SAARLAND UNIVERSITY
FACULTY OF NATURAL SCIENCES AND TECHNOLOGY I

BACHELOR'STHESIS

A COQ LIBRARY FOR FINITE TYPES

Author:
Jan Christian Menz

Advisor:
Prof. Dr. Gert Smolka

Reviewers:
Prof. Dr. Gert Smolka
Prof. Dr. Holger Hermanns

Submitted: 27th July 2016

Saarbrücken, 27th July, 2016

Abstract

For my Bachelor's thesis I develop a library for finite types in the proof assistant Coq. This includes the definition of basic finite types, features like decidability, cardinality, a constructive choice function, and type operations like option, (dependent) pairing, sum, taking subtypes and the conversion from lists to finite types. The library defines vectors as a way to represent extensional functions with a finite domain and implements an iterative algorithm to obtain subsets of finite types.

Canonical structures, coercions and type classes are used to minimise the notational burden for the user. The library is tested with a small formalisation of finite automata including closure and decidability properties and the conversion between deterministic and nondeterministic automata.

The pre-existing Ssreflect library for finite types serves as an orientation. The goal is to achieve a compact and understandable development in pure Coq.

Acknowledgments

This bachelor thesis would not have been possible without the support of a great number of people.

Firstly I want to thank my advisor Prof. Smolka. He spent numerous hours in meetings with me discussing my bachelor thesis, pointing out if I had gone of the wrong track and suggesting alternatives. I also owe a lot of thanks to my family and friends for supporting me all the way. Without them I would probably not even have started this bachelor thesis. I am especially obliged to Fabian Kunze, Kathrin Stark, Moritz Lichter and Yannick Forster. The first two helped me to get Coq, Proof General and makefiles to work, while the last three proofread earlier drafts of this thesis. I also want to thank Christian Doczkal for his explanations of both Ssreflect and his and Prof. Smolka's formalisation of finite automata in [3]. Another big thanks is due to Felix Freiberger, Yannick Forster and all the other unknown TeXperts who are responsible for the LaTeX definitions I was able to use.

Finally I want to thank Prof. Smolka and Prof. Hermanns for reviewing this bachelor thesis.

Contents

Chapter 1

Introduction

Finite types, that is types with finitely many inhabitants, are ubiquitous in type theory. They have many properties that are intuitive not only to professionals but also to people who usually do not have anything to do with mathematics. Some of these properties are so fundamental that they would not even be mentioned in an informal proof. However, unthinking use of "obvious" theorems in a proof assistant like Coq is not possible. No matter how "obvious" something might seem to a human, a computer always wants to see a proof.

Considering the prevalence of finite types in some developments, with types like $\mathbb{B}$ and *unit* just being the most basic examples, this can be a lot of work. However, we cannot do without taking finiteness into consideration. One can even argue that the finiteness of a type is more important information than finiteness of a set in classical logic. Finite types have important classical properties that are not provable constructively for types in general.

1.1 Related work

The importance of finite types in type theoretic developments is not a new realisation. People have looked into finite types in type theory before. The mathematical components team accompany their Ssreflect proof script language with the Ssreflect library [8, 17], which includes a library for finite types. This library has been a big inspiration for this Bachelor thesis.

Denis Firsov and Tarmo Uustalu [5] investigate different possibilities to represent finite sets in Agda, especially different notions of listability and discuss their respective advantages and disadvantages. Unlike the Ssreflect library and this Bachelor thesis they also consider non discrete finite sets.

Gonthier and his coauthors explain a simplified version of the Ssreflect implemenation of finite types in their paper about finite group theory [9]. This explanation has been very influential on the initial design of the library. In his doctoral thesis [6] Garillot explains the unsimplified architecture of the Ssreflect library very well.

1.2 Contributions

This bachelor thesis therefore does not embark to discover something new but rather to understand the old. It will investigate how to built a library for finite types in Coq. Unlike in the Ssreflect library [17] serving as a reference point, which uses its own proof script language, the accompanying development will be carried out in pure Coq. The focus will be more on understandability than on high performance. Nevertheless the aim is to arrive at a library which is usable in practice without much of a hassle. In particular this bachelor thesis presents

- A beneficial combination of canonical structures and type classes enabling automatic inference of structures.
- A formalisation of finite types based on the presentation of Gonthier et. al. [9] using canonical structures and type classes.
- Decidability properties of finite types and their consequences including a constructive choice function.
- Constructions of finite types for cartesian products, sums, options, dependent pairs and subtypes.
- The construction of a finite type from a list over a discrete type.
- A definition of vectors as representations of extensional functions with a finite domain, the conversion between vectors and functions and the construction of a finite type for vectors with a finite codomain.
- Properties of cardinality of finite types, especially proofs of the well known pigeon hole principles for injective, surjective and bijective functions between finite types; explicit formulas for the cardinality of most of the finite types obtain by the presented constructions and the relationship between cardinality of lists and cardinality of finite types.
- An adaptation of finite closure iteration [14, 13] to finite types. This algorithm is used to iteratively compute a list over a finite type by just providing a predicate determining which inhabitants can be added to a given list. The presentation includes a condition on which the algorithm computes a smallest fixed point of the iterated function.
- A small formalisation of finite automata used to test the library. This includes closure properties of regular languages, decidability properties, and the conversion between deterministic and nondeterministic finite automata.

All presented lemmas, theorems, facts and corollaries have been proven in the accompanying Coq development.

1.3 Overview

- The 1^{st} chapter contains the introduction.
- The 2^{nd} chapter introduces the preliminaries for the construction of finite types namely decidability, discreteness and both type classes and canonical structures.
- The 3^{rd} chapter contains the definition of finite types, discusses the equivalence between properties of lists and properties of finite types, and gives a detailed explanation of finite type registration in Coq.
- In the 4^{th} chapter finite types for cartesian product, option and sum types are constructed from their finite constituent types.
- The 5^{th} chapter constructs finite types for both general dependent pairs and dependent pairs with a proof as their second component (subtypes). It also includes the construction of a finite type from a list over a discrete type.

- The 6th chapter introduces vectors indexed by a finite type. It contains the construction of a finite type for vectors over a finite type and shows that vectors can be used to represent extensional functions with a finite domain.
- Chapter 7 contains proofs of the well known pigeon hole principles for finite types. It also discusses the relationship between the cardinality of lists and the cardinality of finite types.
- The 8th chapter adapts finite closure iteration to finite types.
- Chapter 9 tests the library with a formalisation of finite automata. It contains the definition of deterministic and nondeterministic finite automata including a proof of their equivalence, proofs of closure properties and the use of different notions of reachability to obtain non trivial decidability properties for deterministic finite automata.
- Chapter 10 summarises the findings. It also discusses possible alternative definitions of finite types and points out important differences between the approach in this bachelor thesis and the approach taken by the Ssreflect [17] team.

Chapter 2

Discrete Types

The formalisation of finite types is based on discrete types.To understand the formalisation of finite types one therefore needs to understand the formalisation of discrete types we will be working with.

2.1 Decidability

This formalisation is based on the notion of decidability as introduced in [14, 15, 13]. Decidability is related to the law of *excluded middle* which says that any proposition either holds or its negation holds. In classical logic excluded middle is a well known law. In type theory excluded middle does not hold in general. Since the *elim-restriction* often prevents case analysis on propositions if the goal is no propostion, there also is a strong version of *excluded middle* which is called *decidability*.

Definition 2.1.1 (Decidability). [14, 13, 15]

```
dec (P:ℙ) := {P} + {¬ P}.
```

A proposition P being decidable means that one can write a program which computes either a proof of P or a proof of $\neg P$. Of course any proposition is decidable if there is an equivalent decidable proposition. The notion of decidability is naturally extended to predicates. A predicate is decidable if it returns a decidable proposition when applied completely.

Fact 2.1.1. *[13]*
The usual logical connectives $\wedge, \vee, \neg, \rightarrow, \leftrightarrow$ are decidable on decidable propositions. Likewise are equality on $\mathbb{N}$ and $\mathbb{B}$.

Fact 2.1.2. *[14, 13]*
For any type X *and list* A *of elements of* X *and decidable predicate* p *of type $X \rightarrow \mathbb{P}$ both $\forall\, x \in A, p\ x$ and $\exists\, x \in A, p\ x$ are decidable because one can just try every element in* A.

2.1.1 Type classes

We will use the implementation of decidability from [14, 13, 15] which uses type classes. A type class is a mechanism for implicit argument inference. One can declare a class of types as a type class.

Definition 2.1.2. [13]

```
Existing Class dec.
```

Afterwards one can declare instances of this class for a specific type[1]:

Example 2.1.1.
```
Instance True_dec :  dec ⊤:= left I.
```

As a result whenever an argument of type *dec* $\top$ is missing, Coq will automatically use *True_dec*. This works for any other decidability type for which an instance of *eq_dec* was declared as an instance, e.g. *dec* $\bot$, *dec* $(1 = 2 \vee \bot)$. In general a declared instance of type X is used if an argument of type X is missing. This does, however, mean that it does not make sense to define several instances for the same type. Instances can also be parametrised.

Example 2.1.2. *[13]*
```
Instance and_dec (P Q : ℙ) :  dec P → dec Q → dec (P ∧ Q).
Proof.
unfold dec; tauto.
Qed.
```

Coq automatically uses several instances to derive an argument. For example an argument of type dec $(1 = 2 \vee \bot)$ can be inferred with the help of instances for disjunction, $\bot$ and equality on natural numbers.

With clever use of implicit arguments and type class inference we can obtain a function which behaves as if it had the type $\forall (P : \mathbb{P}),$ *dec* P.

Definition 2.1.3. [13, 14]
```
decision (P:ℙ) {d:  dec P} := d.
```

The argument *d* is implicit and will therefore be automatically inferred if there is an instance for *dec* P.

2.1.2 De Morgan's laws

De Morgan's laws are a set of well known laws of classical logic concerned with the duality of certain operators[4]:

Remark 2.1.1 (de Morgan's laws).

- $\neg(P \wedge Q) \leftrightarrow \neg P \vee \neg Q$
- $\neg(P \vee Q) \leftrightarrow \neg P \wedge \neg Q$
- $\neg\,(P \rightarrow Q) \leftrightarrow P \wedge \neg\, Q$
- $(\forall x : p\ x) \leftrightarrow \neg\, \exists x\ (\neg\, p\ x)$
- $(\exists x : p\ x) \leftrightarrow \neg\, \forall x\ (\neg\, p\ x)$
- $\neg\,(\exists x : p\ x) \leftrightarrow \forall x : \neg\, p\ x$
- $\neg\,(\forall x : p\ x) \leftrightarrow \exists x : \neg\, p\ x$

Since some proofs require case analysis on the truth value of propositions, not all of these laws hold in a constructive setting. Assuming excluded middle they are of course provable once again. This also means that they hold for decidable propositions. Let's take the time to investigate were classical reasoning is needed in the proof of de Morgan's laws.

Remark 2.1.2.

- $\neg(P \wedge Q) \leftrightarrow \neg P \vee \neg Q$
  - $\rightarrow$ Needs decidability of P or decidability of Q because we need to know whether to prove $\neg P$ or $\neg Q$
  - $\leftarrow$ holds constructively
- $\neg(P \vee Q) \leftrightarrow \neg P \wedge \neg Q$
  - $\rightarrow$ holds constructively
  - $\leftarrow$ holds constructively
- $\neg\ (P \rightarrow Q) \leftrightarrow P \wedge \neg\ Q$
  - $\rightarrow$ needs decidability of P to prove P or arrive at a contradiction. $\neg\ Q$ can be proven constructively
  - $\leftarrow$ holds constructively
- $(\forall x : p\ x) \leftrightarrow \neg\ \exists x\ (\neg\ p\ x)$
  - $\rightarrow$ holds constructively
  - $\leftarrow$ needs p to be a decidable predicate to either prove $p\ x$ or to arrive at a contradiction
- $(\exists x : p\ x) \leftrightarrow \neg\ \forall x\ (\neg\ p\ x)$
  - $\rightarrow$ holds constructively
  - $\leftarrow$ needs $\exists x, p\ x$ to be decidable to either prove $\exists x, p\ x$ or to arrive at a contradiction
- $\neg\ (\exists x : p\ x) \leftrightarrow \forall x : \neg\ p\ x$
  - $\rightarrow$ holds constructively
  - $\leftarrow$ holds constructively
- $\neg\ (\forall x : p\ x) \leftrightarrow \exists x : \neg\ p\ x$
  - $\rightarrow$ needs p to be a decidable predicate and decidability of $\exists x, \neg p\ x$. To either prove the theorem or to arrive at a contradiction
  - $\leftarrow$ holds constructively

At least one direction of each law can be proven constructively. There are, however, several laws where case analysis is needed. Two of these require case analysis on existentially quantified formulas. As quantifications over list elements are decidable these laws hold constructively for quantifications over list elements.

2.2 Discrete types

Next we need to discuss *discrete* types.

Definition 2.2.1. [15]
We call a type *discrete* if equality of elements of this type is decidable.

Decidability of equality is needed so often that it is useful to have an abbreviation for it.

Definition 2.2.2. [13, 14]
```
eq_dec (X:Type) := ∀ x  y, dec (x = y).
```

Discrete types are realised as *eqType*s by bundling a type and a proof of decidability together.

Definition 2.2.3. [13]
```
Structure eqType[1] := EqType {
eqtype :> Type;
decide_eq :  eq_dec eqtype }.
```

We can now define decidability instances for discrete types:

Example 2.2.1. *[13]*
```
Instance bool_eq_dec :  eq_dec 𝔹.
Proof. ... Qed.
```

Example 2.2.2.
```
EqBool := EqType 𝔹.
```

Note that the second argument of *EqType* is automatically inferred.

2.2.1 Usability

Ideally, the usage of discrete types would resemble the usage of plain types as closely as possible but still provide decidability. All in all there are three important features we would like to have:

1. We would like to be able to type things with values of type *eqType*.
2. We would like implicit argument inference to infer arguments of type *eqType*.
3. We would like to infer the discrete type belonging to a regular type.

Coercions

Coercions are a Coq-mechanism which allows us to achieve goal one. Coercions are functions automatically applied by Coq if some statement does not type-check. More specifically: If some expression e expects an argument x of type X but instead receives an argument y of type Y, Coq normally rejects the term. However, if f is a coercion from Y to X Coq automatically applies f to y and accepts the term. In the special case of discrete types we can define *eqtype* as a coercion from *eqType* to *Type*. Normally, this is done using the *coercion* keyword but in structures one can use :> instead of *:* to define a projection as a coercion. Without the coercion *eqTypes* would be relatively useless. Coercions can be chained, which we will exploit when formalising finite types.

The use of coercions means that in Coq we can prove that a plain type is equal to its discrete type. So for example 𝔹= *eqBool*. Similar equations hold for any other discrete type.

Type class inference

Our second goal was to enable the inference of implicit arguments of type *eqType*. We have already seen that type classes can infer implicit arguments. However, there is a catch. Consider a function *count* counting the number of occurrences of an element in a list.

[1]For an explanation of what the Structure keyword means see [12]

Definition 2.2.4.

$$\begin{aligned}
&\mathit{count}\ []\ x && = 0 && \\
&\mathit{count}\ (x :: A)\ x && = 1 + \mathit{count}\ A\ x && \\
&\mathit{count}\ (y :: A)\ x && = \mathit{count}\ A\ x && \text{if } x \neq y
\end{aligned}$$

Example 2.2.3.
The count *2.2.4 function works only for* eqTypes *because we have to decide equality. Because we have declared* EqBool *2.2.2 as an* eqType *for* $\mathbb{B}$*, we would hope to be able to state the following:*

∀ x, count [true; false] x = 1.

Unfortunately this does not type check in Coq because Coq cannot infer the implicit argument of type eqType *which should be* EqBool*.*

The reason for this is that, while *EqBool* incorporates a declared instance of a type class, *EqBool* itself is not an instance of a type class and can therefore not be inferred by type class inference. The type *eqType* itself is not even a type class. We either have to insert the implicit argument explicitly every time or need to change *count* to be of type $\forall (X : \mathit{Type}), \mathit{eq\_dec}\ X \to \mathit{list}\ X \to X \to \mathbb{N}$. Therefore we cannot use our definition of *eqType*. We also gained an additional argument.

But even if we keep our definition of *eqType*, we still benefit from type class inference. Using type classes we can compute a discrete type from a base type. The trick is similar to the definition of *decision* 2.1.3. For some type X we use an implicit argument of type *eq_dec X*, which is automatically inferred using type class inference, to build an *eqType*.

Definition 2.2.5.
```
toeqType (X:Type) {D:eq_dec X} :  eqType := EqType X.
```

Canonical Structures

So far, except for the definition of *toeqType*, we have seen *eqTypes* as they are used in [13, 14] or [15]. As we have seen there are still problems with this definition.

There is another inference mechanism in Coq called *canonical structures* which we will investigate to find out if they could help us evade the problems we encountered with type classes.

Unlike type classes, canonical structures do not only take the type into consideration but extend Coq's unification algorithm. Thus one can declare more than one canonical structure for each type. Therefore we can separately declare canonical structures for different base types without having to give up on one single type *eqType* as for type classes. How exactly the unification works is described very well in [12]. For our purpose it is sufficient to get acquainted with canonical structures looking at an example. To use canonical structures we do not have to change the definition of *eqType*, but only need to declare the values we wish to infer as canonical. For example:

Example 2.2.4.
Canonical Structure EqBool.

This adds new rules to Coq's unification algorithm. Now whenever Coq needs to unify $\mathbb{B}$ with the *eqtype* projection from some *eqType* it knows that it must use *EqBool*. In theory this works with projections of a structure which do not return a type, too [12, 6]. In practice we will, however, almost never encounter these cases. The example from earlier type-checks now.

Example 2.2.5.
∀ x, count [true; false] x = 1.

As noted in [6, 12] coercions and *canonical structures* complement each other. In particular *canonical structures* allow to infer the original structure from a value obtained by a coercion.

Canonical structures can be parametrised. For example we can build an *eqType* for the cartesian product of two *eqType*s and declare it as a canonical structure if we have already registered an instance of *eq_dec* for products.

Definition 2.2.6.
`Canonical Structure EqProd` $(T_1\ T_2$`: eqType) := EqType` $(T_1 \times T_2)$

Example 2.2.6.
The expression count [(true,false); (false,true)] (true,true) *type checks.*

In this example we can see how several *canonical structures* work together. First *EqProd* provides the desired *eqType* and the arguments of *EqProd* are inferred with the help of *EqBool*. This nested structures called telescopes [7] can trigger powerful inference chains.

All in all we can see that *canonical structures* allow for inference of implicit arguments of type *eqType* in the way we are used to from regular *Type* arguments.

