xys paired with some element of codomain. Formally, we require `All X (map proj₁ xys)`.
2. For unambiguous interpretation, the list xys must not contain multiple pairs with the same domain element. Formally, `NoDup (map proj₁ xys)`

For example:

```
bad1 = (▲ , 1) :: (◆ , 10) :: []
bad2 = (▲ , 1) :: (◆ , 10) :: (◆ , 0) :: []
```

The list bad1 violates the first requirement and the list bad2 violates both.

Now, we translate the above into Agda:

```
Tbl : Set → Set → Set
Tbl X Y = Σ[ xys ∈ List (X × Y) ]
          All X (map proj₁ xys) ×
          NoDup (map proj₁ xys)
```

An element of type `Tbl X Y` is a list of pairs of type `X × Y` with some additional information, namely, proofs that the list of pairs is complete and duplicate-free regarding the first components. Recall that `All X xs` implies `Listable X`.

Since, for small tables, proofs of `All X` and `NoDup` can be inferred by the type-checker, it makes sense to define the following function for creating tables:

```
lstbl2dall : {X : Set} → Listable X
     → (xs : List X) → Dec (All X xs)

lstbl2dnd : {X : Set} → Listable X
     → (xs : List X) → Dec (NoDup xs)

createTbl : {X Y : Set} → (p : Listable X)
  → (xys : List (X × Y))
  → {‖ lstbl2dall p (map proj₁ xys) ‖}
  → {‖ lstbl2dnd  p (map proj₁ xys) ‖}
  → Tbl X Y
```

If the list `map proj₁ xys` contains all the elements of type X and is without duplicates, then the implicit arguments need not be supplied manually, since their types will be evaluated to ⊤ by the type-checker, so that tt is the only possible value.

Next, we implement a function for tabulating functions from a listable set:

```
toTbl : {X Y : Set} → Listable X
  → (X → Y) → Tbl X Y
```

Likewise, tables are convertible into functions:

```
fromTbl : {X Y : Set} → Tbl X Y → X → Y
```

We also show that converting back and forth between the two representations of the function is harmless:

```
fromto : {X Y : Set}
  → (p : Listable X)
  → (f : X → Y)
  → (x : X)
  → fromTbl (toTbl p f) x ≡ f x
```

As a final example of this subsection, we write a conversion function from `Elem MyPauli` to `Pauli`:

```
_↦_ : {U Y : Set}{eq : DecEq U}{b : Bool}
     → {D : FinSubDesc U eq b}
     → (x : U)
     → {‖ x ∈? toList D ‖}
     → Y → (Elem D × Y)
```

```
toPauli : Elem MyPauli → Pauli
toPauli = fromTbl (createTbl (lstblElem MyPauli)
  ('X' ↦ X  ::
   'Y' ↦ Y  ::
   'Z' ↦ Z  ::
   'I' ↦ I  :: []))
```

6.2 Predicate Matching

Assume that X is some finite set. How to implement in Agda a function $f : X \to Y$ that is defined piecewise:

$$f(x) = \begin{cases} f_1(x) & \text{if } p_1(x) \\ f_2(x) & \text{if } p_2(x) \\ \dots & \\ f_n(x) & \text{if } p_n(x) \end{cases}$$

One possibility is to provide an explicit table as described in the previous section. Unfortunately, if X is large this approach requires a lot of manual work. Another possibility is to encode it directly by nesting `if_then_else_` expressions:

```
f x = if p₁ x then f₁ x
         else if p₂ x then f₂ x
             else if p₃ x then f₃ x
                else ...
```

This approach is more concise than giving an explicit table, but it suffers from several of drawbacks:

1. There is always the last `else` branch, which plays the role of a "default" case. It will be applied to all elements which do not satisfy the predicates $\mathtt{P}_1 \dots \mathtt{P}_n$. The "default" branch makes it difficult to discover that some case was forgotten by mistake.
2. There is no good way of checking that the predicates cover all elements of the finite set (i.e., that no elements in the domain reach the "default" branch).
3. Also it is difficult to find whether there are perhaps some "dead" branches which are not satisfied by any element of `X`.

In what follows, we address these issues and introduce a notion of predicate matching.

