University of Bergen
Department of Informatics

Untitled

Author: Andreas Salhus Bakseter

Supervisors: Marc Bezem, Håkon Robbestad Gylterud

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

April, 2023

Abstract

Lorem ipsum dolor sit amet, his veri singulis necessitatibus ad. Nec insolens periculis ex. Te pro purto eros error, nec alia graeci placerat cu. Hinc volutpat similique no qui, ad labitur mentitum democritum sea. Sale inimicus te eum.

No eros nemore impedit his, per at salutandi eloquentiam, ea semper euismod meliore sea. Mutat scaevola cotidieque cu mel. Eum an convenire tractatos, ei duo nulla molestie, quis hendrerit et vix. In aliquam intellegam philosophia sea. At quo bonorum adipisci. Eros labitur deleniti ius in, sonet congue ius at, pro suas meis habeo no.

Acknowledgements

Lorem ipsum

Andreas Salhus Bakseter
Monday 17th April, 2023

Contents

List of Figures

List of Tables

Listings

Chapter 1

Background

1.1 Formalizing Mathematical Problems

1.1.1 Proofs

When solving mathematical problems, we often use proofs to either **warrant** a claim or to **explain** why the claim is true. We can group proofs into two types; *informal* and *formal* proofs.

An informal proof is often written in a natural language, and the proof is adequate if most readers are convinced by the proof [6]. They rely heavily on the readers intuition and often omit logical steps to make them easier to understand for humans [3]. As these proofs grow larger and more complex, they become harder for humans to follow, which can ultimately lead to errors in the proofs' reasoning. This might cause the whole proof to be incorrect [5], and even the result of the proof might be wrong.

A formal proof is written in a formal language, and can be compared to a computer program written in a programming language. Writing a formal proof is more difficult than writing an informal proof. Formal proofs include every logical step, and nothing is left for the reader to assume. This can make them extremely verbose, but the amount of logical errors are reduced [3].

1.1.2 Formalization

1.2 Type theory

Type theory groups mathematical objects with similar properties together by assigning them a "type". Similarily to data types in computer programming, we can use types to represent mathematical objects. For example, we can use the data type `nat` to represent natural numbers.

1.2.1 Propositions as types

The concept of propositions as types sees proving a mathematical proposition as the same process as constructing a value of that type. For example, to prove a proposition P which states "all integers are the sum of four squares", we must construct a value of the type P that shows that this is true for all integers. Such a value is a function that for any input n returns a proof that n is the sum of four squares. Proofs are mathematical objects; thus a proposition can be viewed as having the type of all its proofs (if any!). We can use this correspondance to model a proof as a typed computer program. The power of this concept comes from the fact that we can use a type checker to verify that our program is typed correctly, and thus that the corresponding proof is valid.

1.2.2 Dependent types

(should redo this section probably, too early for Coq example? also find source)

In most programming languages, the data type of a variable does not indicate anything about the value of the variable. Dependent types allow us to define types which depend on the value of the type itself.

Example:

```
Inductive le : nat → nat → Prop :=
  | le_n : forall n, le n n
  | le_S : forall n m, le n m → le n (S m).
```

Listing 1.1: Def. of `le` in Coq, using dependent types

Here, the type `le` has two constructors which depend on the value of the two arguments given to the type. Two natural numbers satisfies the definition of `le` if

1. they are the same number
2. the second number is a successor of `m`, provided that the first number is less than or equal to `m`

Both constructors thus depend on the value of the input; we can check whether the definition holds by repeatedly applying either of the two constructors. If we have done every combination of application, and we are still left with a term that can no longer be reduced, we can conclude that the first number is **not** less than or equal to the second number.

1.3 Proof assistants

Propositions as types allow us to bridge the gap between logic and computing, while dependent types allow us to define more rigorous types which depends on values. The former is a crucial aspect of *proof assistants*, while the latter gives us more expressive power when constructing proofs using a proof assistant. The purpose of a proof assistant is to get computer support for continuity and verify a formal proof mechanically.

1.3.1 Coq

Coq is based on the higher-order type theory *Calculus of Inductive Constructions* (CoC), and functions as both a proof assistant and a dependently typed functional programming language. Coq also allow us to extract certified programs from the proofs of their specification to the programming langauges OCaml and Haskell [8]. Coq implements a specification language called *Gallina*, which is

- an extension of CoC? or based on CoC?
- used to define terms?
- something else?