Synergy of type classes and canonical structures

Together *type classes* and *canonical structures* form a powerful combination. *Canonical structures* allow for inference of implicit arguments while *type classes* allow us to convert types into discrete types. Together they even achieve results that would not have been possible separately:

Example 2.2.7.
Assume we have not only declared EqBool *but also* EqUnit *as the* eqType *for unit. We can now declare* *`EQType_BoolUnit := toeqType (𝔹 × unit).`*

Now let us have a look what the right side looks like, when all implicit arguments are inserted: *`toeqType (𝔹 × unit) (decide_eq (EqCross EqBool EqUnit))`*

$\underline{\text{inferred with canonical structures}}$ $\quad$ inferred with type classes

The implicit argument decide_eq *has been inferred by type class inference. The implicits arguments of* decide_eq *(*EqCross, EqBool, EqUnit*), however, have been inferred with the help of* canonical structures*. Our current definitions would not have allowed this inference without either* type classes *or* canonical structures*.*

Canonical instances and toeqType

Since *toeqType* produces an *eqType* and *eqType* was defined with a coercion to *Type*, we have $X = toeqType\ X$ for any discrete type X in Coq. What is much more interesting is that even without coercions *toeqType* gives us important equalities. For example we have without coercion:

Example 2.2.8.
toeqType 𝔹 = *EqBool*

This is important because it means that *toeqType* produces the *eqType* we have declared as a canonical structure. This property holds for every *eqType* in the library. This is not a trivial equality. Had we defined the canonical structures using a different decider than the one declared as an instance of the type class, it would not hold.

Because *toeqType* normalises discrete types it is idempotent.

Fact 2.2.1.
toeqType *is idempotent, i.e. for any discrete type* X
toeqType (toeqType X) = toeqType X.

All these equations are definitional equalities and can therefore be proven in Coq by reflexivity.

2.2.2 Basic Closure properties

Discrete types are closed under several useful type operations.

Fact 2.2.2.
If X *and* Y *are discrete types, so is* $X \times Y$.

Fact 2.2.3.
If X *is a discrete type, so is* option X.

Fact 2.2.4.
A sum type X + Y *is discrete if both* X *and* Y *are discrete types.*

Chapter 3

Finite Types

The heart of any library for finite types needs to be the definition of finite types. While there are many different ways to ensure finiteness in Coq (several of which are explored in [5] e.g. a bijection to some canonical type with n elements), we will base our definition on listability, i.e. for a finite type we can give a list containing all its inhabitants. This approach has several advantages:

1. We have explicit knowledge about the inhabitants.
2. This approach is very intuitive.
3. One gets many properties of lists which can be transferred to finite types with relative ease. This is exploited in 3.3 and 8.

This notion of finiteness is also the one used in the Ssreflect library [17]. As in Ssreflect [17] we will only allow duplicate free lists, which for example simplifies the definition of *Cardinality* 3.2.5. We also require our types to be discrete. Discreteness is no additional restriction because showing duplicate freeness of a list requires discreteness anyway. We could of course do without it and, depending on the definition, even prove it from the definition of a finite type, but explicitly requiring it makes many proofs much easier and more understandable. Also it makes some other things very obvious:

Remark 3.0.1.

- Most propositions cannot become finite types because of the elim restriction.
- Functions are no finite types unless one assumes functional extensionality.

As we will see in chapter 6 we can partially circumvent the second consequence.

3.1 Count

To create a finite type in Coq we will have to prove that the list of elements we want to use is indeed complete and duplicate free. To formulate this requirement in a nice way we use the *count* function described in 2.2.4. This is the method described in [9] and also used in the Ssreflect library. There are several important properties of *count* we will regularly exploit:

Fact 3.1.1.
Let X *be a discrete type,* x *be of type* X *and* A *be a list containing elements of type* X.
Then $x \notin A \leftrightarrow count\ A\ x = 0$.

Fact 3.1.2.
Let X *be a discrete type,* x *be of type* X *and* A *be a list containing elements of type* X.
Then $x \in A \leftrightarrow$ *count* $A\ x > 0$.

Both of these can be proved by induction on A. Most importantly:

Lemma 3.1.3.
Let X *be a discrete type,* x *be of type* X *and* A *be a list containing elements of type* X. *Then* $dupfree$[1] $A \rightarrow x \in A \rightarrow count\ A\ x = 1$

Proof. By induction on the derivation of dupfree A.

nil:
In this case $x \in []$ is a contradiction.

dupfree A′ and $\mathbf{y} \notin \mathbf{A'}$:
Under the assumption $x \in y :: A'$ we have to show count ($y :: A'$) x = 1.

$\mathbf{x = y}$:
In this case it remains to show that count A' x = 0. x=y and $y \notin A'$ hence $x \notin A'$. By Fact 3.1.1 count A' x = 0.

$\mathbf{x \neq y}$:
count A' *x = 1* remains to be shown which we conclude with the induction hypothesis.

□

Similarly we have:

Lemma 3.1.4.
Let X *be a discrete type and* A *be a list over type* X. *Then*
$\forall$ x, count A x $\leq 1 \rightarrow$ dupfree A.

Proof. By induction on *A*.

nil:
In this case we have to prove *dupfree nil* which is true without assumptions.

x::A′
We have to show that $x \notin A'$ and that A' is duplicate free. We know that *count (x::A′) x* $\leq$ *1*, this means that *count A′ x = 0*. By Fact 3.1.1 we conclude that indeed $x \notin A'$.
We still have to prove *dupfree A′*. Now we can use the induction hypothesis and have to show that $\forall$ *y, count A′ y* $\leq$ *1*. By assumption we know that this is true for *x::A′*. We do a case analysis on $y = x$. Either *y=x*, in this case we have already shown that *count A′ x = 0* ≤ 1, or $y \neq x$, and we have *count A′ y* $\leq$ *1* by assumption.

□

Lemma 3.1.5.
Let X:eqType x:X *and* A B: list X. *Then count (A ++ B) x = count A x + count B x.*

The proof here is by induction on *A*.

[1]For the definition of *dupfree* see [14]

3.2 Definition of finite types

We define finite types using the same approach as with discrete types. This means we define a type class which contains all information about the finite type.

Definition 3.2.1.
```
Class finTypeC (type:  eqType):  Type:= FinTypeC {
enum:  list type;
enum_ok:  ∀ x:  type, count enum x = 1 }.
```

The proof *enum_ok* ensures that the list *enum* is indeed complete and duplicate free.
To get one type *finType* instead of one separate type for each base type we combine the type with *finTypeC*, similar to the definition of *eqType* 2.2.3.

Definition 3.2.2.
```
Structure finType:  Type:= FinType {
type :> eqType;
class :  finTypeC type }.
```

Since *finType*s have a coercion to *eqType*s and *eqType*s have a coercion to types we can use *finType*s both as *eqType*s and as types. Of course this definition has the same disadvantages as the pure definition of *eqType*. Therefore we need to register any finite type we declare as a canonical structure to enable inference.

We can now use the same trick we have used when defining *toeqType* 2.2.5 to build a *finType* from its base type.

Definition 3.2.3.
```
tofinType (X:Type) {D: eq_dec X} {f:  finTypeC _}:  finType :=
FinType (toeqType X).
```

This time both *D* and *f* are automatically inferred implicit arguments. The properties of *toeqType*, i.e. normalisation to canonical forms and idempotents are also properties of *tofinType*.

To get more direct access to the list of elements we declare a new projection:

Definition 3.2.4.
```
elem (F: finType) := enum (type F) (class F).
```

Based on this we can define the cardinality of a finite type.

Definition 3.2.5 (Cardinality)**.**
Let *X* be a finite type. Then *Cardinality X* is defined as $|elem\ X|$[2].

Intuitively the *Cardiniality* of a finite type is the number of its inhabitants. We show that the definition of *enum_ok* does indeed guarantee a complete and duplicate free list.

Fact 3.2.1.
For every finite type X *the list* elem X *is duplicate free.*

Proof. By *enum_ok* we know that ∀ *x, count (elem X) x = 1.* This implies that
∀ *x, count (elem X) x* ≤ *1.* With Lemma 3.1.4 we conclude that *elem X* is duplicate free. □

Fact 3.2.2.
Let X *be a finite type and* x *of type* X*. Then* $x \in (elem\ X)$*.*

[2]We write |A| for the length of a list A[14]

Proof. By 3.1.2 it suffices to show that *count (elem X)* $x > 0$. By *enum_ok* we get *count (elem X)* $x = 1 > 0$. □

Remark 3.2.1.
By the definition of list inclusion[3] this implies that for every list *A: list X* $A \subseteq$ *elem X*.

3.3 Conversion to lists

Sometimes it is helpful to prove properties about finite types by proving them for the list of elements. Fortunately there are straightforward conversions which allow to do exactly that.

Fact 3.3.1.
Let X *be a finite type and* p *be a predicate over* X. *Then* $(\forall x, p\ x) \leftrightarrow \forall x, x \in (elem\ X) \to p\ x.$

Proof.
$\to$: Let *x* be of type *X*. Since $\forall x, p\ x$ also $p\ x$.
$\leftarrow$: Let *x:X*. By assumption it suffices to show that $x \in (elemX)$ which is true by Fact 3.2.2. □

Fact 3.3.2.
Let X *be a finite type and* p *be a predicate over* X. *Then* $(\exists x, p\ x) \leftrightarrow \exists x, x \in (elem\ X) \wedge p\ x.$

Proof.
$\to$: Let *x:X* such that *p x*. Then obviously *p x*. By Fact 3.2.2 also $x \in (elem\ X)$.
$\leftarrow$: Let *x:X* such that *p x* and $x \in$ *elem X*. Then x is an x such that *p x*. □

The next equivalence might look a bit strange since it replaces the $\wedge$ with $\to$. This works because $x \in (elem\ X)$ is a tautology if *X* us a finite type. This version is useful because one gets $x \in (elem\ X)$ as an additional assumption instead of as an additional proof obligation if one wants to prove $\exists x, p\ x$ this way.

Fact 3.3.3.
Let X *be a finite type and* p *be a predicate over* X. *Then* $(\exists x, p\ x) \leftrightarrow \exists x, x \in (elem\ X) \to p\ x.$

Proof.
$\to$: Let *x:X* such that *p x*. Then also *p x*.
$\leftarrow$: Let *x:X* such that $x \in (elem\ X) \to p\ x$. We get $x \in (elem\ X)$ by Fact 3.2.2. Therefore *x* is an *x* such that *p x*. □

Most importantly we this means we can decide quantified formulas over decidable predicates over finite types [5].

Fact 3.3.4 (Decidability of universally quantified formulas)**.**
Let X *be a finite type and* p *a predicate over* X. *Then* (∀ x, dec (p x)) → dec (∀ x, p x).

Proof. By the equivalence property 3.3.1 it suffices to show that *(∀ x, dec (p x)) → dec (∀ x,* $x \in (elem\ X) \to p\ x$*)* which is a well known property of lists 2.1.2. □

Fact 3.3.5 (Decidability of existentially quantified formulas)**.**
Let X *be a finite type and* p *a predicate over* X.

Proof. By the equivalence property above 3.3.2 it suffices to show that *(∃ x, dec (p x)) → dec (∃ x,* $x \in (elem\ X) \wedge p\ x$*)* which is a well known property of lists 2.1.2. □

[3]See [14]

This also implies that the laws of de Morgan's 2.1.2 which need decidability of existential quantifications hold for quantified formulas over decidable predicates over finite types. We can also define a constructive choice function, which converts an existential quantifier into a sigma type[4], allowing us to circumvent the elim-restriction.

Theorem 3.3.6 (Constructive choice)**.**
Let X *be a finite type and* p *be a decidable predicate over* X*. Then there is a constructive choice function with the type* ∃ x, p x → {x | p x}.

Proof. The definition is surprisingly easy. Using the equivalence property above 3.3.2 we can change $\exists x, p\ x$ to $\exists x, x \in\ (elem\ X) \wedge p\ x$. There already is a constructive choice function for lists [14, 13] which we use to obtain the desired result. □

3.4 Finite type creation guide

Having defined *finType* we need to populate the library with frequently used finite types. The steps necessary to declare a type as a *finType* are rather simple and dictated by the design explained in chapters 2 and 3. To illustrate the process we use the example of $\mathbb{B}$.

3.4.1 Step 1: Registering an instance for decidability

The first step to declare a *finType* must always be to show that equality on said type is decidable and to declare the proof as an instance of the *dec* type class.

Example 3.4.1.
```
Instance bool_eq_dec :  eq_dec 𝔹.
```

3.4.2 Step 2: Registering an *eqType*

The second step is to declare the type as a discrete type, i.e. as an *eqType*. Since we already have declared the proof of discreteness as an instance, it can be inferred automatically. In order to enable implicit argument inference of the *eqType* we declare it as a *canonical structure*.

Example 3.4.2.
```
Canonical Structure EqBool := EqType 𝔹.
```

3.4.3 Step 3: Completeness of list

Now we have to prove that the list containing all elements of the type is indeed complete and duplicate free.

For more complicated types it can be helpful to write a function computing the list of elements first. For simple types like $\mathbb{B}$ this is not necessary.

Example 3.4.3.
```
Lemma bool_enum_ok x:  count [true; false] x = 1.
```

[4]see definition of *sig* in [18]

3.4.4 Step 4: Registering an instance of *finTypeC*

Next we have to register an instance of finTypeC. Since the list of elements *enum* can be inferred from the proof of correctness from step 3 this step looks the same way every time.

Example 3.4.4.
```
Instance finTypeC_bool :  finTypeC EqBool.
Proof.
econstructor.  apply bool_enum_ok.
Defined.
```

It is crucial to save the definition using `Defined.` If we accidentally use `Qed.`, the definition will be made *opaque* and we loose access to the list of elements and will not be able to do some proofs.

3.4.5 Step 5: Registering a *finType*

Now we can finally declare the type as a *finType*. Since the instance of *finTypeC* can be inferred automatically the type suffices as an argument.

Example 3.4.5.
```
Canonical Structure finType_bool :  finType := FinType EqBool.
```

At this point it is important *not* to use *tofinType* like this:

Example 3.4.6 (bad idea)**.**
```
Canonical Structure finType_bool_bad :  finType := tofinType 𝔹.
```

Although this declaration works it can lead to problems later on because *type finType_bool* is defined as *toeqType* $\mathbb{B}$. Since Coq only looks at the head symbol (in this case *toeqType*) inference will not work correctly.

Remark 3.4.1. Since steps 2, 4 and 5 are identical for any type, these definitions will mostly be left out in the explanations of constructions of other *finTypes*. Any mentions of names of the from *EqX* and *finType_X* can be assumed to refer to the *eqType* for *X* and *finType* for *X* respectively. For example *finType_Prod* refers to the finite type for product types.

Simple inductive types like $\mathbb{B}$ whose constructors take no arguments are very easy to register as finite types. But they also form the basis for every other type that can be constructed. For these simple types even steps 1 and 3 can normally be copied from the proofs for $\mathbb{B}$. Of course the library supports other simple types, namely $\top$, $\bot$, unit, $\mathbb{B}$ and Empty_set.

Chapter 4

Basic Constructions

Finite types are closed under important type operations. Therefore we can build many interesting types just using very few base types and these operations. In this chapter we will discuss the most important simple operators before we go on to discuss more complicated operations in the following chapters. Since the general scheme for obtaining a finite type has already been explained in section 3.4 these chapters will focus on the most interesting parts of the constructions.

4.1 Cartesian product

4.1.1 Construction

To obtain a finite type for the product of two types we need to construct a list containing all possible pairs. To do this we first construct a function that computes all pairs of elements from two lists.

Definition 4.1.1.
```
Fixpoint prodLists (X Y: Type) (A: list X) (B: list Y) :=
match A with
| nil ⇒ nil
| (x::A′) ⇒ map (λ y ⇒ (x,y)) B ++ prodLists A′ B end.
```

Let X and Y be discrete types with x, x' of type X and y of type Y.

Lemma 4.1.1.
count (map (λ y ⇒ (x,y)) B) (x, y) = count B y.

Lemma 4.1.2.
$x \neq x' \rightarrow$ count (map (λ y ⇒ (x,y)) B) (x′, y) = 0.

Lemma 4.1.3.
count (prodLists A B) (x,y) = count A x ∗ count B y.

Proof. By induction on A.

nil:
This simplifies to $0 = 0$ which is true.

x′::A′:
Using 3.1.5 the left side simplifies to

count (map (λ y ⇒ (x′, y)) B (x,y) + count prodList A B (x,y). Now we can use the induction hypothesis to change *count prodList A B (x,y)* to *count A x ∗ count B y*.
The value on the right side depends on whether $x' = x$ or not.

$\mathbf{x = x'}$:
In this case the right side of the equation simplifies to
*count B y + count A x * count B y*.
So now we have:
*count (map (λ y ⇒ (x′, y)) B (x,y) + count A x ∗ count B y = count B y + count A x * count B y*.
By subtraction of *count A x * count B y* from both sides and substitution of x' we get *count (map (λ y ⇒ (x, y)) B (x,y) = count B y* which follows with 4.1.1.

$\mathbf{x \neq x'}$:
In this case we get
*count (map (λ y ⇒ (x′, y)) B (x,y) + count A x ∗ count B y = count A x * count B y*.
By subtraction of *count A x * count B y* from both sides we obtain
count (map (λ y ⇒ (x′, y)) B (x,y) = 0 which we can prove using 4.1.2.

□

As a result we obtain the following result:

Fact 4.1.4.
Let X *and* Y *be finite types and* z *be of type $X \times Y$. Then* count (prodList (elem X) (elem Y)) z = 1.

Proof. *z* needs to be a pair *(x,y)*. Because of 4.1.3 it suffices to show that *count (elem X) x ∗ count (elem Y) y = 1*. Because *X* and *Y* are finite types, we can use the correctness property *enum_ok* to obtain *count (elem X) x = 1* and *count (elem Y) y = 1*. All that remains to be shown is $1 * 1 = 1$ which is trivial. □

This is all we need to obtain a *finType* for products of finite types.

Theorem 4.1.5.
If X *and* Y *are finite types, then* `X (×) Y := FinType (EqProd X Y)` *is a finite type such that* X × Y = X (×) Y.

Proof. With the coercion *type X (×) Y* is equal to *EqProd X Y*. *EqProd X Y* is equal to $X \times Y$ using the coercion *eqtype*. □

4.1.2 Cardinality

As the name suggest the cardinality of a finite product type is the product of the cardinalities.

Theorem 4.1.6.
Let X *and* Y *be finite types. Then* Cardinality (X (×) Y) = Cardinality X ∗ Cardinality Y.