We start by implementing a function `unreached` that takes a list of predicates and a list of elements and returns the list of those predicates that are not satisfied by any element:

```
unreached : {X : Set} → List (X → Bool)
  → List X → List (X → Bool)
unreached [] xs = []
unreached (p :: ps) xs =
  if (any p xs) then rc else (p :: rc)
  where
    rc = unreached ps (filter (not ∘ p) xs)
```

It is important to note that at the recursive call we filter out the elements that are satisfied by the head of the list of predicates. It means that the order of predicates matters. A predicate is only reached, if there is at least one element that satisfies it but does not satisfy any preceding predicates. We formalize this in the following soundness theorem:

```
unreachedSound : {X : Set}
  → (ps₁ ps₂ : List (X → Bool))
  → (p : X → Bool)
  → (xs : List X)
  → unreached (ps₁ ++ p :: ps₂) xs ≡ []
  → Σ[ x ∈ X ] x ∈ xs × p x ≡ true ×
  ((p' : X → Bool) → p' ∈ ps₁ → p' x ≡ false)
```

Soundness states that, if for some list `xs`, the list of predicates `ps` contains no unreachable ones, then for any split of `ps` into three parts, `ps ≡ ps₁ ++ p :: ps₂`, there exists at least one element `x` that satisfies `p` but does not satisfy any of the predicates in `ps₁`.

On the other hand, completeness states that, if there exists an element `x` of the list `xs` that does not satisfy any of the predicates in `ps` and does satisfy some predicate `p`, then the list `ps ++ p :: []` is also reachable:

```
unreachedComplete : {X : Set}
  → (ps : List (X → Bool))
  → (xs : List X)
  → unreached ps xs ≡ []
  → (p : X → Bool)
  → (x : X)
  → x ∈ xs
  → p x ≡ true
  → ((p' : X → Bool) → p' ∈ ps → p' x ≡ false)
  → unreached (ps ++ p :: []) xs ≡ []
```

Now, let us address the issue of unmatched elements. We implement a function `unmatched` that returns the list of all those elements in a given list `xs` that do not satisfy any predicate in the given list `ps`:

```
isMatched : {X : Set} → List (X → Bool) →  X
  → Bool
isMatched ps x = any (λ p → p x) ps


unmatched : {X : Set} → List (X → Bool) → List X
  → List X
unmatched ps [] = []
unmatched ps (x :: xs) = if (isMatched ps x)
                         then unmatched ps xs
                         else x :: unmatched ps xs
```

The soundness theorem for the `unmatched` function states that, if there are no unmatched elements in the list `xs`, then, for any element `x` in `xs`, the list of predicates `ps` can be split into three parts, `ps ≡ ps₁ ++ p :: ps₂`, so that no predicate from `ps₁` is satisfied by `x` and `p` is satisfied by `x`:

```
unmatchedSound : {X : Set}
  → (ps : List (X → Bool))
  → (xs : List X)
  → unmatched ps xs  ≡ []
  → (x : X) → x ∈ xs
  → Σ[ ps₁ ∈ List (X → Bool) ]
    Σ[ ps₂ ∈ List (X → Bool) ]
    Σ[ p ∈ (X → Bool) ]
    ps₁ ++ p :: ps₂ ≡ ps ×
    isMatched ps₁ x ≡ false ×
    p x ≡ true
```

Completeness says that, if each element in the list `xs` satisfies at least one predicate in `ps`, then there are no unmatched elements:

```
unmatchedComplete : {X : Set}
  → (ps : List (X → Bool))
  → (xs : List X)
  → ((x : X) → x ∈ xs
          → Σ[ p ∈ (X → Bool) ]
             p ∈ ps × p x ≡ true)
 → unmatched ps xs ≡ []
```

We can now define a combinator that takes a list of predicates and functions from a listable set with proofs thats all predicates are reached and all elements matched, and returns a function built from the pieces:

```
predicateMatching : {X Y : Set}
  → (ps : List ((X → Bool) × (X → Y)))
  → (p : Listable X)
  → unmatched (map proj₁ ps) (proj₁ p) ≡ []
  → unreached (map proj₁ ps) (proj₁ p) ≡ []
  → X → Y
```

Let us look at some examples, but first, we want to have a combinator `fromPure` for restricting the domain of a function to a finite subset:

```
fromPure : {U Y : Set}{eq : DecEq U}{b : Bool}
  → {D : FinSubDesc U eq b}
  → (U → Y)
  → Elem D → Y
fromPure f (x , _) = f x
```