This is an example of the syntax of Gallina:

```
Inductive nat : Type :=
  | O
  | S : nat → nat.

Definition lt_n_S_n :=
  (fun n : nat ⇒ le_n (S n)) : forall n : nat, n < S n.
```

Listing 1.2: Example of Gallina syntax

Looking at the final definition in the example, we can see the concept of propositions as types in action. `lt_n_S_n` defines a function which takes a natural number `n` as input, and returns a value of the type `forall n : nat, n < S n`, denoted by the colon before the type itself. The return value is therefore a proof of `forall n : nat, n < S n`, and since the definition has been type-checked by Coq, we know that this proof is valid! In this case, the function is `fun : nat ⇒ le_s (S n)`, where `le_n` is a constructor of the type `forall n : nat, n ≤ n`. By applying this constructor to `S n`, we get a value of the type (and a proof) of `forall n : nat, S n ≤ S n`. By Coq's definition of $<$, our initial theorem can be rewritten as `forall n : nat, S n ≤ S n`. This matches the type of our function, and the proof is complete.

Proving theorems like this is not really intutive for a human prover, and that is why Coq gives us the *Ltac* meta-language for writing proofs. Ltac provides us with tactics, which are shorthand syntax for defining Gallina terms

```
Theorem lt_n_S_n : forall n : nat, n < S n.
Proof.
  intro n. destruct n.
  - apply le_n.
  - apply le_n_S. apply le_n.
Qed.
```

Listing 1.3: Example of Ltac syntax

When developing proofs using Ltac, each tactic is executed or "played" one by one, much like an interpreter. The tactics are seperated by punctuation marks. When the use of a tactic causes the proof to depend on the solving of multiple sub-proofs (called "goals"), we can use symbols like "-", "+", and "*" to branch into these sub-proofs and solve their goals independently. Once a goal has been solved, we can move on the next. When there are no more goals, the proof is complete. Coq provides us with tooling that gives us the ability to see our goals and the proof state to further simplify the process [7]. Ltac is not the only proof langauge, with another example being *SSReflect* [2].

1.3.2 Agda

1.3.3 Isabelle

1.3.4 Lean

1.3.5 Higher-Order Logic

1.4 Extraction of programs from verified proofs

Chapter 2

Our case

2.1 Overview

We have used the Coq proof assistant to formalize parts of the proofs of the following paper, Bezem and Coquand [1]. This paper solves two problems that occur in dependent type systems where typings depend on universe-level constraints. We focused on formalizing the proof of theorem 3.2 from the paper. Since this proof is complex enough that mistakes are possible, it was a good candidate for formalization. It also has direct applications to the formalization and verification of the Coq proof assistant itself, since the algorithm outlined in the proof is being tested (ref to coq/metacoq github?) for use in checking loops in Coqs type system.

2.2 Relevant parts of the paper

In the paper, join-semilattices with inflationary endomorphisms are simply called semilattices. An inflationary endomorphism is a function that maps an element to itself or to a greater element in the ordered set. A join-semilattice is a partially ordered set in which any two elements have a least upper bound, called their join.

insert def. of frontier/f here

A semilattice presentation consists of a set V of generators (also called variables) and a set C of constraints (also called relations).

insert def. of S_C and related notation here

2.2.1 Theorem 3.2

Theorem 3.2 states that for any finite semilattice representation (V, C) and any function $f : V \to N^{\infty}$, the least $g \geq f$ that is a model of $\overline{S_C}$ can be computed.

2.2.2 Lemma 3.3

Theorem 3.2 has a special case that is solved by an additional lemma, lemma 3.3. This lemma states that given a finite semilattive presentation (V, C) and a strict subset $W \subset V$, if for any function $f : W \to N^{\infty}$, the least $g \geq f$ that is a model of $\overline{S_C}|W$ can be computed, then for any function $f : V \to N^{\infty}$ with $f(V - W) \subseteq N$, the least $h \geq f$ that is a model of $\overline{S_C} \downarrow W$ can be computed.

Due to time constraints we have given a formalization of this lemma, but not proven it. When testing the algorithm generated by our formalization of theorem 3.2, we have manually edited the code to use the identity function instead of crashing due to the lack of a proof of lemma 3.3.