Proof. This is by definition equivalent to |*prodLists (elem X) (elem Y)*| = | *elem X* | ∗ | *elem Y* | . We do induction on *elem X*.

nil:
In this case *prodList* returns *nil*. Since $0 * |elem\ Y| = 0$ we have to prove $0 = 0$ which is trivial.

x::A:

In this case we have to prove $|map\ (\lambda\, y \Rightarrow\ (x,y))\ (elem\ Y) ++ prodLists\ A\ (elem\ Y)| = |elem\ Y| + |A| * |elem\ Y|$. By the induction hypothesis we already have $|prodLists\ A\ (elem\ Y)| = |A| * |elem\ Y|$. All that remains to be proven is $|map\ (\lambda\, y \Rightarrow\ (x,y))\ (elem\ Y)\ | = |\ elem\ Y\ |$. Since *map* does not change the length of a list, this is true.

□

4.2 Option

Option types are a valuable tool if one wishes to write a function which might not necessarily have a result or if one just needs a type with one more element.

4.2.1 Construction

We construct a function which creates a list of type *option X* given a list of type *X*.

Definition 4.2.1.
```
toOptionList (X: Type) (A: list X) := None ::  map Some A .
```

Lemma 4.2.1.
Let X *be a discrete type and (A: list X). Then* count (toOptionList A) None = 1
and count (toOptionList A) (Some x) = count A x.

Both of these can be proven by induction.

Fact 4.2.2.
Let X *be a finite type and* x *be of type* X*. Then* count (toOptionList (elem X)) z = 1.

Proof. By case analysis on z.

None:

This is exactly the first equation of 4.2.1.

Some x:

By 4.2.1 it suffices to show that *count (elem X) x = 1* which follows by *enum_ok*.

□

Again this suffices to define a *finType* for option types of finite types.

Theorem 4.2.3. *If* X *is a finite type, then* `?X := FinType (EqOption X)` *is a finite type such that* option X = ?X.

Proof. With the coercion *type ?X* is equal to *EqOption X* which with the coercion *eqtype* is equal to *option X*. □

4.2.2 Cardinality

Intuitively an option type is one element larger than the original type because we add *None*. So we have:

Theorem 4.2.4.
Let X *be a finite type. Then* Cardinality (?X) = 1 + Cardinality X.

Proof. Cardinality (?X) = |None :: map Some (elem X) | $= 1 + |$ *map Some (elem X)* $|$.
Again map does not change the length of the list so this equals $1 + |$ *elem X* $|$ which is what we wanted to prove. □

4.3 Sum

Sum types are the *Type* equivalent of disjunctions. They are the best thing we have as a union operator for types.

4.3.1 Construction

As usual we define a function computing a list of type $X + Y$. But this time we use two functions: One for the left type and one for the right type.

Definition 4.3.1.
`toSumList1 {X: Type} (Y: Type) (A: list X): list (X + Y) := map inl A.`

Definition 4.3.2.
`toSumList2 {Y: Type} (X: Type) (A: list Y): list (X + Y) := map inr A.`

There are some obvious properties of these function which can easily be proven by induction on the list argument. For discrete types X and Y the following holds:

Lemma 4.3.1.
count (toSumList1 Y A) (inl x) = count A x.

Lemma 4.3.2.
count (toSumList2 X B) (inr y) = count B y.

Lemma 4.3.3.
count (toSumList1 Y A) (inr y) = 0.

Lemma 4.3.4.
count (toSumList2 X B) (inl x) = 0.

As a result we get:

Fact 4.3.5.
Let X *and* Y *be finite types and* z: X + Y.
Then count (toSumList1 Y (elem X) ++ toSumList2 X (elem Y)) z = 1.

Proof. By 3.1.5 it suffices to show.
count (toSumList1 Y (elem X) z + toSumList2 X (elem Y)) z = 1. We do a case analysis on *z*

inl x:

By 4.3.4 *toSumList2 X (elem Y)) (inl x) = 0.* Therefore we just need to show
count (toSumList1 Y (elem X) (inl x) = 1 which we obtain by 4.3.1 and *enum_ok*.

inr y:

By 4.3.3 *toSumList2 Y (elem X)) (inr y) = 0*. Therefore we just need to show *count (toSumList2 X (elem Y) (inr y) = 1* which we obtain by 4.3.2 and *enum_ok*.

□

Theorem 4.3.6. *If* X *and* Y *are finite types, then* `X (+) Y := FinType (EqSum X Y)` *is a finite type such that* X + Y = X (+) Y.

Proof. With the coercion *type X (+) Y* is equal to *EqSum X Y* which with the coercion *eqtype* is equal to $X + Y$. □

4.3.2 Cardinality

As the name suggests the Cardinality of a finite sum type is the sum of the Cardinality of its constituent types.

Theorem 4.3.7.
Let X *and* Y *be finite types.* Cardinality (X (+) Y) = Cardinality X + Cardinality Y.

Proof. By the definition of *X (+) Y* the left side equals
$|toSumList_1$ *Y (elem X)* ++ $toSumList_2$ *X (elem Y)* $|$. By the definitions of $toSumList_1$ and $toSumList_2$ this is again equal to $|$ *map inl (elem X)* ++ *map inr (elem Y)*$|$. Since map does not change the length of the list this is equal to $|elem\ X| + |elem\ Y|$ which is exactly what we wanted to prove. □

Chapter 5

Dependent Pairs

Dependent pairs or sigma types are pairs where the type of the second component is determined by a function on the first component. Dependent pairs come in two flavours: In the first the function maps to *Type*, in the second to $\mathbb{P}$. While the construction of the finite types for regular pairs, options, and sums is relatively straightforward this does not apply to dependent pairs.

Remark 5.0.1.
We will use π_1 (π_2) for the projection to the first component (the second component) of a dependent pair. In this notation we will not differentiate between the two types of dependent pairs.

5.1 Subtypes

The second variety is useful to define subtypes of elements satisfying a predicate. It seems obvious that subtypes of finite types should be finite as well. Unfortunately equality of proofs in $\mathbb{P}$ is usually not decidable making it hard to define an *eqType*. Fortunately there is a workaround.

5.1.1 Pure predicates

This workaround are *pure predicates*.

Definition 5.1.1. [15]
A predicate $p : X \to \mathbb{P}$ is called *pure* if for every *x:X* there is only one proof of *p x*.

Because pure predicates have just one proof of *p x*, equality of proofs is decidable. This helps us in defining subtypes. Firsov and Uustalu used the same trick for the very same purpose [5].

For every decidable predicate there is an equivalent *pure predicate*. We can obtain it by simply injecting it into $\top$ and $\bot$.

Definition 5.1.2. [15]
```
pure := {X: Type} (p:  X → ℙ) {D:∀ x, dec (p x)} x :=
if decision (p x) then ⊤ else ⊥.
```

Fact 5.1.1. *[15]*
If p *is a decidable predicate, then* pure p *is* pure.

Fact 5.1.2. *[15]*
Let p *be a decidable predicate over* X *and* x:X. *Then p x* $\leftrightarrow$ *pure p x.*

Remark 5.1.1.
Note that Fact 5.1.2 allows us to convert between proofs of *pure* and *impure* versions of the same predicate. This will be very handy.

5.1.2 Subtypes of finite types

Now we can define *suptypes* an idea I came across in [15], although they did not define them generally.

Definition 5.1.3.
Let *X* be a type and *p* be a decidable predicate over *X*. We define *subtype p* as *{x:X | pure p x}*.

Lemma 5.1.3 (Extensionality)**.**
Let p *be some decidable predicate and* x x': subtype p. *Then* $x = x'$ *if and only if* $\pi_1\ x = \pi_1\ x'$.

Proof.

$\rightarrow$:
Trivial.

$\leftarrow$:
The first components of *x* and *x'* are equal by assumption. The second components are proofs of *pure predicates* by Fact 5.1.1 and are therefore equal by the definition of *pure*.

□

Fact 5.1.4.
Subtypes of discrete types are discrete.

Proof. Assume for some decidable predicate *p* we have two pairs p_1 and p_2 of type *subtype p*. They need to be of the form *exist x px* and *exist x' px'* where *px* and *px'* have types *pure p x* and *pure p x'*, respectively. Now we do case analysis on $x = x'$.

$\mathbf{x = x'}$:
Now we can show equality by the extensionality principle above 5.1.3.

$\mathbf{x \neq x'}$:
In this case p_1 and p_2 are unequal. We prove this by contradiction.
Assume *p1 = p2*. In this case also $\pi_1\ p_1 = \pi_1\ p_2$. But this reduces to $x = x'$ which is a contradiction.

□

Now we can define a function which converts a list into a list of the *subtype* we wish to obtain. The idea behind the function is that we test every element of the list for the subtype property. If an element satisfies that property, it is added to the result together with a *pure* version of the proof. If not, the element is cast off. In the definition below *purify* performs the conversion between proofs of *impure* and *pure* versions of a decidable predicate implied by Fact 5.1.2.

Definition 5.1.4.

```
Fixpoint toSubList (X: Type) (A: list X) (p:  X → ℙ) (D:∀ x, dec (p x))
:  list (subtype p) :=
match A with
| nil ⇒ nil
| cons x A′ ⇒ match decision (p x) with
    | left px ⇒ (exist _ x (purify px)) ::  toSubList A′ D
    | right _ ⇒ toSubList A′ D
    end
end.
```

To prove that the list obtained with *toSubList* is indeed complete and duplicate free, it helps to have the following lemma.

Lemma 5.1.5.
Let X *be a discrete type and* p *be a decidable predicate over* X. *Then for every* A: list X *and* x *of type* subtype p *the equation* count (toSubList A p) x = count A (π_1 x) *holds.*

Proof. By induction on A.

nil:
This reduces to $0 = 0$ which is true.

a::A′:
First we do a case analysis on *p a*.

p a:
Let *pa* be the proof of *p a*. Now we have to prove that
count (exist (pure p) a (purify pa) :: toSubList A p) x = count (a :: A) (π_1 x)
We have to consider two cases:

π_1 *x* = *a*:
In this case *x = exist (pure p) a (purify pa)* by extensionality of subtypes 5.1.3. As a result we need to show *1 + count (toSubList A p) x = 1 + count A (π_1 x)* now. By the induction hypothesis we have *count (toSubList A p) x = count A (π_1 x)* which implies the goal.

π_1 x $\neq$ a:
In this case by extensionality of subtypes 5.1.3 we also get
x $\neq$ exist (pure p) a (purify pa). The remaining goal is equal to the induction hypothesis.

¬p a:
In this case π_1 x cannot be equal to *a* because π_2 x is a proof of *pure p x)* which is equivalent to a proof of *p x*. Therefore π_1 $x \neq a$ and all that need to be shown is
count (toSubList A p) x = count A (π_1 x) which is the induction hypothesis.

□

Now we can prove the correctness property for the list of a finite type:

Fact 5.1.6.
Let X *be a* finType *and* p *be a decidable predicate on* X. *Then for all x*
count (toSubList (elem X) p) x = 1.

Proof.
By Lemma 5.1.5 it suffices to show that *count (elem X) (π_1 x) = 1*. This is guaranteed by the correctness property *enum_ok* of *finTypes*. □

Now we have everything that is needed to define finite subtypes of finite types.

Theorem 5.1.7. *If* X *is a finite type and* p *a predicate on* X *, then*
`finType_sub p := FinType (EqSub p)` *is a finite type such that* subtype p = finType_sub p*.*

Proof. With the coercion *type finType_sub p* is equal to *EqSub p* which with the coercion *eqtype* is equal to *subtype p*. □

5.2 Finite types from Lists

Since every list is finite it seems reasonable that one can convert a list A over a discrete type into some finite type representing the elements of the list. Having defined subtypes this is not difficult. We simply have to define a *finType* version of *subtype (λ x ⇒ x ∈ A)*. Unfortunately we cannot simply use the construction for subtypes of finite types from 5.1.2 because the base type of the list is not necessarily a finite type, but for example ℕ.

5.2.1 Removing duplicates

Because the list of elements of a finite type needs to be duplicate free, we have to remove the duplicates in our list in some way. We will use an *undup* function designed for exactly this purpose from [14, 13].

Fact 5.2.1. *[13, 14]*
If A is a list over a discrete type, then $x \in A \leftrightarrow x \in undup\ A$.

Fact 5.2.2. *[13, 14]*
If A is a list over a discrete type, then undup *A is* duplicate free*.*

5.2.2 Construction

We will again use the *toSubList* function to obtain our list of elements. It is important that we only remove duplicate elements after we used *toSubList* to obtain the list of *subtype* elements. When we remove duplicates first, the type of the resulting list will not be *list {x | $x \in A$}* but
list {x | $x \in undup\ A$}.

Fact 5.2.3.
Let A *be a list over some discrete type* X*. Then for all* x
count (undup (toSubList A (λ x ⇒ x ∈ A))) x = 1.

Proof.
By 3.1.3 it suffices to show that *undup (toSubList A (λ x ⇒ x ∈ A))* is *duplicate free* and that
x ∈ undup (toSubList A (λ x ⇒ x ∈ A)). We get the first by Fact 5.2.2. For the second by Fact 5.2.1 it suffices to show that *x ∈ toSubList A (λ x ⇒ x ∈ A)*. This is equivalent to
count (toSubList A (λ x ⇒ x ∈ A)) x > 0 by 3.1.2. Thanks to Lemma 5.1.5 we can get rid of *toSubList* and prove *count A (π_1 x)* > 0 instead. Now with 3.1.2 we can go back to showing that $\pi_1\ x$ is an element of *A*. Since *x* has type *{x | $x \in A$}* the type ensures that this is indeed true. □

We declare the instances of *finTypeC*and *finType* as usual.

Definition 5.2.1.
```
Canonical Structure finType_fromList (X: eqType) (A: list X) :=
FinType (EqSubType (λ x ⇒  x ∈ A)).
```

Finally we want to prove that this definition is correct in the following sense: The list of the the first components of the element list of *finType_fromList* is equivalent to the original list A. To show this we need to know the following fact from the Coq standard library.

Fact 5.2.4. *[18]*
$x \in map\, f\, A \leftrightarrow \exists\, x, x \in A \wedge f\, x = y$

Theorem 5.2.5.
Let X *be a discrete type and* A: list X*. Then* finType_fromList A *is a finite type such that* map π_1 (elem (finType_fromList A) $\equiv$ A.

Proof. We have to show both directions:

$\rightarrow$:

Let $x \in A$. We have to show $x \in$ *map* π_1 *(elem (finType_fromList A)*. By Fact 5.2.4 it suffices to show that there exists a *p: subtype (fun x ⇒ x ∈ A)* such that $\pi_1\ p = x \wedge p \in$ *(elem (finType_fromList A))*. We can construct such a *p* from *x* and the proof of $x \in A$. Obviously $\pi_1\ p = x$.
By the definition of *finType_fromList* we need to show *p* ∈ *undup (toSubList A (fun x ⇒ x ∈ A))*. By Fact 3.1.2 it suffices to show
count (undup (toSubList A (fun x ⇒ x ∈ A))) p > 0. By Fact 5.2.3 the left side is equal to 1 and $1 > 0$.

$\leftarrow$:

Let $x \in$ *map* π_1 *(elem (finType_fromList A)*. We need to show that $x \in A$. By Fact 5.2.4 there exists a *p: subtype (λ x ⇒ x ∈ A)* such that $\pi_1\ p = x$. Because of the type of *p*, $x \in A$.

□

Remark 5.2.1. While we have been able to convert lists to finite types this is a construction which is not very useful in practice. Nearly all important properties of finite types are also properties of lists over a discrete type. In fact we have shown many properties for finite types using properties of lists. This means that the conversion to a type mainly adds a clunky package around the list without adding many advantages. This packaging makes access to the list elements more complicated than before. The only advantage of this packaging is that one can use the list as a type. The situation in Ssreflect [17] is a bit different. Since Ssreflect is not only a library for finite types it also provides support for finite sets. This includes set operations like incusion on subtypes. Since support for list inclusion and similar operation already exists in [13] which we use as the foundation of our library, the existence of these set operations would not really be a good reason to convert lists into types in our case, either.

5.3 General sigma types

Surprisingly the definition of finite types for general dependent pairs[1] is much more complicated than the definition of subtypes although this time equality on the second component can be assumed to be decidable.

[1]For the definition look up *sigT* in the Coq standard library

5.3.1 Discreteness

The problem arises when trying to prove discreteness. At one point in the proof one wants to show that, if the first components of two dependent pairs are equal but the second components are unequal, the complete dependent pair is unequal. The intuitive approach seems to be to prove this by contradiction. After all, one might think that if the dependent pairs are equal the second components should be equal as well. It turns out it is not possible to prove that they are equal in general [10].

It is, however, provable if the first type of the dependent pair is decidable. This has to do with the following insight of Hedberg's [10].

It turns out that while proof irrelevance is independent in Coq (neither provable nor disprovable) one can prove it for equality proofs of discrete types.

Fact 5.3.1 (Hedberg's theorem)**.** *[10]*
Lemma Hedberg (X: eqType) (x y: X) (E E′: x = y) : E = E′.

The proof of this in the accompanying Coq development was heavily inspired by the proof of the theorem in the Ssreflect library [17].

A result of this theorem which Hedberg already proved in his original paper [10] is that discrete types are closed under dependent pairing. During the proof one shows that projection to the second component of a dependent pair preserves equality.

Fact 5.3.2. *[10]*
Let x *and* y *be of type* {x:X & f x} *. If $x = y$ then also $\pi_2\ x = \pi_2\ y$.*

Fact 5.3.3. *[10]*
Let X *be a discrete type and* f: X → eqType*. Then* {x:X & f x} *is a discrete type as well.*

5.4 Construction of a finite type

As usual we define a function enumerating every element of the type to define a finite type.

Definition 5.4.1.

```
Fixpoint toSigTList {X: Type} (f: X → finType) (A: list X) :
list (sigT f) :=
match A with
| nil ⇒ nil
| x::A′ ⇒ (map (existT f x) (elem (f x))) ++ toSigTList f A′ end.
```

Note that because for every *x*, *f x* produces a finite type, we have easy access to ever inhabitant of *f x* through *elem*. Again we will have to prove that the generated list is correct. To do this we first need to know how many elements each inhabitant of *X* generates.

Lemma 5.4.1.
Let X *be a discrete type and* f: X → eqType*. Let* x:X, y: f x *and* A: list (f x)*. Then*
count (map (existT f x) A) (existT f x y) = count A y.

Proof. By induction on *A*.

nil:
In this case this reduces to *nil* = *nil* which is trivially true.

a::A′: We do case analysis on $y = a$.

$\mathbf{y = a}$:
In this case also *existT f x y = existT f x a*. Therefore we need to show *1 + count (map (existT f x) A′) (existT f x y) = 1 + count A′ y*. By the induction hypothesis we get *count (map (existT f x) A′) (existT f x y) = count A′ y* which suffices.

$\mathbf{y \neq a}$:
In this case by Fact 5.3.2 also *existT f x y ≠ existT f x a*. This means we have to prove *count (map (existT f x) A′) (existT f x y) = count A′ y* which is exactly the induction hypothesis.

□

We also need to prove that dependent pairs with the wrong first component are not in the list.