We define a finite subset of naturals `MyNats` containing five natural numbers.

```
MyNats : FinSubDesc ℕ _≟_ false
MyNats = fsd-nodup (1 :: 42 :: 3 :: 8 :: 17 :: [])
```

Next we define a function `even2odd3` that doubles the even and triples the odd numbers of `MyNats`:

```
even2odd3 : Elem MyNats → ℕ
even2odd3 = predicateMatching
  (fromPure odd  , (λ  (x , p) → x * 3)  ::
   fromPure even , (λ  (x , p) → x * 2)  :: [])
  (lstblElem MyNats) refl refl
```

The two last arguments (`refl`) are proofs of `[] ≡ []` and indicate that there are no unmatched elements and no unreachable predicates. However, if we remove the first equation

```
even2odd3Bad1 = predicateMatching
  (fromPure even , (λ  (x , p) → x * 2)  :: [])
  (lstblElem MyNats) ??? refl
```

then there are unmatched elements and the type-checker wants us to supply a proof of

```
(1 , tt) :: (3 , tt) :: (17 , tt) :: [] ≡ []
```

for the hole. The goal gives us a nice hint about which elements exactly are unmatched.

If instead we replace the first equation with a predicate which is satisfied by any element

```
even2odd3Bad2 = predicateMatching
  ((λ _ → true)  , (λ  (x , p) → x * 3)  ::
   fromPure even , (λ  (x , p) → x * 2)  :: [])
  (lstblElem MyNats) refl ???
```

then the type-checker asks us to prove that `fromPure even :: [] ≡ []`, which again hints which equations are unreachable.

7. Prover

7.1 Motivation

The module `Data.Fin.Dec` of the standard library of Agda [3] is a toolkit for building deciders of properties of elements of `Fin n`. The library contains many combinators, but for illustration purposes, it is enough to look at one them:

```
all? : {n : ℕ} {P : Fin n → Set}
       → ((i : Fin n) → Dec (P i))
       → Dec ((i : Fin n) → P i)
```

The combinator `all?` takes some decidable predicate `P` on elements of `Fin n` and returns a decision of whether `P` holds for all elements of `Fin n`.

Suppose we want to use `all?` to establish the property from Section 1, namely, commutativity of the operation `_·_ Pauli`. To do so, we can use the previously established fact that there is a bijection `f2p` from `Fin 4` to `Pauli` and decide by using `all?` whether `f2p i₁ · f2p i₂` is equal to `f2p i₂ · f2p i₁` for all `i₁` and `i₂`:

```
commDec : Dec ((i₁ i₂ : Fin 4)
  → f2p i₁ · f2p i₂ ≡ f2p i₂ · f2p i₁)
commDec = all? (λ i₁ →
      all? (λ i₂ →
         (f2p i₁ · f2p i₂) ≡P? (f2p i₂ · f2p i₁)))
```

Then, using the proof `f2p-surj` of `p2f` being a pre-inverse of `f2p`, we can establish the property itself:

```
·-comm : (x₁ x₂ : Pauli) → x₁ · x₂ ≡ x₂ · x₁
·-comm x₁ x₂ with fromSq commDec (p2f x₁) (p2f x₂)
·-comm x₁ x₂
    | p rewrite f2p-surj x₁ | f2p-surj x₂ = p
```

This approach appears to generate much less boilerplate comparing than the direct proof given in Section 1. However, there are two shortcomings that we would like to eliminate:

1. The standard library combinators work with `Fin n`. Therefore, before setting out to prove anything about some finite type, we need to provide a bijection from an initial segment of natural numbers. In Section 3.3, we showed that for listable subsets this is not always possible.
2. The property is then first proved for `Fin n` (`commDec`) and then mapped back to `Pauli` using the conversions `f2p` and `p2f` and the proof `f2p-surj`.

7.2 Definition

We start by defining a combinator `subAll?` which is very similar to `all?` shown above:

```
subAll? : {U : Set}{P : U → Set}
  → ListableSub U P
  → {Q : U → Set}
  → ((x : U) → {P x} → Dec (Q x))
  → Dec ((x : U) → {P x} → Q x)
```

The main difference is that the predicates `P` and `Q` now range over some listable subset instead of `Fin n`. Recall that the elements of `ListableSub U P` are the elements of `U` satisfying the `P`.