2.3 Related work

Chapter 3

Approach & Design Choices

When translating an informal proof to a formal proof or specification, one often has to decide how to model certain mathematical objects and their properties. For example, in Coq, there are several implementations of the mathematical notion of a *set*. When choosing which implementation to use, there are often tradeoffs to consider.

3.1 Modeling Sets in Coq

Sets in mathematics are seeminlgy simple structures. A set is a collection of elements, where the elements are of a similiar type. The set cannot contain more than one of the same element (*no duplicates*), and the elements are not arranged in any specific order (*no order*). This is the most basic definition, ignoring more complex paradoxes and different set theories etc...

Sets are easy to work with when writing informal proofs. We do not care about how our elements or sets are represented, we only care about their properties. This does not hold for formal proofs though. In a formal proof, we need to specify exactly what happens when you take the union of two sets, or how you determine if a set contains an element.

One of the most important data structures in functional computer programming is the *list*. Unlike a set, a list *can* contain more than one of the same element, and the elements *are* arranged in a specific order. Lists (in functional programming) usually have the same inductive definition, which is also a formal definition. The definition from Coq's standard library is as follows:

```
Inductive list {A : Type} : list A :=
    | nil : list
    | cons : A → list → list.
```

Listing 3.1: Def. of a list in Coq

Using the cons constructor, we can easily define any list containing any elements of the same type; we can even have lists of lists. The problem is of course that lists are not sets. We want to find a way to include the two important properties of *no duplicates* and *no order* into our definition of lists. In Coq, there are several ways to do this.

3.1.1 List & ListSet

As stated previously, Coq gives us a traditional definition of a list in the **List** module of the standard library. Due to the nature of its definition, it is very easy to construct proofs using induction or case distinction on lists; we only need to check two cases. This list implementation is type polymorphic, meaning any type can be used to construct a list of that type. We do not need to give Coq any more information about the properties of the underlying type of the list other than the type itself.

The **List** module also gives us a tool to combat the possibility of duplicates in a list, with NoDup and nodup. NoDup is an inductively defined proposition that gives evidence (is this correct?) of whether a list has duplicates or not. nodup is a function that takes in a list and returns a list without duplicates. These two can be used effectively in proofs since we still keep the underlying list type, but we also gain additional information about whether the list has duplicates or not.

Having just the implementation of the set structure is rarely enough; we also want to do operations on the set, and reason about these. That is where the **ListSet** module comes in, which defines a new type called set. This type is just an alias for the list type from the **List** module, but the module also contains some useful functions. Most of these functions treat the input as a set in the traditional sense, meaning that they try to preserve the properties of *no duplicates* and *no order*. Examples of some of these functions are set_add, set_mem, set_diff, and set_union. We also get useful lemmas that prove common properties about these functions. One thing to note is that all these functions use bool instead of Prop when reasoning about if something is true or false. This makes them decidable, but it also requires the equality of the underlying type of the set to be decidable. A proof of this for the underlying type must be supplied as an argument to all the functions. An example of such a proof for the `string` type would be:

```
Lemma string_eq_dec :
    forall x y : string, {x = y} + {x <> y}.
Proof.
    (* proof goes here *)
Qed.
```

Listing 3.2: Decidability proof for string equality in Coq

These proofs are often given for the standard types in Coq such as `nat`, `bool` and `string`. As such, they can just be passed to the functions as arguments. This convention of always passing the proof as an argument can be cumbersome and make the code hard to read, but it is a necessary evil to get the properties we want.

The module also gives us some lemmas to transform the boolean functions into propositions, which is especially useful when reasoning about the functions in proofs.

Many of these set functions, such as `set_union`, take in two sets as arguments and pattern match on the structure of one of them. For example, `set_union` pattern matches on the second set given as an argument. This makes proofs where we destruct or use induction on the second argument easy, such as this example:

```
(* example 1 *)
```

Listing 3.3: Easy proof of lemma in `ListSet`

The downside is that even easy and seamingly trivial proofs that reason about the other argument are frustratingly hard, for example:

```
(* example 2 *)
```

Listing 3.4: Hard proof of lemma in `ListSet`

The **ListSet** module gives us no concrete way to combat the order of elements in the set, but there are ways to circumvent the problem. Since we often reason about if an element is in a list, or if the list has a certain length, we do not care about the order of the elements. If we construct our proofs with this in mind, **ListSet** is a viable implementation. There might however be cases where the order of the elements in the lists come into play (i.e. strict equality of two lists), and that is where this implementation falls short.