Lemma 5.4.2.
Let X *be a discrete type and* f: X → eqType. *Let* x, x′: X, y: f x *and* A: list (f x). *Then, if $x \neq x'$,* count (map (existT f x) A) (existT f x′ y) = 0.

Proof. By induction on A.

***nil*:**
Again this reduces to *nil = nil*.

***a::A′*:**
$x \neq x'$ therefore also *existT f x a ≠ existT f x′ y*.
This leaves *count (map (existT f x) A′) (existT f x′ y) = 0* to be shown which is the induction hypothesis.

□

Now we can have a look at *toSigTList*. We can reduce counting elements of *toSigTList f A* to counting elements of *A*.

Lemma 5.4.3.
Let X *be a discrete type,* f: X → finType, A: list X *and* s: sigT f.
Then count (toSigTList f A) s = count A (π_1 s).

Proof. By induction on *A*.

***nil*:**
toSigTList f nil is equal to *nil*. And *count nil s = 0 = count nil (π_1 s)*.

***a::A′*:**
s consists of two elements *x:X* and *y: f x*. So the left side becomes

$$count\ (map\ (existT\ f\ a)(elem\ (f\ a)) ++ toSigTList\ f\ A')\ (existT\ f\ x\ y).$$

By Lemma 3.1.5 we can change the left side to

$$count\ (map\ (existT\ f\ a)\ (elem\ (f\ a))\ (existT\ f\ x\ y) + count\ (toSigTList\ f\ A')\ (existT\ f\ x\ y).$$

The induction hypothesis allows us to change this to

$$count\ (map\ (existT\ f\ a)\ (elem(f\ a))\ (existT\ f\ x\ y) + count\ A'\ x.$$

There are two cases:

$\mathbf{a = x}$:

This means the right side of the equation reduces to *1 + count A' x*. Therfore it now suffices to show that *count (map (existT f a) (elem (f a)) (existT f x y) = 1*. We can change this to *count (elem (f x)) y = 1* by Lemma 5.4.1 which is true by *enum_ok*.

$\mathbf{a \neq x}$:

In this case the right side of the equation reduces to *count A' x*. We therefore only have to show *count (map (existT f a) (elem (f a)) (existT f x y) = 0* which is true by Lemma 5.4.2.

□

This is all we need to show correctness of the list produced by *toSigTList*.

Fact 5.4.4.
Let X *be a finite type,* f: X $\rightarrow$ finType *and* s: sigT f. *Then*
count (toSigTList f (elem X)) s = 1.

Proof. By Lemma 5.4.3 it suffices to show *count (elem X) (π_1 s) = 1* which is true by *enum_ok*. □

Theorem 5.4.5. *Let* X *be a finite type and* f *a function* X $\rightarrow$ finType.
Then `finType_sigT f:= FinType (EqSigT f)` *is a finite type such that*
finType_sigT f = sigT f.

Proof. With the coercion *type* the finite type *finType_sigT f* is equal to *EqSigT f* which with the coercion *eqtype* is equal to *sigT f.* □

Chapter 6

Vectors

As already discussed in chapter 3 function types cannot be declared as *finTypes* since they are not discrete. For functions with a finite domain and codomain we can, however, find a replacement which can be declared as a finite type. This replacement is a *vector*. A *vector* in mathematics is usually a collection of objects of a certain kind with a fixed size. For example a mathematical vector of 'type' $\mathbb{R}^n$ is a collection of real numbers of size *n*. Our vectors, however, are not indexed by numbers but by finite types. A Y-vector indexed by a finite type *X* is a collection of *Cardinality X* elements of type *Y*.

Based on cardinality we can define *vector*s. We will use a subtype.

Definition 6.0.2.
`Card_X_eq X Y (A: list Y) :=` $|A| =$ `Cardinality X`

Definition 6.0.3.
`vector (X: finType) (Y: Type) := subtype (Card_X_eq X Y)`

How are *vector*s related to functions? We can interpret a *Y-vector* indexed by *X* as a function $X \to Y$ in the following way: If *n* is the position of *x:X* in *elem X*, we interpret the n^{th} component of the vector as the result of the function for the argument *x*. Ssreflect uses the same idea to represent functions [9, 17]. To endorse the function view of vectors and to make notation easier we use long arrows for vector types.

Definition 6.0.4.
$X \longrightarrow Y$ `:= vector X Y`

We also define the *image* of a vector.

Definition 6.0.5.
The *image* of a vector $X \longrightarrow Y$ is the first component of the dependent pair, i.e. the list of type Y.

This gives us a very nice extensionality principle for *vectors*.

Fact 6.0.6 (Vector extensionality)**.**
If f *and* g *are two vectors with* image f = image g, *then* $f = g$.

Proof.
This is a direct consequence of the extensionality principle for subtypes 5.1.3. □

Because *vectors* are defined as subtypes, *vectors* ranging of discrete types are discrete as well.

Fact 6.0.7.
Equality on vectors ranging over discrete types is decidable.

6.1 Construction of finite type

Before we can declare a finite type for vectors, we first need to obtain a list of all possible vectors. In order to achieve this we first, for a natural number *n* and a list *A*, compute all lists of length *n* containing only elements from *A*. In the function view this means we compute all possible images a function mapping elements from a domain of size *n* to elements of *A* can have.

Definition 6.1.1.

```
Fixpoint images (Y: Type) (A: list Y) (n:  ℕ) :  list (list Y) :=
match n with
| 0 ⇒ [[]]
| S n′ ⇒ concat (map (λ x ⇒  map (cons x) (images A n′)) A)
end.
```

The first case is rather obvious: There is just one list of length 0 namely the empty list. The second case recursively adds each element of *A* to all lists of length *n*′. This time we will proceed differently than usual to prove that our final list is correct. We will use Fact 3.1.2 and show that the list is *dupfree* and every element of the type is in it. While the direct way is possible we need a lot of generalisations and additional arguments which leads to a very convoluted proof. The first step to prove that we can indeed built the list we want is to prove that *images* produces a duplicate free list including all lists of the required length. In order to do this we first need a few lemmas.

Lemma 6.1.1.
If the list A *given to* images *as an argument is non-empty, then the result of* images A n *is a non-empty list.*

This can be proven by induction on *n* using some properties of lists and *map*.

In order to show that the function *images* does indeed return a duplicate free list, we first have to show that the lists which are later concatenated are pairwise disjoint. This takes several steps.

Lemma 6.1.2.
Let X *be a type and* A: list list X *and (x y:X).*
If $x \neq y$, then map (cons x) A *and* map (cons y) A *are* disjoint.

Lemma 6.1.3.
Let X *be a type and* A *a list over* X, B *and* B′ *lists of lists over X, and* x *of type* X. *If B is not empty and $x \notin A$, then* B′ ∈ map (λ y ⇒ (map (cons y) B)) A *implies that* B′ *and* map (cons x) B *are disjoint.*

Proof. Because *B* is not empty it must be of the form *A*′::*B*″. Now we do induction on A.

nil:
In this case *B*′ cannot be in *map (λ y ⇒ (map (cons y) (A′::B″))) nil* because this simplifies to *B*′ ∈ *nil* which is contradictory.

***y::A*″**:
By the second of de Morgan's laws from 2.1.2 and the definition of ∈ the assumption $x \notin y :: A''$ can be changed to $x \neq y$ and $x \notin A''$.
From the assumption *B*′ ∈ *map (λ y ⇒ (map (cons y) (A′::B″)) (y::A″)* we get two cases:

***B*′ = map (cons y) (*A*′::*B*″)**:
This means we have to show that *map (cons y) (A′::B″)* and *map (cons x) (A′::B″)* are disjoint which is true by Lemma 6.1.2.

***B*′ ∈ map (λ y ⇒ (map (cons y) (*A*′::*B*″)) *A*″**:
Now our goal is to show that *B*′ and *map (cons x) (A′::B″)* are disjoint. This is exactly what the induction hypothesis promises provided we can show that $x \notin A''$ and

$B' \in$ *map* $(\lambda\ y \Rightarrow$ *(map (cons y)* $(A'::B''))\ A''$. This is no problem because we have both as an assumption.

□

Lemma 6.1.4.
Let X *be a type and* A: list X *and* B: list list X. *If* B *is not empty and* A *is duplicate free, then any two list* C *and* C′ *with* $C \neq C'$ *which are elements of* map (λ y ⇒ (map (cons y) B)) A *are disjoint.*

Proof. By induction on the derivation of the proof of *dupfree A*.

nil:
In this case both C and C' cannot be an element of *map* $(\lambda\ y \Rightarrow$ *(map (cons y) B)) nil* because this reduces to *nil*.

$x \notin A'$ ***and dupfree*** A':
B must be of the form $A''::B'$ because it is non empty. This leaves us with C and C' as elements of *map* $(\lambda\ y \Rightarrow$ *(map (cons y)* $(A''::B')))\ (x::A')$. We distinguish 4 cases:

C = map (cons x) $(A''::B')$ = C':
This case is impossible because we know that $C \neq C'$.

C = map (cons x) $(A''::B')$ *and* $C' \in$ map $(\lambda$ y $\Rightarrow$ (map (cons y) $(A''::B')))\ A'$:
This leaves us with having to show that *map (cons x)* $(A''::B')$ and C' are disjoint which we obtain using 6.1.3.

C' = map (cons x) $(A''::B')$ *and* $C \in$ map $(\lambda$ y $\Rightarrow$ (map (cons y) $(A''::B')))\ A'$:
Analogous to the previous case.

$C, C' \in$ map $(\lambda\ y \Rightarrow$ (map (cons y) $(A''::B')))\ A'$:
This are exactly the assumptions we need to use the induction hypothesis to prove that C and C' are disjoint.

□

Lemma 6.1.5.
Let X *be a type and* A: list X *and* B: list list X.
Then (∀ C, C ∈ B → disjoint A C) → disjoint A (concat B).

Lemma 6.1.6.
Let X *be a type and* B *be a duplicate free list of duplicate free and pairwise disjoint lists of type* X. *Then* concat B *is duplicate free, as well.*

Proof. By induction on B.

nil:
If B is *nil*, so is *concat B* and *nil* is duplicate free.

A::B′:
We have to show that *concat* $(A::B')$ is duplicate free. For this is suffices to show that both A and *concat* B' are duplicate free and that A and *concat* B' are disjoint.
A is an element of B and therefore duplicate free by assumption.
By the induction hypothesis *concat* B' is duplicate free if B' is a duplicate free list of duplicate free and pairwise disjoint lists. Since $A::B'$ satisfies this property, so does B'.
Now we only need to show that A and *concat* B' are disjoint. By Lemma 6.1.5 it suffices to show that every list $C \in B'$ is disjoint from A. This is true because $A::B'$ is a duplicate free list of pairwise disjoint (and also duplicate free, but this does not matter here) lists.

□

To go on we need to know a property of mappings to duplicate free lists.

Fact 6.1.7. *[13, 14]*
Let X *and* Y *be types,* A: list X *and* f: $X \to Y$ *be a function that behaves like a injective function on the elements of* A, *i.e. when* x *and* y *are in* A, *then* f x = f y → x = y. *Under this condition*
dupfree A → dupfree (map f A).

Now we can essentially prove that the second case of *images* produces a duplicate free list, although our statement will be a bit more general, namely:

Lemma 6.1.8.
Let X *be a type and* A: list X *and* B: list list X *be duplicate free lists. If* B *is not empty, then*
concat (map (λ x ⇒ map (cons x) B) A)) *is duplicate free.*

Proof. By Lemma 6.1.6 it suffices to show that
map (λ x ⇒ map (cons x) B) A is a duplicate free list of duplicate free and pairwise disjoint list.

- First we show that *map (λ x ⇒ map (cons x) B) A)* is duplicate free. By Fact 6.1.7 it suffices to show that *A* is duplicate free and *λ x ⇒ map (cons x) B* behaves like an injective function on the elements of *A*.
  - *A* is duplicate free by assumption.
  - *B* cannot be empty by assumption therefore it must be of the form *A′::B′*.
    Now we have to show that ∀ x y, *map (cons x) (A′::B′) = map (cons y) (A′::B′)* → *x = y*. Now assume that *map (cons x) (A′::B′) = map (cons y) (A′::B′)*. In particular this would mean that *x::A′ = y::A′* and consequently $x = y$.
- Now we show that *map (λ x ⇒ map (cons x) B) A)* contains only duplicate free lists. Let *C* ∈ *map (λ x ⇒ map (cons x) B) A)*. The proof continues by induction on the derivation of *dupfree A*.

  nil:
  This would mean that $C \in nil$ which is impossible.

  $x \notin$ *A′* **and *dupfree A′***:
  There are two cases:

  C = map (cons x) B:
  In this case we have to show that *map (cons x) B* is duplicate free. The function *cons x* is obviously injective so by Fact 6.1.7 all that remains to be shown is that *B* is duplicate free which we have as an assumption.

  C* ∈ *map (λ x ⇒ map (cons x) B) A′:
  This is all we need, to get *dupfree* C with the induction hypothesis.
- We have to show that *map (λ x ⇒ map (cons x) B) A)* only contains pairwise disjoint lists. This is exactly what we have proven in Lemma 6.1.4

□

Now we can finally prove that *images* produces a duplicate free list.

Lemma 6.1.9.
Let Y *be a type,* A:list Y *and* $n : \mathbb{N}$. *Then* images A n *is duplicate free if* A *is duplicate free.*

Proof. Assume A is duplicate free. We do induction on n.

0:
In this case we have to show *dupfree [[]]*, which by use of the second inference rule for *dupfree*[1] is equivalent to *nil* $\notin$ *nil* and *dupfree nil* which is both true.

***S n*′**:
In this case we are in the second case of the definition of *images* 6.1.1. This means we have to show that *concat (map (λ x ⇒ map (cons x) (images A n)) A)* is duplicate free.
We do case analysis on A

nil:
In this case the goal simplifies to the trivial goal *dupfree nil*.

***x::A*′**:
By 6.1.8 it suffices to show that *images (x::A′) n′* and *x::A′* are duplicate free and *images (x::A′) n* is non-empty.

- *x::A′* is duplicate free by assumption.
- We obtain *dupfree (images (x::A′) n)* by the induction hypothesis and *dupfree (x::A′)*.
- By Lemma 6.1.1 it suffices to show that *x::A′* is non-empty which is obviously the case since x is an element of *x::A′*.

□

To prove that the list of all lists of the appropriate form are elements of *images* we first prove that the lengthening of the lists by one works as expected.

Lemma 6.1.10.
Let Y *be a type,* A C: list Y, B: list (list Y) *and* y:Y. *If* $y \in A$ *and* $C \in B$.
Then y::C ∈ concat (map (λ x ⇒ map (cons x) B) A)

Proof. By induction on A.

nil:
This would mean that $y \in nil$ which is a contradiction.

***x::A*′**:
The goal simplifies to *y::C ∈ map (cons x) B ++ concat (map (λ x ⇒ map (cons x) B) A′)*. It suffices to show either *y::C ∈ map (cons x) B* or
y::C ∈ concat (map (λ x ⇒ map (cons x) B) A′. Now either $y = x$ or $y \in A'$.

y = x:
In this case *y::C = x::C ∈ map (cons x) B*. By Fact 5.2.4 it suffices to show that
∃ C′,x::C′ = x::C ∧ C′ ∈ B. Obviously C is such a C′.

y ∈ A′:
With this assumption and the induction hypothesis we get exactly our goal.

□

Lemma 6.1.11.
Let Y *be a type and* A B: list Y. *If* B *is a sublist of* A *in the sense that* ∀ x, x ∈ B → x ∈ A,
then B ∈ images A $|B|$.

Proof. By induction on B.

[1]see [14]

nil:

In this case we have to show that *nil* $\in$ *[nil]* which is trivial.

x::B′:

In this case we have to show
x::B $\in$ *concat (map (λ x ⇒ map (cons x) (images A |B′|)) A′)*. By the previous Lemma 6.1.10 it suffices to show $x \in A$ and $B' \in$ *images A ′*.

- Because *x::B* is a sublist of A and $x \in$ *x::B)* we get $x \in A$.
- By the induction hypothesis it suffices to show that B' is a sublist of A. This is true because *a::B′* is a sublist of A.

□

Remark 6.1.1.
At first this sublist property might seem like a pretty strong requirement. But to get a finite type in the end we will have to choose Y to be a finite type and A to be *elem Y*. This makes this condition trivially true for every list B of type Y

Now we can finally prove that for finite types X and Y *images* produces the list of all Y-vectors of size *Cardinality X*

Lemma 6.1.12.
Let X *and* Y *be finite types. Then for any list* V *over* Y *of size* Cardinality X
count (images (elem Y) (Cardinality X) V = 1.

Proof. By 3.1.3 it suffices to show that $V \in$ *images (elem Y) (Cardinality X)* and *images (elem Y) (Cardinality X)* is duplicate free.

- Because V has size *Cardinality X*, we can reformulate our first goal to $V \in$ *images (elem Y)* $|V|$. By Lemma 6.1.11 this is the case if every element of V is an element of *elem Y* as well. Since every element of V has type Y and by Fact 3.2.2 every inhabitant of Y is an element of *elem Y*, this is true.
- By Lemma 6.1.9 it suffices to show that *elem Y* is duplicate free. By Fact 3.2.1 this is the case.

□

So far we have proven that for finite types X, Y we can produce a list containing every list with elements of Y of length *Cardinality X*. To put it differently: If X and Y are finite types, we can compute the list containing a list representation of the image of every possible function from X to Y. To change this to a *vector* representation we need a proof of the predicate of the *subtype*. More specifically we need to prove that every element of *images (elem Y) (Cardinality X)* does indeed have length *Cardinality X*.

Lemma 6.1.13.
Let Y *be a type and* n:ℕ*. Then* $\forall\ (A:\ list\ Y)\ B,\ B \in$ images A n $\rightarrow |B| = n$.

Proof. By induction on *n*.

0:

Let *A B: list Y*. *images A 0 = [nil]*. This means if $B \in$ *images A 0*, then *B = nil* and $|nil|$ is indeed 0.

S n′:

In this case we have to prove
B $\in$ *concat (map (λ x ⇒ map (cons x) (images A n)) A)* $\to |B|$ = *S n* for some lists *A* and *B*.
Instead of proving this we are going to strengthen the statement to decouple the two lists *A*:
$\forall$ *C, B* $\in$ *concat (map (λ x ⇒ map (cons x) (images C n)) A)* $\to |B|$ = *S n*.
Now we do induction on *A*.

nil:

The first part reduces to $B \in$ *nil* which is equivalent to $\bot$. Therefore the implication is true.

x::A′:

Let *C: list Y* and
B $\in$ *(map (cons x) (images C n) ++ concat (map (λ x ⇒ map (cons x) (images C n)) A′))*. This leaves us with two cases:

B = map (cons x) (images C n):

In this case by Fact 5.2.4 there exists a *D: list Y* such that $B = x :: D$ and
D $\in$ *images C n*. This leaves us with having to prove $|x :: D| = S\ n'$ or equivalently $|D| = n'$. From the first induction on *n* we have the following induction hypothesis:
$\forall$ *(A B: list Y),B* $\in$ *images A n* $\to |B| = n'$. By choosing *A* as *C* and *B* as *D* we obtain the goal.