The same can be done for the existential quantifier:

```
subAny? : {U : Set}{P : U → Set}
  → ListableSub U P
  → {Q : U → Set}
  → ((x : U) → {P x} → Dec (Q x))
  → Dec (Σ[ x ∈ U ] P x × Q x)
```

If `Q` is a decidable predicate on some subset, then we can find out whether at least one element of that subset satisfies `Q`.

The combinators `subAll?` and `subAny?` are sufficient to decide properties which are in prenex form with the quantifiers ranging over the whole finite subset given by `P`. However, for the convenience of the user we have also added combinators for restricted quantification. These combinators allow narrowing the range of quantification by a further predicate decidable on the subset. We will not discuss them here.

Now we can provide some syntactic sugar for our combinators:

```
syntax subAll? f (λ x → z) = Π x ∈ f , z
```

```
syntax subAny? f (λ x → z) = ∃ x ∈ f , z
```

(Agda will automatically rewrite expressions matching the right hand side into the corresponding terms on the left.)

7.3 Example

Recall that the elements of `ListableSub U P` are the elements of `U` that satisfy `P`. For the special case when `U` is a listable set and `P = λ _ → ⊤`, we have simplified versions of `subAll?` and `subAny`, eliminating the overhead of dealing with trivial proofs of `P x` when `P = λ _ → ⊤`.

The proof of commutativity of the operation `_·_` on `Pauli` amounts essentially to just restating the property:

```
·-comm : (x₁ x₂ : Pauli) → x₁ · x₂ ≡ x₂ · x₁
·-comm = fromSq (
   Π x₁ ∈ listablePauli ,
   Π x₂ ∈ listablePauli , x₁ · x₂  ≡P? x₂ · x₁ )
```

The proof that the group operation has a left unit is similar:

```
·-id : Σ[ x ∈ Pauli ] (y : Pauli) → x · y ≡ y
·-id = fromSq (
   ∃ x ∈ listablePauli ,
   Π y ∈ listablePauli , x · y  ≡P? y )
```

8. Related Work

Intuitionistic frameworks give rise to a rich variety of notions of finiteness that collapse classically. In [8], [5] and [14], the author describe various concepts of finiteness and their interrrelation. According to their classification, this paper focuses on the strongest notion of finiteness, namely, finitely enumerable (listable) sets.

Since finite sets are essential for many formal theories, the users of proof assistants are asking for ways to define new finite sets [1, 7] and the developers are implementing libraries.

The Agda standard library [3] contains a toolkit for building deciders of properties of `Fin n`. Before using it for proving the property of a finite set `X`, the user needs to provide a bijection from an initial segment of natural numbers. After establishing the property for `Fin n`, it can be lifted to the original set `X`.

In [9], Gélineau improves the approach of the standard library by implementing an elegant library in Agda for proving properties quantified over finite sets. The user of a library is only asked to prove finiteness of of the set of interest by specifying a bijection from `Fin n` and then the property can be checked without transporting it to and from `Fin n` manually.

In [15], Spiwack implements a Coq library for finite subsets of countable sets. Countable sets are sets equipped with a surjection from $\mathbb{N}$. Countable sets have decidable equality: it is sufficient to test for equality the natural numbers corresponding to the elements. Finite sets then can be specified by providing a list of elements of the countable base set without duplicates. The library has support for proving decidable propositions and has a syntax for defining sets by comprehension.

In Ssreflect [10], a finite type is a type together with an explicit enumeration of its elements. Finite types can be constructed from finite duplicate-free sequences. Finite types come with boolean quantifiers `forallb` and `existsb` taking boolean predicates and returning booleans. If `X` is a finite type, the type `{set X}` is the type of sets over `X` , which is itself a finite type. Ssreflect provides the usual set theoretic-operations including membership and set comprehensions.

The authors of [4] show a systematic way for building combinators for finite sets declaratively and provide lemmas that encapsulate commonly used reasoning steps. Their work is implemented on top of Ssreflect.

9. Conclusions

In this work we addressed the problem of programming with finite sets in the dependently typed setting of the Agda programming language.

We showed that the direct approach of defining listable types as datatypes with nullary constructors is verbose and introduces brittle interdependencies between different definitions that are tedious to maintain.