B* $\in$ *concat (map (λ x ⇒ map (cons x) (images C n)) A′):

Here the strengthening becomes important. The induction hypothesis for the second induction reads as follows:
$\forall$ *(C: list Y), B* $\in$ *concat (map (λ x ⇒ map (cons x) (images C n)) A′)* $\to |B| = S\ n'$.
Had we not strengthened the statement *images C n* would now be *images A′ n*. Unfortunately the list argument of *images* is parametric meaning that it does not get changed in recursive calls. This would mean that we could not use the induction hypothesis because in our assumptions we would only have terms containing
images (x::A′) n. With the strengthening we are, however, perfectly capable of using it. If we choose C for C, the premise is true and we get $|B| = S\ n'$ which is what we need to prove.

□

Having proven that every list in *images* does indeed have the correct length we can finally convert them to *vectors*. In order to do this we simply have to add the proof of correct length to the elements of the list produces by *images*. Unfortunately the definition of the function doing this involves tricky case analysis on dependent types which is not easy to get right. This is why the function doing this is defined by a proof script, which makes the job a lot easier. We still need a trick, though. The idea behind the function will be to recursively modify every element of
images Y (Cardinality X) for finite types *X* and *Y*. However, for recursive functions we need a list as an argument and this list is not always going to be *images Y (Cardinality X)*. By the nature of recursion it is going to be shorter after the first recursive function call. Unfortunately we have only proven that the list are of length *Cardinality X* for *images Y (Cardinality X)* and not for arbitrary list. Fortunately we can circumvent the problem by simply adding an additional argument which is a proof that any element in our list is also an element of *images Y (Cardinality X)*. This property is obviously true for *images Y (Cardinality X)* and any sublist of *images Y (Cardinality X)* produced during recursion.

Definition 6.1.2.

```
Fixpoint extensionalPower (X Y:finType) (L: list (list Y))
(P: L ⊆ images Y (Cardinality X)): list (X ⟶ Y).
```

Proof. If *L* is the empty list, then there is nothing more to do and we return *nil*. If *L=A::L′*, then we do two things.

- We convert A into a *vector*. By assumption we know that $A \in$ *images Y (Cardinalty X)*. Therefore we know by Lemma 6.1.13 that $|A| =$ *Cardinality X*. By 5.1.2 we can convert this proof into a proof for the *pure* version of the predicate. This is all we need to obtain a new *vector* V.
- We recursively call *extensionalPower* with L'. To be able to do this we still need a proof that $L' \subseteq$ *images Y (Cardinality X)*. By assumption we know that this is true for *A::L'*. Consequently this is also the case for L'.

In the end we return the list obtained by the recursive call and add V to it. □

The following fact helps to break down correctness of *extensionalPower* to correctness of *images*. It can be proven by induction on L using extensionality of *vectors* 6.0.6.

Lemma 6.1.14.
For all X, Y, L, P *and vectors* f
count (extensionalPower X Y L P) f = count L (image f).

Now we can finally prove that for finite types X and Y we can indeed enumerate all vectors of type $X \longrightarrow Y$. Note how type inference can infer the missing list *images Y (Cardinality X)* from a very simple proof term.

Fact 6.1.15.
For all finite types X *and* Y *and any vector* f: X ⟶ Y
count (extensionalPower (λ x ⇒ λ y ⇒ y) f = 1.

Proof. By Lemma 6.1.14 it suffices to show that
count (images (elem Y) (Cardinality X)) (image f) = 1. By Lemma 6.1.12 it suffices to show
|*image f*| = *Cardinality X*. This is guaranteed by the definition of *vector* and the type of *f*. □

Now we have everything to define a finite type for vector types of finite types.

Theorem 6.1.16.
Let X *and* Y *be finite types. Then* Y^X *:=* `FinType (EqVect X Y)` *is a finite type such that* $X \longrightarrow Y = Y^X$.

Proof. With the coercion *type* Y^X is equal to *EqVect X Y* which with the coercion *eqtype* is equal to $X \longrightarrow Y$. □

6.2 Cardinality

For finite types $X$and Y any X *indexed Y-vector* has cardinality $Cardinality\ Y^{(Cardinality\ X)}$. To prove this we have to do some ground work. Firstly we need to know that *extensionalPower* does not change the length of the list supplied as an argument and secondly that *images* has the correct length.

Lemma 6.2.1.
Let X *and* Y *be finite types,* L: list (list Y) *and* P *be a proof that* L *is a sublist of* images (elem Y) (Cardinality X). *Then* |extensionalPower L P| = |L|.

Lemma 6.2.2.
Let X *be a type,* A: list X *and* B: list (list X).
In this case | concat (map (λ x ⇒ map (cons x) B) A) | = $|A| * |B|$.

Both these facts can be proven with very simple inductive proofs.

Now we can prove that the function *images* returns a list of the correct length.

Lemma 6.2.3.
Let Y *be a type,* A: list Y *and* n: $\mathbb{N}$. *Then* | images A n| = $|A|^n$.

Proof. By induction on n.

nil:
images A nil = [[]] and $|A|^0 = 1$. Since $|[[]]| = 1$ as well, all is well.

S n':
In this case
images A (S n') = concat (map (λ x ⇒ map (cons x) (images A n)) A). By Lemma 6.2.2 it suffices to prove $|A| * |$*images A n'*$| = |A|^{Sn'}$. By the induction hypothesis | *images A n'* | $= |A|^{n'}$. The rest is trivial.

□

The final proof now only consist of sticking the lemmas together in the correct way.

Theorem 6.2.4.
Let X *and* Y *be finite types. Then* Cardinality Y^X = Cardinality Y $^{\text{Cardinality X}}$.

Proof. By the definition of Y^X and Lemma 6.2.1 it suffices to show
| *images (elem Y) (Cardinality X)* | = *Cardinality* $Y^{Cardinality\ X}$ which we get by Lemma 6.2.3. □

6.3 Conversion between functions and vectors

We have already seen that we can interpret vectors as functions. We will formalise this idea and create ways to switch between the two representations.

To interpret a *vector* as a function we need to be able to look up an element at a specific position in a list. We will return a default element whenever the position does not exist in the list.

Definition 6.3.1.
```
Fixpoint getAt {X: Type} (A: list X) (n:ℕ) (x:X):X :=
match n with
|0 ⇒ match A with
    |nil ⇒ x
    |y::A' ⇒ y
    end
|S n' ⇒ match A with
    |nil ⇒ x
    |x::A' ⇒ getAt A' n' x
    end
end.
```

We also need a function that looks up the position of a value in the list of elements of a finite type.

Definition 6.3.2.
```
Fixpoint getPosition {X: eqType} (A:list X) x :=
match A with
|nil ⇒ 0
|y :: A' ⇒ if decision (x = y) then 0 else 1+ getPosition x A' end.
```

Definition 6.3.3.
Let *X* be a finite type and *x* be of type *X*. We define *index x* as *getPosition (elem X) x*. For the index of *x* we also write #*x*.

Now we can define a function that applies a *vector* to an argument. Again we will use a proof script.

Definition 6.3.4.
```
applyVect (X: finType) (Y: Type) (f:  X ⟶ Y): X → Y.
```

Proof. Let *x:X* be the argument applied to the *vector*. *X* cannot be uninhabited because *x* has type *X*. Therefore *elem X* contains at least one element. The vector *f* is composed of a list *A* of length $Cardinality\ X$ and a proof that this is indeed the length of *A*. *A* cannot be the empty list because *X* is inhabited and must contain at least one element *y*. We return *getAt A (#x) y* □

In Coq we define *applyVect* as a coercion from *vectors* to functions. This allows us to use *vectors* in the way we would use a function.

Now we still miss the opposite direction: From functions to vectors. In order to do this we first need to compute the image of a function over a finite domain. This is easily achieved.

Definition 6.3.5.
```
getImage (X: finType) (Y: Type) (f:  X → Y) := map f (elem X).
```

Of course the produced list contains the right elements and has the correct length:

Lemma 6.3.1.
Let X *be a finite type,* Y *a type,* f: X → Y *and* x:X. *Then* f x ∈ getImage f.

Proof. By definition *getImage f = map f (elem X)*. By Fact 5.2.4 it suffices to show that there is a *y* such that *f y = f x* and *y ∈ elem X*. Obviously *x* is such a *y*. □

Lemma 6.3.2.
Let X *be a finite type,* Y *a type and* f: X → Y. *Then |getImage f| = Cardinality X*

Proof. The *map* function returns a list of the same length as its input list. So it suffices to show *|elem X| = Cardinality X* which is true by the definition of *Cardinality*. □

Now we can convert a function into a vector:

Definition 6.3.6.
```
vectorise {X: finType} {Y: Type} (f:  X → Y): X ⟶ Y :=
exist (pure (Card_X_eq X Y)) (getImage f) (purify (getImage_length f)).
```

Of course we want to know that theses definitions are correct in the sense that *applyVect* and *vectorise* are inverse to each other in both directions.
For the first direction we need to know that looking up an element in *map f A* at the position of *x* in *A* will return *f x* provided that *x* is actually in *A*. This is easily proven by induction on *A*.

Lemma 6.3.3.
Let X: eqType, Y: Type, A: list X y: Y *and* f: X → Y. *If* x:X *is an element of* A, *then*
getAt (map f A) (getPosition A x) y = f x

Now we can prove the first direction. Note that due to the coercion we would not have to write *apply* in Coq and we will omit it here as well.

Theorem 6.3.4.
Let X: finType, Y: Type *and* f: X $\rightarrow$ Y. *Then for every* x:X *we have* (vectorise f) x = f x.

Proof. In the definition of *applyVect* we have already argued that if we apply any vector to an argument, the *image* of the vector cannot be empty. So let's say that *y* $\in$ *image (vectorise f)*. By the definition of *applyVect* the left side is equal to
getAt (image (vectorise f)) (# x) y and this is again, by the definitions of *vectorise* and *getImage*, equal to *getAt (map f (elem X)) (# x) y*. By Lemma 6.3.3 this is equal to *f x* which is what we wanted to prove. □

The opposite direction, that is for any *vector f*, *vectorise f = f*, turns out to be much more complicated to prove. The problem is that vectors are dependent on the finite type *X*. The list *image* must have the same length as *Cardinality X*. This means that we cannot do induction on *image* because in the inductive case the list has become shorter and is not long enough anymore. We can still prove the theorem by splitting the list in two lists which have the correct length if they are concatenated. We then just do induction on the second list. In the inductive case we can put what originally was the first element of the second list at the back of the first one. This allows us to use the induction hypothesis. This means we first have to know a few properties of split lists.

If for some finite type *X elem X* contains *x* at some position *n*, then if we apply a vector *f* where *y* is at position *n* in the image of *f* to *x*, the result is *y*.

Lemma 6.3.5.
Let X *be a finite type,* B, B′: list X *and* x:X . *Also let* Y *be a type,* A, A′: list Y *and* y: Y.
Let elem X = B′ ++ x::B *and* $|A'| = |B'|$. *Then any* X indexed Y-vector f *with the image* A′ ++ y::A *applied to* x *returns* y.

The final proof boils down to showing *map f A = image f* for some vector *f*. We will first show a more general version on which we can perform the induction. Again we will split both *elem X* and *image f*. We will show that if we just apply *f* to every element of the second list of *elem X*, the result will be the second list of *image f*.

Lemma 6.3.6.
Let Y *be a type,* X *be a finite type and* B: list X. *Then for all lists* B′ *such that* B′ ++ B = elem X *and lists (A A′: list Y) such that* $|A| = |B|$ *and* $|A'| = |B'|$ *and any* X indexed vector f *with image* A′ ++ A *the following holds:* map f B = A.

Proof. By induction on *B*.

nil:
In this case we have to show that *A = nil*. Since |*A*| = |*nil*| by assumption and nil is the only list with length zero *A=nil* must indeed be true.

x::B′′:
This means we have to show *f x :: map f B′′ = A*. By assumption $|A| = |x :: B'|$ therefore *A* cannot be empty. Therefore it must be of the form $y :: A''$. Now we need to show two things:

- We show that *f x = y*. This is were we need Lemma 6.3.5 which is indeed applicable because $|A'| = |B'|$ and *elem X = B′++x::B′′*.
- We need to show *map f B′′ = A′′*. This can be shown using the induction hypothesis $\forall$ *(A′′, A′: list Y) (B′: list X),* $|A'| = |B'| \rightarrow |A''| = |B''| \rightarrow$*A′ ++ A′′ = image f* $\rightarrow$ *elem X = B′ ++ B′′* $\rightarrow$ *map f B′′ = A′′*. Of course we choose *A′′* to be *A′′*. This is after all the only instantiation which produces the desired result. Now the splitting finally pays off. We can choose *A′* to be *A′ ++ [y]* and *B′* to be *B′ ++ [x]*. With this instantiations the preconditions hold.

- $|A'$ ++ *[y]*$|$ = $|$ B' ++ *[x]*$|$ because by assumption $|A'| = |B'|$.
- $|A''| = |B''|$ because by assumption $|y :: A''| = |x :: B''|$.
- *(A′ ++ [y])++ A″ = image f* because by assumption *A′ ++ y::A″ = image f*.
- *(B′ ++ [x]) ++ B″ = elem X* because be assumption *B′ ++ x::B″ = elem X*.

□

Now we can finally prove what we wanted to show all along:

Theorem 6.3.7.
Let Y *be a type,* X *be a finite type, and* f: X $\longrightarrow$ Y. *Then* vectorise f = f.

Proof. By the extensionality principle for vectors 6.0.6 it suffices to show that *image (vectorise f) = image f*. This simplifies to *getImage f = image f* or *map f (elem X) = image f*. We use Lemma 6.3.6 with *A′ = nil = B′*, *A= image f* and *B = elem X*. Obviously $|nil|$ = $|nil|$, *nil ++ image f = image f* and *nil ++ elem X = elem X*. All that remains to be shown is that $|imagef|$ = $|elem\ X|$. This is true because *f* is a *X-indexed* vector. So the second component of *f* is a pure version of a proof of this equation. □

Now we have everything we need to change representations between vectors and functions. This will prove to be useful in chapter 9.

Chapter 7

Cardinality

Cardinality is a fundamental property of a finite type and should be examined further. Obviously, if we have an element of a finite type, it's cardinality should be positive.

Fact 7.0.8.
Let X: finType *and* x: X. *Then* Cardinality X > 0.

Proof. By Fact 3.2.2 $x \in$ *(elem X)*. We do case analysis on *elem X*.

nil:
In this case $x \in$ *nil* which is a contradiction.

x′::A:
In this case *Cardinality X* = $|x' :: A| > 0$.

□

7.1 Cardinality of finite types and lists

We investigate the connection between the cardinality of finite types and the cardinality of lists [14, 13]. Cardinality for lists counts the number of different elements in a list.

Intuitively the following should be true:

Fact 7.1.1. *[14, 13]*
For any list A *and* B*:* $A \subseteq B \rightarrow$ card A $\leq$ card B.

Fact 7.1.2.
For any list A *we have* card A $\leq |A|$.

We also have by induction on the proof of duplicate freeness:

Fact 7.1.3. *[14, 13]*
Let X: eqType *and* A: list X. *Then* dupfree A $\rightarrow |A|$ = card A.

This allows us to convert between *Cardinality* and *card*:

Fact 7.1.4.
Let X *be a finite type. Then* Cardinality X = card (elem X).

Proof. By Fact 7.1.3 it suffices to show that *elem X* is duplicate free, which is true according to Fact 3.2.1. □

The cardinality of a finite type also gives us an upper bound for the cardinality of any list over that type.

Fact 7.1.5.
Let X *be a finite type and* A: list X. *Then* card A $\leq$ Cardinality X.

Proof. By Fact 7.1.4 the cardinality of X is equal to the cardinality of *elem* X. By Fact 7.1.1 it suffices to show that $A \subseteq$ *(elem X)*. Since every list over X is a sublist of *elem* X (see Remark 3.2.1) this is true. □

We also get an upper bound for the length of duplicate free lists.

Fact 7.1.6.
Let X *be a finite type and* A: list X. *If* A *is duplicate free, then* $|A| \leq$ Cardinality X.

Proof.
By Fact 7.1.4 *Cardinality X = card (elem X)*. By Fact 7.1.3 and *dupfree A* we get that $|A| =$ *card A*. So it suffices to show *card A* $\leq$ *card (elem X)*. By Fact 7.1.1 it suffices to show $A \subseteq$ *(elem X)* which is true according to Remark 3.2.1. □

7.2 Pigeon hole principles

We can do prove the pigeon hole principles, which are well known from set theory, for finite types.

Remark 7.2.1 (pigeon hole principles).
Let A and B be two finite sets and f be a function mapping elements of A to elements of B.

- If f is injective, then $|A| \leq |B|$.
- If f is surjective, then $|A| \geq |B|$.
- If f is bijective, then $|A| = |B|$.

We will transfer these results to finite types. It turns out to be useful to compute the image of f as we have done when converting a function into a *vector*. Let us first remind ourselves what injective, surjective and bijective mean:

Definition 7.2.1.
Let X and Y be types and f: $X \to Y$.

- f is called *injective* if $\forall x\ y, f\,x = f\,y \to x = y$.
- f is called *surjective* if $\forall y\ \exists x, f\,x = y$.
- f is called *bijective* if it is both *injective* and *surjective*.

The following lemma will prove to be useful.

Lemma 7.2.1.
Let X *be a finite type,* Y *be a type and* f: X $\to$ Y. *If* f *is injective, then* getImage f *is a duplicate free list.*

Proof. Assume *f* is injective. By definition of *getImage* 6.3.5 we have to poof that *map f (elem X)* is duplicate free. By Fact 6.1.7 it suffices to show that *elem X* is duplicate free and *f* behaves like an injective function on the elements of *elem X*. Since *f* is an injection the second condition is certainly true. By Fact 3.2.1 *elem X* is duplicate free. □

With this we can prove the first pigeon hole principle.

Theorem 7.2.2.
Let X *and* Y *be finite types and* f: X $\to$Y *be an injective function. Then* Cardinality X $\leq$ Cardinality Y.

Proof. By Lemma 6.3.2 *Cardinality X* = | *getImage f* |. So we prove
| *getImage f* | $\leq$ *Cardinality Y* instead. By Fact 7.1.6 it suffices to show *dupfree (getImage f)*. In the previouse lemma (7.2.1) we have shown that this is the case for all injective functions of this type and therefore also for *f*. □

For the pigeon hole principle for *surjective* functions we need the following property of *surjective* functions.

Lemma 7.2.3.
Let X *and* Y *be finite types and* f: X $\to$Y *be a surjective function. Then* elem Y $\subseteq$ getImage f.

Proof. Let $y \in$ *elem Y*. By surjectivity of *f* we have an x such that $f\ x = y$. Therefore we can also show *f x* $\in$ *getImage f*. This is true because we have already shown in Lemma 6.3.1 that *getImage* contains every element in the image of *f*. □

We can tackle the complete pigeon hole principle now:

Theorem 7.2.4.
Let X *and* Y *be finite types and* f: X $\to$Y *be a surjective function. Then* Cardinality X $\geq$ Cardinality Y.

Proof. As in the first pigeon hole principle we replace *Cardinality X* with | *getImage f* | by Lemma 6.3.2. We also use Fact 7.1.4 to replace *Cardinality Y* with *card (elem Y)*. With Lemma 7.2.3 and Fact 7.1.1 we obtain *card (elem Y)* $\leq$ *card (getImage f)* and by Fact 7.1.2 we get *card (getImage f)* $\leq$ | *getImage f*|. Now we can build a chain from which we can obtain the goal by transitivity of $\leq$.
card (elem Y) $\leq$ *card (getImage f)* $\leq$ | *getImage f*|. □

The third pigeon hole principle is now a simple corollary.

Corollary 7.2.5.
Let X *and* Y *be finite types and* f: X $\to$Y *be a bijective function. Then* Cardinality X = Cardinality Y.

Proof.
f is bijective and therefore injective and surjective. By the other pigeon hole principles we obtain *Cardinality X* $\leq$ *Cardinality X* and *Cardinality X* $\geq$ *Cardinality X*.
Therefore *Cardinality X* = *Cardinality Y*. □

Chapter 8

Finite Closure Iteration

Iteration of a function until it reaches a fixed point is a frequently used computational idea.

Definition 8.0.2.
A *fixed point* of a function *f* is a value *x* such that $f\ x = x$. We will write *fp f x* to say that *x* is a fixed point of *f*.

Usually it is not easy to formalise a fixed point algorithm in Coq because they are not structurally recursive by nature. With finite types we will, however, be able to formalise a special fixed point algorithm called *finite closure iteration* (FCI). The idea behind *finite closure iteration* is to iteratively add elements to some set until this set fulfils a desired property. If we add elements from a finite type, this process has to end because there are only finitely many values that can be added. To adapt the finite closure iteration algorithm from [14] to finite types we will broadly follow the design and outline in [13], but we prove its correctness slightly differently.

8.1 Fixed point properties

Fixed points of a function *f* are preserved by application of *f*.

Fact 8.1.1.
Let x *be a fixed point of a function* f*. Then* f x *is also a fixed point of* f*.*

If *f* preservers some property *p*, then iteration of *f* preserves this property as well. This gives us a nice induction principle.

Fact 8.1.2 (f-induction)**.** *[14, 13]*
Let X *be a type,* f: $X \to X$ *and* x:X*. Let* p: $X \to \mathbb{P}$ *be a predicate that is preserved by* f*. That is to say:* $\forall y, p\ y \to p\ (f\ y)$*. If* p x*, then* $\forall n : \mathbb{N}, p\ (f^n$ x).

8.2 Admissable functions

For finite closure iteration we want to iterate a function on lists until we get the list of elements of a finite type we desire. We should therefore find sufficient property that we have a fixed point of a function on lists. The development in [14, 13] uses a termination function (a monotonically decreasing function to $\mathbb{N}$) for this purpose. Naturally, if we can give such a function, the iteration must terminate because at some point the function returns zero and cannot decrease the any further. While this idea is very intuitive we will not use it because in our case there is a natural upper bound (the number of elements) instead of a natural lower bound.

We therefore choose a different approach and use the natural upper bound of the cardinality of a finite type to show that we terminate. Since FCI will iteratively add new elements to a list the cardinality increases until we reach a fixed point. Now we formalise this idea.

Definition 8.2.1.
Let *X* be a discrete type and *f: list X →list X*. We call *f* an *admissible* function if it has the following property:
Every list *A: list X* is either a fixed point of *f* or *card (f A) > card A*.

Remark 8.2.1.
We do not use list inclusion because this would allow for infinite reordering or insertion of elements already contained in the list.

We can now make a meaningful statement about iteration of admissible functions.

Lemma 8.2.1.
Let X *be a discrete type,* A: list X *and* f: list X → list X *an admissible function.*
Then ∀ n, fp f (f^n A) ∨ card (f^n A) $\geq n$

Proof. By induction on *n*:ℕ.

0:
f^0 *A* = *A* and *card A* ≥ 0.

Sn′:
We do case analysis on the induction hypothesis. We have to cases:

fp f **(**$f^{n'}$ ***A)***:
In this case by Fact 8.1.1 also *fp f (f (*$f^{n'}$ *A)) = fp f (*$f^{(S\ n')}$ *A)*.

card **(**$f^{n'}$ ***A)*** $\geq$ ***n′***:
Because *f* is admissible there are two possibilities:

fp f **(**$f^{n'}$ ***A)***:
We have seen before that the theorem holds in this case.

card (f ($f^{n'}$ ***A))*** > ***card (***$f^{n'}$ ***A)***:
Since *card (*$f^{n'}$ *A)* is at least *n′ card (f (*$f^{n'}$ *A)) = card (*$f^{(S\ n')}$ *A)* must be at least *S n′*.

□

As a result we know that if we iterate any admissible function *Cardinality X* times we have reached a fixed point.

Theorem 8.2.2.
Let X *be a finite type,* A: list X *and* f: list X → list X *an admissible function.*
Then fp f ($f^{(Cardinality\ X)}$ A)

Proof. By Lemma 8.2.1 every *A: list X* is either a fixed point of *f* or *card (*f^n *A)* $\geq n$. Now choose *n = Cardinality X*. Assume f^n *A* is no fixed point of f. Then *card(f (*f^n *A)) > Cardinality X* because f is monotone. But we have already shown that *Cardinality X* is an upper bound for any list of type *list X* 7.1.5 so this is impossible. It follows that $f^{(Cardinality\ X)}$ *A* is indeed a fixed point of *f*. □

8.3 Finite closure iteration

In the rest of this section we assume *X* to be a finite type. The definitions and proofs in this section very closely resemble their counterparts in [14, 13] but differ in detail because of the adaption from lists to finite types. The proofs of Lemma 8.3.1 and Corollary 8.3.2 are the only exceptions because of the different approach we have taken in the previous section.

Now we know everything needed to implement finite closure iteration. In the end a user of FCI only supplies a decidable predicate *step: list* $X \to X \to \mathbb{P}$ that given a list of already picked elements decides whether a given element can be picked in the next step. We also assume the existence of such a *step* predicate in the future. We can now write a function that either picks an element which we are allowed to pick or returns a proof that all elements we are allowed to pick are already in the list. Again this function is defined using a proof script. We will profit from the decidability features discussed in section 3.3.

Definition 8.3.1.

```
Lemma pick A : {x | step A x ∧ ¬(x ∈ A)} + ∀ x, step A x → x ∈ A.
```

Proof. We do case analysis on ∀ *x, step A x* → $x \in A$ (*X* is a finite type).

If indeed ∀ *x, step A x* → $x \in A$, we are done. Otherwise we change ¬ ∀ *x, step A x* → $x \in A$ into ∃ *x*,¬ *step A x* → $x \in A$ using one of de Morgan's laws 2.1.2. Using the constructive choice function 3.3.6 for finite types we can compute such an *x*. We return said *x* together with a proof of *step A x* ∧ $\neg(x \in A)$ obtained from ¬ *step A x* → $x \in A$ using another of de Morgan's laws 2.1.2. □

We define the function we want to iterate using *pick* to obtain an element which it adds to the list. If there is no such element, it returns the original list.

Definition 8.3.2.

```
FCStep A :=
match (pick A) with
| inl L ⇒ match L with exist _ x _ ⇒ x::A end
| inr _ ⇒ A end.
```

Now we can define the function that computes the final list:

Definition 8.3.3.

`FCIter := FCStep`$^{(\texttt{Cardinality X})}$.

Remark 8.3.1.
Note that $FCStep^{(Cardinality\ X)}$ is still missing its argument. We still need to supply a starting list. In [14] this list is already applied and the empty list. Not committing to the empty list we are more flexible which we will use in chapter 9.

Of course this definition only works if *FCStep* is an admissible function.

Lemma 8.3.1.
FCStep *is admissible.*

Proof. Let *A: list X*. We do case analysis on *step A*.

{x** | **step A x ∧ $\neg(x \in A)$***}***:
> This means *FCStep A* = *x::A* and since $x \notin A$ we have *card (FCStep A)* > *card A*.

$\forall x$***, step A x*** → ***x*** ∈ ***A***:
> In this case *FCStep A* = *A* and *A* is a fixed point of *FCStep*.

Thus FCStep is admissible. □

Consequently *FCIter* computes a fixed point of *FCStep*.

Corollary 8.3.2.
FCIter A *is a fixed point of* FCStep *for every list* A *over type X.*

Proof. *FCIter* is defined as $FCStep^{(\text{Cardinality } X)}$ and thus by Theorem 8.2.2 it suffices to show that *FCStep* is admissible which we have already shown in Lemma 8.3.1. □

Normally one will use *FCIter* to compute some list of elements satisfying a predicate. So it is very likely that you want to know whether every element in the computed list does actually satisfy this predicate. Luckily we get an induction principle for *FCIter* which helps a lot with these tasks.

Definition 8.3.4. [14]
To indicate that every element of a list *A: list X* satisfies a predicate *p:* $X \to \mathbb{P}$ we will write $A \subseteq p$.

Theorem 8.3.3 (FCIter induction)**.**
Let p *be a predicate over* X *and* A: list X.
Then $A \subseteq p \to (\forall\ A\ x, A \subseteq p \to step\ A\ x \to p\ x) \to FCIter\ A \subseteq p$

Proof. Let $A \subseteq p$ and assume that $\forall\ A\ x, A \subseteq p \to step\ A\ x \to p\ x$. The proof proceeds by f-induction (Fact 8.1.2). This means we have to show two things.

- We have to show $A \subseteq p$ which is true by assumption.
- We have to show $\forall\ B, B \subseteq p \to FCStep\ B \subseteq p$. So let $B \subseteq p$ and *x* ∈ *FCStep B*. We need to show *p x*. We do case analysis on *pick B*:

  {y** | **step B y $\wedge \neg(y \in B)$***}***:
  Since $x \in y :: B$ there are two cases:

  x = y:
  In this case *p x* since *step B y* and $B \subseteq p$ and by assumption
  $B \subseteq p \to$ *step B y* $\to$ *p y*.

  x ∈ B:
  In this case *p x* because $B \subseteq p$.

  $\forall y$***, step B y*** $\to$ ***y*** ∈ ***B***:
  In this case *FCStep B* = *B* so $x \in B$. Because $B \subseteq p$ we have *p x*.

□

Every fixed point of *FCStep* contains all pickable values.

Theorem 8.3.4 (Closure)**.**
Let x:X *and* A *be a fixed point of* FCStep. *If* step A x, *then* $x \in A$.

Proof. Because *A* is a fixed point of *FCStep* we know that *pick A* can only return a proof of
$\forall y$*, step A y* $\to$ *y* ∈ *A* because otherwise *FCStep A* ≠ *A*. Therefore since *step A x* also $x \in A$. □

This gives us the corollary for *FCIter*.

Corollary 8.3.5.
For every A: list X *and* x:X. *If* step (FCIter A) x, *then* x ∈ FCIter A.

Proof. By Closure 8.3.4 and the fact that *FCIter* is a fixed point of *FCStep* 8.3.2. □

This is helpful if one wants to show that every element in *FCIter A* satisfies some property.

8.4 Smallest fixed points

When doing fixed point iteration people are often interested in acquiring a *smallest* or *least* fixed point. The *smallest* or *least* fixed point of a function is a fixed point that is smaller than any other fixed point according to some ordering, in our case list inclusion ($\subseteq$).

It is clear that in general *FCIter* does not compute a smallest fixed point simply because we already start with a list we cannot get rid of. We have:

Lemma 8.4.1 (Preservation FCStep)**.**
Let A: list X*. Then* $A \subseteq$ FCStep A*.*

Proof. By case analysis on *pick A*.

{x** | **step A x $\wedge \neg(x \in A)$***}***:
  In this case *FCStep A* = *x::A* and $A \subseteq x :: A$.

$\forall x$***, step A x** $\rightarrow$ **x** $\in$ **A***:
  In this case *FCStep A* = *A* and $A \subseteq A$.

□

As a result *A* is preserved during iteration of *FCStep* as well.

Lemma 8.4.2.
Let A: list X *and* n: $\mathbb{N}$*. Then* $A \subseteq FCStep^n$ A*.*

The proof is by induction on *n* using preservation of FCStep in the inductive case. A simple corollary is that *A* is a subset of *FCIter A*.

Corollary 8.4.3 (Preservation)**.**
Let A: list X*. Then* $A \subseteq FCIter\ A$*.*

Proof. By Lemma 8.4.2. □

Example 8.4.1.
If we take λ A x $\Rightarrow$ $\bot$ *as the step predicate,* nil *would be the smallest fixed point of* FCStep*. However, because of preservation,* A $\subseteq$ FCIter *A. So in this particular case* FCIter *does not compute the smallest fixed point if* $A \neq nil$*.*

This means all we should really be talking about are fixed points containing the list we start with.

Definition 8.4.1 (Least fixed point containing A)**.**
Let *A: list X* and *f: list X* $\rightarrow$ *list X*. A fixed point *B* of *f* is called the *least fixed point containing A* if $A \subseteq B$ and for any other fixed point B' of *f*: $A \subseteq B' \rightarrow B \subseteq B'$.

So does *FCIter A* compute the least fixed point containing *A*? Well, no. The problem is that we could have weird *step* predicates.

Example 8.4.2.
Consider the following step predicate on an option type:
λ A x $\Rightarrow$ if decision (None $\in$ A) then $\bot$ else $\top$*. This means that* FCIter *will add elements until* None *is in the list. Obviously the smallest fixed point containing* A *would be* None::A*. However, depending on the order in which elements get picked by* pick *a lot of other elements could be added to the list before* None*.*

To get *smallest fixed points containing A* we need to ensure that the *step* predicate we use is *consistent*. Intuitively this means that *step A x* cannot become false by adding new values to *A*.

Definition 8.4.2 (Consistency)**.**
A *step* predicate is called *consistent* if $\forall$ *A x, step A x* $\rightarrow \forall$ *A*$'$, $A \subseteq A' \rightarrow$ *step A*$'$ *x*.

Example 8.4.3.
The step *predicate* λ A x $\Rightarrow$ if decision (None $\in$ A) then $\perp$ else $\top$ *from Example 8.4.2 is not consistent because* step nil (Some x) = $\top$ *while* step [None] (Some x) = $\perp$ *although* $nil \subseteq [None]$.

We prove that this condition is indeed sufficient.

Lemma 8.4.4.
If step *is consistent, then for every* A: list X, x: X *and* n: $\mathbb{N}$ *we have:* step A x $\rightarrow$ step ($FCStep^n$ A) x.

Proof. Let *A: list X* and *x: X* such that *step A x*.
By consistency of *step* 8.4.2 it suffices to show that *step A x* and $A \subseteq (FCStep^n\ A$. The first is true by assumption. The second is true by preservation of A during iteration (Lemma 8.4.2). □

Now we prove that any fixed point of *FCStep* that contains a list A also contains any list one obtains by iterating *FCStep* on A.

Lemma 8.4.5.
If step *is consistent, then for every list* A *and fixed point* B *of* FCStep *such that* A $\subseteq$ B *we have for any* n:$\mathbb{N}$ *that* $FCStep^n$ A $\subseteq B$.

Proof. Using f-induction 8.1.2. We have two things to show:

- We have to show $A \subseteq B$ which we have by assumption.
- We have to show that for all $C \subseteq B$ also *FCStep C* $\subseteq B$. We do case analysis on *pick C*:

  {x | step C x $\wedge \neg(x \in C)$***}***:
  This means that *FCStep C* = *x::C*. To show *x::C* $\subseteq B$ it suffices to show $x \in B$ because we already know that $C \subseteq B$. Because B is a fixed point of FCStep's, by Theorem 8.3.4 it suffices to show that *step B x*. Since step is consistent and $C \subseteq B$ and *step B x* we also have *step B x*.

  $\forall x$***, step C x*** $\rightarrow$ ***x*** $\in$ ***C***:
  In this case *FCStep* C = C and $C \subseteq B$ anyway.

□

Now we can finally show that *FCIter A* computes the smallest fixed point containing A provided that *step* is consistent.

Theorem 8.4.6.
If step *is consistent, then for any* A: list X *the list computed by* FCIter A *is the least fixed point of* FCStep *containing* A.

Proof. Two show this we have to show three things:

- We need to show that *FCIter A* is a fixed point of *FCStep* which we have already shown in Corollary 8.3.2.
- We have to show that *FCIter A* contains A i.e. $A \subseteq FCIter\ A$ which is the preservation property 8.4.3 we have shown before.
- We have to show that any fixed point *B* containing A also contains *FCIter A*. Because *FCIter* is defined as iteration we can simply use Lemma 8.4.5.

□

Chapter 9

Case Study: Finite Automata

To find out whether our library is useful we have to use it. We consider the formalisation of finite automata as a test case. Finite automata have already been formalised using the finite types from Ssreflect [2]. For the formalisation I have used an updated [3] version of this original formalisation as inspiration and guidance for my definitions.

Since we will look at the formalisation only to test the library we will not look at all proofs in detail, but only consider those parts were finite types play an important role. We will only define finite automata and show interesting closure and decidability properties. Although this would certainly be interesting we will not endeavour to prove something like the pumping lemma or equivalence to regular expressions.

9.1 DFA

Definition 9.1.1 (Deterministic finite automata)**.** [11]
A *deterministic finite automaton* (DFA) is is a structure (Q,Σ,δ,s,F) where

1. Q is a finite set; elements of Q are called states;
2. Σ is a finite set, the input alphabet;
3. $\delta : Q \times \Sigma \to Q$ is the transition function;
4. $s \in Q$ is the *start state*;
5. F is a subset of Q; elements of F are called *accept* or *final* states

The idea behind a finite automaton is that the automaton "reads" a *word* i.e. a string of *letters* from the alphabet and changes states according to the transition function starting in the start state until the word has been fully read. If the automaton ends up in an accept state, the word is accepted otherwise it is rejected.

We will largely follow the definition above. Like Doczkal and Smolka [3] we will fix the alphabet to some Σ since we want to consider closure properties of automata, which do not make any sense if the alphabet is not identical.

We choose Σ to be some finite type. We can now define words.

Definition 9.1.2. [3]
word := list Σ.