Afterwards, we introduced different variations of the notion of a listable set. We proved that giving a complete list of elements of a set is equivalent to providing a surjection from an initial segment of natural numbers. Also, giving a complete list of elements without duplicates is equivalent to providing a surjection. Moreover, all four definitions are equivalent, the reason being that equality on a listable set is decidable.

Next, we introduced a more general notion of a listable subset (where a subset is specified by a base set and a predicate, but not necessarily explicitly constructed as a set). We showed that, in general, listability of a subset does not imply decidability of equality on its elements. We also proved that the union, intersection, product, and disjoint sum of listable subsets are listable subsets.

Then we proposed a pragmatic way of specifying a finite set as a subset of an already constructed set with decidable equality. A specification in this form defines a set that is listable and as a consequence also has decidable equality.

We developed two approaches for defining functions from listable subsets. In the first approach, we convert a well-formed list of argument–value pairs into a function. This is convenient to use for smaller domains. The second approach uses a list of predicate–function pairs and proofs that the predicates cover the whole domain and there are no unreachable predicates. The user receives feedback from the type-checker about predicates that are not reached and elements of the domain that are not matched.

Finally, we implemented combinators for proving propositions quantified over listable subsets. The unusual aspect is that they can be used even for subsets without decidable equality.

Acknowledgement The first author thanks Anna and Albert for their support and patience.

This research was supported by the ERDF funded Estonian CoE project EXCS, the Estonian Ministry of Education and Research institutional research grant no. IUT33-13 and the Estonian Science Foundation grant no. 9475.

References

[1] The Agda Community. Collections/containers/finite sets. *The Agda Mailing List*, 2011. `http://comments.gmane.org/gmane.comp.lang.agda/3326`

[2] The Agda Team. *The Agda Wiki*, 2015. `http://wiki.portal.chalmers.se/agda/`

[3] The Agda Team. The Agda standard library version 0.9, 2014. `http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Libraries.StandardLibrary`.

[4] Y. Bertot, G. Gonthier, S. O. Biha, I. Pasca. Canonical big operators. In O. A. Mohamed, C. A. Muñoz, S. Tahar, eds., *Proc. of 21st Int. Conf. on Theorem Proving in Higher Order Logics, TPHOLs 2008*, v. 5170 of *Lect. Notes in Comput. Sci.*, pp. 86–101. Springer, 2008.

[5] M. Bezem, K. Nakata, T. Uustalu. On streams that are finitely red. *Log. Methods in Comput. Sci.*, v. 8, n. 4, article 4, 2012.

[6] E. Bishop, D. Bridges. *Constructive Analysis*. V. 279 of *Grundlehren der mathematischen Wissenschaften*. Springer, 1985.

[7] The Coq Community. Finite sets in proofs. *The Coq Mailing List*, 2010. `http://comments.gmane.org/gmane.science.mathematics.logic.coq.club/4682`.

[8] T. Coquand, A. Spiwack. Constructively finite? In L. Laureano Lambán, A. Romero, J. Rubio, eds., *Contribuciones científicas en honor de Mirian Andrés Gómez*, pp. 217–230. Universidad de La Rioja, 2010.

[9] S. Gélineau. Library for proving propositions quantified over finite sets, 2011. `https://github.com/agda/agda-finite-prover`

[10] G. Gonthier, A. Mahboubi, E. Tassi. A small scale reflection extension for the Coq system. Rapport de recherche RR-6455. INRIA, 2008. `http://hal.inria.fr/inria-00258384`.

[11] K. Kuratowski. Sur la notion d'ensemble fini. *Fund. Math.*, v. 1, pp. 129–131, 1920.

[12] M. Nielsen, I. Chuang *Quantum Computation and Quantum Information*. Cambridge Univ. Press, 2000.

[13] U. Norell. 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*, v. 5832 of *Lect. Notes in Comput. Sci.*, pp. 230-266. Springer, 2009.

[14] E. Parmann. Some varieties of constructive finiteness. In *Abstracts of 19th Int. Conf. on Types for Proofs and Programs*, pp. 67–69. 2014.

[15] A. Spiwack. A Coq library for extensional finite sets and comprehension, 2014. `https://github.com/aspiwack/finset`

[16] P. Wadler. Comprehending monads. *Math. Struct. in Comput. Sci.*, v. 2, n. 4, pp. 461-493, 1992.