Remark 9.1.1.
For the empty word, i.e. *nil* we will also use the usual notation ϵ.

Since the set of states is finite we will choose it as some finite type as well. To denote the set of final states we will use a decidable predicate (*decPred*). This means we end up with the following definition.

Definition 9.1.3. [3]

```
Record dfa :  Type:=
{
S :> finType;
s:S;
F: decPred S;
```
$\delta_S : S \to \Sigma \to S$
```
}.
```

Note that like Doczkal and Smolka [3] we define *S* as a coercion.

To avoid confusion we will index the projections with the automaton they belong to. For example s_A is the start state of automaton *A*.

9.2 Acceptance and reachability

9.2.1 Acceptance

For this section we assume a DFA *A*. To define acceptance we lift δ_S to words [3]. We call the lifted version δ_S^*.

Definition 9.2.1. [3]
`accept (w:word) :=` F_A `(`$\delta^*_{S_A}$ s_A `w).`

Remark 9.2.1.
Since *F* is a decidable predicate acceptance is decidable.

9.2.2 Reachability

Unlike Doczkal and Smolka we will focus on reachability. We define different notion of reachability. First we define general reachability by an inductive predicate summarised by the following inference rules:

Definition 9.2.2 (reachable)**.**

$$\textbf{refl}\;\frac{}{\text{reachable q q}} \qquad\qquad \textbf{step}\;\frac{\text{reachable } (\delta_S \text{ q x}) \text{ q}'}{\textit{reachable q q}'}$$

We also define reachability with a word. This is easily done using δ_S^*.

Definition 9.2.3.
`reachable_with q w q′ :=` $\delta^*_{S_A}$ `q w = q′.`

We can convert between general reachability and reachability with a word:

Fact 9.2.1.
$\forall$ q q′, reachable q q′ $\leftrightarrow$ $\exists$ w, reachable_with q w q′.

Fact 9.2.2.
Reachable is a transitive relation.

Fact 9.2.3.
$\forall$ q w, reachable q (δ_S^* q w).

Reach

We will now compute a list *reach q* of all states which are reachable from some state *q*. We can use *finite closure iteration* 8.3 to do this. All we need is a suitable *step* predicate. Now it pays off that we have defined *finite closure iteration* in a way that allows to start with arbitrary lists. This means we can put *q* into the list we start with and do not have to consider the case of an empty list separately in the step predicate. Our *step* predicate simply decides for a state *q′* whether it is reachable in one step from a state in the list.

Definition 9.2.4.
```
step_reach (set:  list A) (q:A) :=
∃ q′ x, q′ ∈ set ∧ reachable_with q′ [x] q.
```

Lemma 9.2.4.
step_reach *is* consistent.

Proof. By the definition of *consistency* 8.4.2 we have to show *step_reach B′ q* for some *B′: list A* and *q:A* under the assumption that *B* $\subseteq$ *B′* and *step_reach B q*. By *step_reach B q* we know that there exists some *q′* $\in$ *B* and *x:*Σ such that *reachable_with q′ [x] q*. Because *B* $\subseteq$ *B′* we know that *q′* is also an element of *B′*. Therefore there also exists a *q′* in *B′* such that *reachable_with q′ [x] q* namely *q′*. □

Now we can define reach:

Definition 9.2.5.
reach (q:A) := FCIter step_reach [q].

Corollary 9.2.5.
The list reach *is the least fixed point of* FCStep reach_step *containing [q].*

Proof. By Theorem 8.4.6 is suffices to show that *step_reach* is consistent which we have already shown (9.2.4). □

We also want to know that *reach q* is indeed the list of all elements reachable from *q*. To prove this we will use the induction principle for *FCIter* 8.3.3 and *Closure* for *FCIter* 8.3.5.

Lemma 9.2.6.
Let q:A*. Every element in* reach q *is reachable from* q *(written* reach q $\subseteq$ reachable q*).*

Proof. By *FCIter induction* 8.3.3 it suffices to show two things:

- We have to show *[q]* $\subseteq$ *reachable q.* so we have to show *reachable q q* which is provable by the *refl* rule of *reachable* 9.2.2.
- We have to show: $\forall$ *set q′, set* $\subseteq$ *reachable q* $\rightarrow$ *step_reach set q′* $\rightarrow$ *reachable q q′*.
  So let *set* $\subseteq$ *reachable q* and *q′* such that *step_reach set q′*. This means there is a *q″* $\in$ *set* such that *reachable_with q″ [x] q′*. By Fact 9.2.1 this means *reachable q″ q*. Since *set* $\subseteq$ *reachable q* and *q″* $\in$ *set* we also have *reachable q q″*. By transitivity of reachable 9.2.2 we get *reachable q q′*.

□

Lemma 9.2.7.
Let q *and* q′ *be states of* A. *If* q′ ∈ reach q, *then every state* q′′ *which is reachable from* q′ *is also in* reach q.

Proof. By induction on the derivation of *reachable q′ q′′*.

q′ = q′′:
By assumption *q′ ∈ reach q*.

***reachable (δ_S q′ x) q′′*:**
By induction hypothesis it suffices to show *δ_S q′ x ∈ reach q*.
By the closure property of *FCIter* 8.3.5 it suffices to show *step_reach (reach q) (δ_S q′ x)*. Obviously *reachable_with q′ [x] ((δ_S q′ x))* holds.

□

Lemma 9.2.8.
Let q *and* q′ *be states of* A. *Then* reachable q q′ → q′ ∈ reach q.

Proof. By Lemma 9.2.7 it suffices to show that *q ∈ reach q*. Since *FCIter* preserves the orginal list 8.4.3 *[q] ⊆ reach q* and consequently *q ∈ reach q*. □

Consequently *reach* is correct.

Corollary 9.2.9 (Correctness of reach)**.**
Let q, q′: A. *Then* reachable q q′ ↔ q′ ∈ reach q.

Proof. We have to show both directions.

→: By Lemma 9.2.8.

←: By Lemma 9.2.6.

□

Theorem 9.2.10.
Reachability is decidable, i.e. for all states q *and* q′ *of* A *it is decidable whether* reachable q q′.

Proof. *reachable q q′* is equivalent to *q′ ∈ reach q* and list inclusion is decidable for discrete types. □

Corollary 9.2.11.
Let q q′: A. (∃ w, reachable with q w q′) *is decidable.*

Proof. By Fact 9.2.1 this is equivalent to *reachable q q′* which is decidable by Theorem 9.2.10. □

Remark 9.2.2.
This is a very important result. Remember that while Σ is a finite type *word* is not. Therefore we did not have decidability for existential quantifications over words up until now. While we still cannot decide all existential quantifications over words we can at least decide an interesting one. As we will see very shortly *reach* allows us to decide even more quantifications over words.

9.3 The empty language and sigma star

For this section we assume that A is some DFA. Finite automata describe languages. We call the language of A $\mathcal{L}$(A). $\mathcal{L}$(A) is the set of all words accepted by A. If a word w is accepted by A, we say that it is in $\mathcal{L}$(A) or w $\in \mathcal{L}$(A).

Thanks to *reach* we can decide whether a given automaton computes the empty language or the language of all words Σ^*.

Fact 9.3.1.
A *accepts every word if and only if every state in* reach s *is an accepting state.*

Theorem 9.3.2.
dec ($\forall$ w, $w \in \mathcal{L}$(A)).

Proof. By Fact 9.3.1 it suffices to show that it is decidable whether every state in *reach s* is accepting. Because list inclusion is decidable for discrete types and we can decide whether a state is accepting, this is the case. □

Definition 9.3.1.
We call the language of A *empty* if $\forall$ w, $\neg$ w $\in \mathcal{L}$(A).

To show that we can decide whether a language is *empty* we show that we can build an automaton $\overline{A}$ which accepts the complementary language to A. To do this we simply have to negate F_A [3]. The complement automaton is correct.

Fact 9.3.3. *[3]*
Let w *be a word. Then* accept ($\overline{A}$) w $\leftrightarrow$ $\neg$ accept A w.

As a result we can decide *emptiness* $\mathcal{L}$(A) by simply checking whether $\overline{A}$ is Σ^*.

Fact 9.3.4.
It is decidable whether $\mathcal{L}$(A) is empty.

One result of decidability of emptiness is that we can decide whether there exists an w which is accepted by A. This is one of de Morgan's laws 2.1.2 applied to the definition of *empty*. Note that *word* is no finite type which means that this particular law of de Morgan's cannot be applied here. The trick to proving this is again to use *reach.*

Fact 9.3.5.
$\mathcal{L}$(A) is empty if and only if every state in reach s_A *is not accepting.*

Now we can do the actual decidability proof.

Theorem 9.3.6.
dec ($\exists$ w, accept A w).

Proof. We decide whether $\mathcal{L}$(A) is empty.

yes:
In this case by definition of *empty* there is no such w.

no:
By the equivalence above 9.3.5 we can change $\neg$ *empty $\mathcal{L}$(A)* to
$\neg\forall$ *q*, *q* $\in$ *reach* $s_A \rightarrow \neg$ F_A *q*. We apply de Morgan's laws 2.1.2 (S_A is a finite type). This means we can assume the existence of a q such that $\neg$(*q* $\in$ *reach* $s_A \rightarrow \neg$ F_A *x)*. By another of de Morgan's laws we can change $\neg$(*q* $\in$ *reach* $s_A \rightarrow \neg$ F_A *x)* to *q* $\in$ *reach* s_A and $\neg\neg F_A$ *q*.

Because F_A is a decidable predicate $\neg\neg F_A\ q \leftrightarrow F_A\ q$. Since $q \in$ *reach* s_A there is a w such that *reachable_with s w q* by 9.2.9 and 9.2.1. Because $F_A\ q$ we finally know that A accepts w we have therefore found a w which is accepted by A.

□

By very similar reasoning we also get decidability of $\exists\ w, \neg$ *accept A w*. In fact the proof is almost identical.

Fact 9.3.7.
($\exists\ w, \neg$ accept A w) *is decidable.*

9.4 Automata accepting words

We can construct an automaton accepting only a given word w. First we must build an automaton accepting the empty word ϵ.

Definition 9.4.1 (Epsilon automaton)**.**
We built an automaton only accepting ϵ called *epsilon_autom*. It needs only two states an accepting starting state and another rejecting state to which it switches when it reads a letter.

- We choose *unit + unit* as *S* which is automatically inferred from the other arguments thanks to canonical structure inference.
- *inl tt* is the starting state
- The predicate λ *q: unit + unit* $\Rightarrow$ *if q then* $\top$ *else* $\bot$ is decidable and computes whether a state is accepting.
- λ _ _ $\Rightarrow$ *inr tt* is the transition function.

In particular this definition ensures that we cannot leave the state *inr tt*.

Lemma 9.4.1.
$\forall$ *w*, δ_S^* *epsilon_autom (inr tt) w = inr tt.*

As a result the definition is correct.

Fact 9.4.2.
The automaton epsilon_autom *only accepts* ϵ*.*

As a next step we built an automaton *cons* which accepts all words which start with a given letter *x* and continue with a word accepted by some automaton *A*. To obtain an automaton accepting a word *w* we will of course start with *epsilon_autom* and then apply this automaton for every letter of *w*.

Definition 9.4.2 (cons)**.**
Let *A* be some DFA and *x:*Σ. We build *cons A x* in the following way.

- We choose *A + unit* as *S*. We need an additional state to first read *x*, hence *option A* and we need a non accepting state to go to if the first letter is incorrect, hence unit.
- The starting state is *inl None*.

- The decidable acceptance predicate *F* is described by the following equations.

$$\begin{aligned} F\ (inl\ None) &= \bot \\ F\ (inl\ (Some\ q)) &= F_A\ q \\ F\ (inr\ tt) &= \bot \end{aligned}$$

- The transition function δ_S is described by the following equations:

$$\begin{aligned} \delta_S^*\ inl\ (None)\ x &= inl\ (Some\ s) \\ \delta_S^*\ inl\ (None)\ y &= inr\ tt && \textit{if } x \neq y \\ \delta_S^*\ inl\ (Some\ q)\ y &= inl\ (Some\ (\delta_{S_A}^*\ q\ y)) \\ \delta_S^*\ (inr\ tt)\ y &= inr\ tt \end{aligned}$$

We cannot leave the state *inr tt* in *cons*.

Lemma 9.4.3.
Let A *be a DFA and* x *be a letter. Then* ∀ *w*, δ_S^* *(cons A x) (inr tt) w = inr tt.*

cons computes the correct automaton.

Fact 9.4.4.
Let A *be a DFA and* x *be a letter and* w *be a word. Then* accept (cons A x) (x::w) ↔ accept A w.

We recursively build an automaton which accepts exactly one word *w*.

Definition 9.4.3.
```
Fixpoint exactW (w:  word) :=
match w with
| nil ⇒ epsilon_autom
| (x::w′) ⇒ cons (exactW w′) x end.
```

The resulting automaton is correct.

Fact 9.4.5.
Let w *be some word.* exactW w *only accepts* w.

9.5 Product automata

Several closure properties (insersection, union, difference) of finite automata can be shown using the cartesian product of the states of two automata as a new set of states. The only notable difference between the different constructions is which states are accepting. Because of this Doczkal and Smolka [3] defined a general product automaton which takes some operator as an argument which is used to combine the two acceptance predicates into a new one. This is a very good idea and we will follow the same design.

Definition 9.5.1 (product automaton)**.** [3]
Let *op:* ℙ→ ℙ→ ℙ be some operation such that *op P Q* is decidable if *P* and *Q* are decidable. We define the product automaton *prod A B* of two deterministic finite automata *A* and *B* in the follwing way.

- The set of states *S* is $A \times B$ and can be automatically inferred from the other arguments.
- The starting state *s* is (s_A, s_B).

- The transition function δ_S is described by the equation
  $\delta_S\ (q_1, q_2)\ x = (\delta_{S_A}\ \ q_1\ x,\ \delta_{S_B}\ \ q_2\ x)$
- The acceptance predicate F is defined by the equation
  $F\ (q_1, q_2) = op\ (F_A\ q_1)\ (F_B\ q_2)$

For the δ^*_S function of the product automaton we have the same equation as for the δ_S function.

Fact 9.5.1.
Let A *and* B *be DFA,* w *a word, and* q_1 *a state in* A *and* q_2 *a state in* B.
Then $\delta^*_{S_{prodAB}}\ (q_1, q_2)$ w = ($\delta^*_{S_A}\ q_1$ w, $\delta^*_{S_B}\ q_2$ w)

The resulting product automaton accepts the correct words.

Fact 9.5.2. *[3]*
Let A *and* B *be deterministic finite automata and* w *be a word.*
Then accept (prod A B) w ↔ op (accept A w) (accept B w).

We can now show that regular languages are closed under intersection. This means we can build a deterministic finite automaton $A \cap B$ from two other automata A and B which only accepts words which are accepted by both A and B. We simply need to instantiate the *op* with logical conjunction ($\wedge$).

Corollary 9.5.3.
Let A *and* B *be two deterministic finite automata. Then* accept ($A \cap B$) w ↔ accept A w ∧ accept B w.

Proof. By Fact 9.5.2 because *op* is instantiated by logical conjunction. □

Similarly we can define a union automaton $A \cup B$ for two deterministic finite automata A and B which only accepts words which are accepted by either A or B. This time we need to instantiate *op* with logical disjunction ($\vee$).

Corollary 9.5.4.
Let A *and* B *be two deterministic finite automata. Then* accept ($A \cup B$) w ↔ accept A w ∨ accept B w.

Proof. By Fact 9.5.2 because *op* is instantiated by logical disjunction. □

We also define the difference $A \setminus B$ of two languages of automata A and B. We could do this as $A \cap \overline{B}$ but since we already have the product automaton it is even easier to use just choose
$\lambda\ P\ Q \Rightarrow\ P \wedge \neg\ Q$ as *op*.

Corollary 9.5.5.
Let A *and* B *be two deterministic finite automata.*
accept ($A \backslash B$) w ↔ accept A w ∧ ¬ (accept B w).

Proof. By Fact 9.5.2 because *op* is instantiated by $\lambda\ P\ Q \Rightarrow\ P \wedge \neg\ Q$. □

9.5.1 Language inclusion and equivalence

Language inclusion and equivalence are decidable because we can compute difference.

Definition 9.5.2 (Language inclusion)**.**
We say $\mathcal{L}$(A) includes $\mathcal{L}$(B) ($\mathcal{L}$(B) ⊆ $\mathcal{L}$(A)) if ∀ w, w ∈ $\mathcal{L}$(B) → w ∈ $\mathcal{L}$(A).

Definition 9.5.3 (Language equivalence)**.**
We say $\mathcal{L}$(A) is equivalent to $\mathcal{L}$(B) ($\mathcal{L}$(A)≡ $\mathcal{L}$(B)) if ($\mathcal{L}$(B) ⊆ $\mathcal{L}$(A)) and ($\mathcal{L}$(A)⊆ $\mathcal{L}$(B)).

Language inclusion can also be represented using difference and emptiness.

Fact 9.5.6.
$\mathcal{L}(B) \subseteq \mathcal{L}(A) \leftrightarrow$ empty (B \ A)

Since we can decide emptiness, we can also decide language inclusion and equivalence.

Fact 9.5.7.
Language inclusion is decidable.

Fact 9.5.8.
Language equivalence is decidable.

9.6 Nondeterministic finite automata

We also want to show that regular languages are closed under concatenation and the so called *Kleene operator*. This is much easier using *nondeterministic finite automata* (NFA). The main difference between deterministic and nondeterministic finite automata is that the transition function changes into a transition relation. We represent that relation using a decidable predicate. The definition was again inspired by [3] but is different in some details. It just uses one starting state and uses decidable predicates instead of boolean functions.

Definition 9.6.1.
```
Record nfa := NFA {
Q :> finType;
q0:Q;
Qacc: decPred Q;
δQ : Q → Σ → decPred Q }
```

We lift δ_Q to words as we have done for δ_S. We call the lifted version δ_Q^*. Of course δ_Q^* needs to be a decidable predicate as well.

Definition 9.6.2.
δ_Q^* is decribed by the following equations.

$$\begin{aligned} \delta_Q^*\ q\ nil\ q' = &\quad q = q' \\ \delta_Q^*\ q\ (x :: w)\ q' = &\quad \exists q'',\ \delta_Q\ q\ x\ q'' \wedge\ \delta_Q^*\ q''\ w\ q' \end{aligned}$$

Because existential quantification over finite type are decidable for decidable predicates δ_Q^* is decidable.

δ_Q^* is transitive in the following sense:

Lemma 9.6.1.
Let B *be a nondeterministic finite automaton,* w *and* w′ *words and* q, q′ *and* q″ *states of* B.
Then $\delta_{Q_B}^*$ q w q′ → $\delta_{Q_B}^*$ q′ w′ q″ → $\delta_{Q_B}^*$ q (w++w′) q″.

We also need to define acceptance for NFA.

Definition 9.6.3.
A nondeterministic finite automaton B accepts a word w if there exists an accepting state q in B such that $\delta_{Q_B}^*\ q_{0_B}\ w\ q$.

Because Q is a finite type acceptance of words is decidable for NFA.

9.6.1 Equivalence of NFA and DFA

For NFA to be useful to us they should be equivalent to DFA. To show this we need to be able to convert DFA in NFA in a way that preserves acceptance and vice versa.

Definition 9.6.4 (Conversion from DFA to NFA)**.** [3]
To built an NFA *toNFA A* from a DFA *A* we do the follwing:

- Q = S_A
- $q_0 = s_A$
- $Q_{acc} = F_A$
- The transition relation δ_Q is given by the following equation:

$$\delta_Q \;\; q\, x\, q' = \quad \delta_S\, q\, x = \; q'.$$

This definition is correct in the following sense:

Fact 9.6.2.
Let A *be a DFA,* q, q′ *states of* A *and* w *a word. Then* $\delta^*_{S_A}$ q w = q′ ↔ $\delta^*_{Q_{\text{toNFA } A}}$ q w q′ *and* toNFA A *accepts* w *if and only if* A *accepts* w.

The opposite direction is a more complicated because we have to get rid of cases were we have more than one transition with some letter *x* from a state or where there is no transition with *x* from a state. The usual way to do this is to take the power-set of all states of the NFA *B* as the set of states of the DFA. A transition to several states is modelled by a transition to the set of all the states there is a transition to. We use the same approach but instead of a powerset we will use vectors $\mathbb{B}^{Q_B}$. They represent boolean functions. A vector *f* represents the set of all states *q* for which *f q* = *true*.

Remark 9.6.1.
In the following we will not differentiate between decidable proposition and $\mathbb{B}$. It is clear that one can convert between the two of them which is regularly done in the formalisation. This is something where the Ssreflect library [17] has an advantage because the whole library is based on the convertibility between $\mathbb{P}$ and $\mathbb{B}$.

Definition 9.6.5 (Conversion to DFA)**.**
Let *B*be a NFA. We construct a DFA *toDFA B* in the following way:

- *F* is given by the finite vector type $\mathbb{B}^{Q_B}$ and can be automatically inferred.
- The starting state is the vector obtained from the function $\lambda\, q \Rightarrow\; q = q_0$.
- The set of accepting states is given by the predicate $\lambda f \Rightarrow\; \exists\, q', f\, q \wedge Q_{acc_B}\; q$.
- The transition function is given by $\lambda f\, x \Rightarrow\;$ *vectorise* $(\lambda\, q \Rightarrow\; \exists\, q', f\, q' \wedge \delta_Q\; q'\; x\; q)$.

The correctness proofs for this construction are not particularly interesting. There are, however, two things one should know.

Firstly, one needs to use Theorem 6.3.4 several times to convert vector applications back into function applications. That is the price one has to pay for using functions as finite types.

Secondly, sometimes the coercions do not work as expected. Instead of δ^*_S*(toDFA B) f w q′* one has to write *applyVect* (δ^*_S*(toDFA B) f w) q′* because although there is a conversion which converts *toDFA B* into a finite type and another coercion which converts this particular finite type into a vector and a third coercion which converts this vector into a function Coq only sees the following Coercions:

- NFA $\to$ finType
- finType $\to$ eqType
- eqType $\to$ Type
- vector $\to$ function

Because Coq checks which coercions it can chain before computing the values of the coercions [6], it does not find a Coercion which returns a function. Coq would need to chain all 4 coercions to obtain a function. However Coq does not know that the fourth coercion can be chained to the third coercion. On the other hand for *applyVect (δ_S^*(toDFA B) f w) q′* Coq only needs to chain the first 3 coercions. That these can be chained can be seen from the types alone which is why Coq can do it.

The end result is that the conversion works correctly.

Theorem 9.6.3.
Let B *be some NFA and* w *be a word. Then* accept B w $\leftrightarrow$ accept (toDFA B) w

9.7 Automata concatenation

We can now use NFA to concatenate automata. The basic idea is that transitions starting in the starting state of the second automaton get copied to the accepting states of the first automaton. We need nondeterministic automata because accepting states already have transition so adding transitions from the starting state of a second automaton leads to duplication. This and the construction for the Kleene operator differ from the constructions in [3] because we will not introduce ϵ-NFA as an intermediate step.

Definition 9.7.1 (Concatenation of NFA)**.**
Let *A* and *B* be two NFA. We construct an automaton *concat A B* in the following way.

- The set of states *Q* is given by $Q_A + Q_B$ and can be automatically inferred.
- The starting state is *inl* q_{0_A}.
- The predicate Q_{acc} is given by the following equations:

$$\begin{aligned} Q_{acc}\, \mathit{inl}\ q &= Q_{acc_A}\ q && \textit{if } Q_{acc_B}\ q_{0_B} \\ Q_{acc}\, \mathit{inr}\ q &= Q_{acc_B}\ q \end{aligned}$$

Note that even accepting states of *A* can be accepting if the starting state of *B* is an accepting state.

The predicate describing the transition relation is given by the following equations:

$$\begin{aligned} \delta_Q\ \ (\mathit{inl}\ q)\ x\ (\mathit{inl}\ q') &= \delta_{Q_A}\ q\ x\ q' \\ \delta_Q\ \ (\mathit{inl}\ q)\ x\ (\mathit{inr}\ q') &= \delta_{Q_B}\ q_{0_B}\ x\ q' && \textit{if } Q_{acc_A}\ q \\ \delta_Q\ \ (\mathit{inl}\ q)\ x\ (\mathit{inr}\ q') &= \bot && \textit{if } \neg\,(Q_{acc_A}\ q) \\ \delta_Q\ \ (\mathit{inr}\ q)\ x\ (\mathit{inl}\ q') &= \bot \\ \delta_Q\ \ (\mathit{inr}\ q)\ x\ (\mathit{inr}\ q') &= \delta_{Q_B}\ q\ x\ q' \end{aligned}$$

This construction is correct.

Fact 9.7.1.
Let A *and* B *be NFA and* w, w′ *and* w″ *be words.*
Then accept (concat A B) w $\leftrightarrow$ $\exists$ w′ w″, accept A w′ $\wedge$ accept B w″ $\wedge$ w = w′ ++ w″.

9.8 Kleene operator

The Kleene operator allows any number of repetitions (even 0) of a word in a language. We want to show that regular languages are closed under the Kleene operator. The resulting automaton must accept any combination of $n : \mathbb{N}$ words accepted by the original automaton. This is basically an arbitrary amount of concatenations. Unfortunately we cannot use concatenation because we do not know in advance how often we have to concatenate. Therefore, instead of adding transitions into the beginning of a different automaton, we will simply add transitions from accepting states back into the automaton. We also have to add a new accepting starting state because the original starting state might not have been accepting.

Definition 9.8.1.
We define a nondeterministic automaton B^* for an NFA *B* which realises the Kleene Operator in the following way:

- Q is realised by *option* S_B. This gives us an additional starting state. Again this argument can be inferred automatically.
- The start state q_0 is *None*.
- Q_{acc} is given by the follwing equations:

$$Q_{acc}\ None = \top$$
$$Q_{acc}\ (Some\ q) = Q_{acc_B}\ q$$

- The transition relation is given by the following equations.

$$\delta_Q\ \ q\ x\ None = \bot$$
$$\delta_Q\ \ None\ x\ (Some\ q') = \delta_{Q_B}\ \ q_{0_B}\ x\ q'$$
$$\delta_Q\ \ (Some\ q)\ x\ (Some\ q') = \delta_{Q_B}\ \ q\ x\ q' \lor Q_{acc_B}\ q \land \delta_{S_B}\ q_{0_B}\ x\ q'$$

This construction is correct as well.

Fact 9.8.1.
Let B *be some NFA and* w *be a word.*
Then accept B^* w $\leftrightarrow \exists$ w′, concat w′ = w $\land\ \forall$ w″ $\in$ w′, accept B w″.

As we have seen finite types work well in this test. Finite closure iteration has proved to be particularly helpful for obtaining additional decidability properties. The definition of vectors helped to covert NFA in DFA. The formalisation in [3] had to use the Ssreflect formalisation of finite sets (which do not exist in our library) instead. All in all there are only two minor grievances related to the library. Sometimes coercions behave as bit unintuitively because the necessary type information is not there in advance (see 9.6.1) and the conversion between vector application and function application has to be done manually.

Chapter 10

Conclusion

10.1 Summary

Using *canonical structures*, *coercions* and *type classes* we have defined finite types. We have seen how these features enable us to use finite types very similarly to normal *Type*s and to switch between the two representations. We have shown that decidability is preserved by quantifications over finite types and that this turn out to be extremely useful in a variety of circumstances. We have seen how to construct finite types for products, sums, options and dependent pairs. We have seen how to use subtypes to convert lists over discrete types into finite types. We have defined vectors as a way to represent finite extensional functions and have seen how we can switch between the two representations. We have defined cardinality of finite types and compared it to cardinality of lists. We have proven explicit formulas for the cardinality of most of the finite type constructions. Additionally we showed that the well known pigeon hole principles from set theory also hold for finite types. Finally we have adapted the finite closure iteration [14, 13] to finite types and have shown that it computes smallest fixed points if supplied with a consistent predicate. In the end we have tested the library with a formalisation of finite automata. We have seen that the decidability feature of finite types are very useful in practice and have used finite closure iteration to define reachability. We have also seen a practical use of vectors in a dual role as functions and types.

10.2 Final discussions

We have seen a successful combined use of canonical structure and type classes to achieve inference of discrete and finite types. In this regard there is one question which still needs to be addressed. Could we have made it work just using *type classes* or *canonical structures*?

10.2.1 Merely type classes

This could work if we were prepared to forgo single types for *eqType* and *finType* and to just use *eq_dec* and *finTypeC* instead. This would, however, mean that every definition would need not only the type but also additional arguments. While this might not seem problematic at first, these arguments quickly accumulate. For finite types we already have two additional arguments: one for decidability and one for *finTypeC*. For larger and more complicated structures this quickly becomes messy and slow. This is a well known issue [6] and even authors who advocate just to use *type classes* admit that this can lead to serious problems [16]. Fortunately the additional arguments can at least be automatically inferred.

10.2.2 Merely canonical structures

Just using *canonical structures* would work. It is much harder to declare *toeqType* and *tofinType* this way, because if both *eqType/finType* and the regular *Type* are taken as arguments, this does not express the syntactical relationship they must fulfil. Mahboubi and Tassi [12] describe an approach which would allow the computation of *finType*s and *eqType*s from *Type*s. In fact a version of this algorithm has been implemented in Ssreflect [17] for exactly that purpose. This approach does, however, require a fairly involved use of so called *phantom types* which serves no other purpose than to present the problem in a way that can be solved by *canonical structures*. Since the use of type classes makes the design easier to understand and a class structure such as *finTypeC* is recommended anyway [7, 6] I decided not to implement this using *canonical structures*.

10.3 Differences to Ssreflect

It has already been mentioned that Ssreflect [17, 8] contains a library for finite types, which served as inspiration for this development. While many things are very similar (e.g. how to represent finiteness), there are also some significant differences. One of the most important differences concerns the notions of decidability and discreteness. Ssreflect uses an approach called *boolean reflection* which equates $\mathbb{P}$ and $\mathbb{B}$ [8]. This is means that they often use boolean functions instead of predicates. For example Ssreflect ensures decidablity of equality of a discrete type X by demanding a function $X \to X \to \mathbb{B}$ which computes whether two inhabitants of X are equal [17].

This means that in many cases simple function evaluation will compute a result. This is also possible with our notion of *decidability*. In principle every proof of *dec P* is a function which computes either *P* or ¬ *P*. Therefore it can be used analogously to a function computing either *true* or *false*. However, you might have noticed that the Instances 2.1.2 and 3.4.1 were defined using `Qed.` instead of `Defined.` As a result Coq can infer that an argument of the appropriate type exists, but cannot compute its value. The idea behind this is that

(a) with large examples the computation could get quite costly and

(b) this makes the use of decidability rather explicit. One can always see where it is used and where case analysis is needed. This leads to some very obvious cases in some proofs, e.g. one has to decide $x = x$.

One disadvantage of the approach used in this bachelor thesis is that proofs with many decisions get more complicated. Additionally to the very concise Ssreflect proof script language this is another reason why many proofs in the formalisation of finite automata are longer than the ones in the development by Doczkal and Smolka [3].
This different approach to decidability leads to small differences all over the formalisation. For example Ssreflect uses a self declared structure with a boolean function as a predicate to formalises subtypes instead of the already existing dependent pairs.

Another difference is the construction of *eqType* and *finType*. While the main idea is the same, Ssreflect uses more structures nested more deeply. This design called *packed classes* is highly optimised for efficiency and used all over the mathematical components and Ssreflect libraries. Unfortunately it is confusing to people who have not seen it before. Because one of the main aims of this bachelor thesis was to obtain a minimal and easy to understand formalisation I used the simpler approach described in chapters 2 and 3. For the reader who wants to understand the design used by Ssreflect Garillot's doctoral thesis [6], which contains an excellent explanation, is highly recommended.

Appendix: Realisation in Coq

All results (i.e. theorems, lemmas, facts and corollaries) in this bachelor thesis have been formalised in the proof assistant Coq without the use of axioms. The formalisation is available at https://www.ps.uni-saarland.de/∼menz/bachelor.php. The development was compiled using version 8.5pl2 of Coq from July 26^{th} 2016.

The accompanying Coq development consists of 4 files.

- *External.v* contains preexisting definitions from the base library [13] used in ICL. Some of the definitions in *External.v* have been replaced by more recent versions from the Coq development of Hereditarily finite sets by Smolka and Stark [15]. The main difference is the consistent definition of type class instances for decidability using `Qed`.
- *BasicDefinitions.v* contains definitions and proofs concerning discrete types and definitions of functions and lemmas which do not rely on finite types. This file is dependent on *External.v*
- *FinTypes.v* is the library for finite types. It depends on *BasicDefinitions.v*
- *Automata.v* contains the formalisation of finite automata used to test the library. It depends on *FinTypes.v*.

Excluding *External.v* the formalisation contains 958 lines of specification and 1342 lines of proof. The biggest part is taken up by *FinTypes.v* which accounts for 49.8% of the specification and 50.6% of the proofs.

Bibliography

[1] Pierre Castéran and Matthieu Sozeau. *A gentle introduction to type classes and relations in Coq*. This document presents the main features of type classes and user-defined relations in the Coq proof assistant. May 2014. URL: `http://www.labri.fr/perso/casteran/CoqArt/TypeClassesTut/typeclassestut.pdf`.

[2] Christian Doczkal, Jan-Oliver Kaiser, and Gert Smolka. "A Constructive Theory of Regular Languages in Coq". In: *Certified Programs and Proofs, Third International Conference (CPP 2013)*. Ed. by Geroges Gonthier and Michael Norrish. Vol. 8307. LNCS. Springer, Dec. 2013, pp. 82–97.

[3] Christian Doczkal and Gert Smolka. "Two-Way Automata in Coq". In: *Interative Theorem Proving (ITP 2016)*. To appear. 2016.

[4] *Duality principle. Encyclopedia of Mathematics*. URL: `https://www.encyclopediaofmath.org/index.php/Duality_principle`.

[5] Denis Firsov and Tarmo Uustalu. "Dependently Typed Programming with Finite Sets". In: *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*. WGP 2015. Vancouver, BC, Canada: ACM, 2015, pp. 33–44. ISBN: 978-1-4503-3810-3. DOI: `10.1145/2808098.2808102`. URL: `http://doi.acm.org/10.1145/2808098.2808102`.

[6] François Garillot. "Generic Proof Tools and Finite Group Theory". English. Thesis. Logic in Computer Science [cs.LO]. Ecole Polytechnique X, Dec. 2011. URL: `https://pastel.archives-ouvertes.fr/pastel-00649586`.

[7] François Garillot et al. "Packaging Mathematical Structures". In: *Theorem Proving in Higher Order Logics*. Ed. by Tobias Nipkow and Christian Urban. Vol. 5674. Lecture Notes in Computer Science. Munich, Germany: Springer, 2009. URL: `https://hal.inria.fr/inria-00368403`.

[8] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. *A Small Scale Reflection Extension for the Coq system*. Research Report RR-6455. Inria Saclay Ile de France, 2015. URL: `https://hal.inria.fr/inria-00258384`.

[9] Georges Gonthier et al. "A Modular Formalisation of Finite Group Theory". In: *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics*. TPHOLs'07. Kaiserslautern, Germany: Springer-Verlag, 2007, pp. 86–101. ISBN: 3-540-74590-4, 978-3-540-74590-7. URL: `http://dl.acm.org/citation.cfm?id=1792233.1792241`.

[10] Michael Hedberg. "A Coherence Theorem for Martin-Löf's Type Theory". In: *J. Funct. Program.* 8.4 (July 1998), pp. 413–436. ISSN: 0956-7968. DOI: `10.1017/S0956796898003153`. URL: `http://dx.doi.org/10.1017/S0956796898003153`.

[11] Dexter C. Kozen. *Automata and Computability*. 1st. Ithaca, NY, USA: Springer-Verlag New York, Inc., 1997. ISBN: 0387949070.

[12] Assia Mahboubi and Enrico Tassi. "Canonical Structures for the working Coq user". In: *ITP 2013, 4th Conference on Interactive Theorem Proving*. Ed. by Sandrine Blazy, Christine Paulin, and David Pichardie. Vol. 7998. LNCS. Rennes, France: Springer, July 2013, pp. 19–34. DOI: `10.1007/978-3-642-39634-2\_5`. URL: `https://hal.inria.fr/hal-00816703`.

[13] Gert Smolka. *Base Library for ICL*. Saarland University. 2016.

[14] Gert Smolka and Chad E. Brown. "Introduction to Computational Logic. Lecture Notes SS 2014". Saarland University. 2014.

[15] Gert Smolka and Kathrin Stark. "Hereditarily Finite Sets in Constructive Type Theory". In: *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-27, 2016*. Ed. by Jasmin Christian Blanchette and Stephan Merz. LNCS. To appear. Springer-Verlag, 2016.

[16] Bas Spitters and Eelis van der Weegen. "Type Classes for Mathematics in Type Theory". In: *MSCS, special issue on 'Interactive theorem proving and the formalization of mathematics'* 21 (2011), pp. 1–31. DOI: `10.1017/S0960129511000119`. URL: `http://journals.cambridge.org/action/displayAbstract?aid=8319570`.

[17] Enrico Tassi and Georges Gonthier et al. *Ssreflect*. URL: `http://math-comp.github.io/math-comp/`.

[18] The Coq development Team. *The Coq Proof Assistant The standard library*. 2016. URL: `https://coq.inria.fr/stdlib/`.