Another thing to note is because of the polymorphic nature of the `set` type, any additional lemmas proven about a set can be used for any decidable type. This is useful if one needs sets with elements of different types.

3.1.2 MSetWeakList

The Coq standard library also gives us another implementation of sets, **MSetWeakList**. This implementation is a bit more complicated than the previous one, but gives us more guarantees about the properties of the set. The module is expressed as a functor, which in this case is a "function" that takes in a module as an argument, and again returns a module. The module we give to the functor must define some basic properties about the type we want to create a set of, namely equality, decidability of equality and the equivalence relation of (or on?) equality. The output from the functor is a module containing functions and lemmas about set operations, with our input type being the type of the elements of the set.

This means that for every type we want to use as an element in the set, we have to go through this process. In **List** and **ListSet**, we just had to pass in the proof of the equality of the type as an argument to the set functions and lemmas. The structure of the sets in **MSetWeakList** is also a lot more complicated than the simple and intuitive definition of **List**. This makes it harder to reason about the sets in proofs.

3.1.3 Ensembles

Another implementation of sets is given by the **Ensembles** module, which defines the structure of a set as inductive propositions. This means it uses `Prop` instead of `bool`, making **Ensembles** useful for proofs where we do not care about decidability. The biggest downside to this implementation, is that we cannot reason about the size of the set. We can only determine if an element is in the set, not how big the set is. In our case, this makes the **Ensembles** module useless, since the theorem we are formalizing requires us to reason about the size of the set.

Chapter 4

Implementation

4.1 Choice of set implementation

The simplest set (or set-like) implementation in Coq are the **List** and **ListSet** modules. These require minimal knowledge of advanced Coq syntax and behave like lists, making proofs by induction easy. They are also polymorphic, meaning ease of use when making sets of different or self-defined types. Because of these reasons, we chose to go with **List** and **ListSet**.

4.2 The Basics

4.2.1 Atom, Clause and Frontier

The paper [1] uses heavily Horn clauses, which it (and we) simply call clauses. Following the definition of a Horn clause, a clause contains a body of a set of atomic formulas, or atoms, and a single atom as the head [4].

We also define the atoms in the clauses as containing one string and one natural number, since this is sufficient for our implementation.

```
Inductive Atom : Type :=
  | atom : string → nat → Atom.

Notation "x & k" := (atom x k) (at level 80).

Inductive Clause : Type :=
  | clause : set Atom → Atom → Clause.

Notation "ps ↝ c" := (clause ps c) (at level 81).
```

Listing 4.1: Def. of `Atom` and `Clause` in Coq

Note also the `Notation`-syntax, which allow us to define a custom notation, making the code easier to read. The expression on the left-hand side of the := in quotation marks is equivalent to the expression on the right-hand side in parentheses. The level determines which notation should take precedence, with a higher level equaling a higher precedence.

We also want to model functions of the form $f : V \to \mathbb{N}^\infty$, where V is the set of strings (variables) and $\mathbb{N}^\infty$ is the set of natural numbers $\mathbb{N}$ extended by ∞, totally ordered by $n < \infty$ for all $n \in \mathbb{N}$.

We implement this in Coq using two types, `Ninfty` and `Frontier`. `Ninfty` is either a natural number or infinity. `Frontier` is a function from a string (variable) to `Ninfty`.

```
Inductive Ninfty : Type :=
  | infty : Ninfty
  | fin   : nat → Ninfty.

Definition Frontier := string → Ninfty.
```

Listing 4.2: Def. of `Ninfty` and `Frontier` in Coq

Using these definitions of `Atom`, `Clause` and `Frontier`, we can define functions that check whether any given atom or clause is satisfied for any frontier.

```
Definition atom_true (a : Atom) (f : Frontier) : bool :=
  match a with
  | (x & k) ⇒
    match f x with
    | infty ⇒ true
    | fin n ⇒ k ≤ ? n
    end
  end.

Definition clause_true (c : Clause) (f : Frontier) : bool :=
  match c with
  | (conds ⇝ conc) ⇒
    if fold_right andb true (map (fun a ⇒ atom_true a f) conds)
    then (atom_true conc f)
    else true
  end.
```

Listing 4.3: Def. of `atom_true` and `clause_true` in Coq

The infix function $\leq ?$ is the boolean (and decidable) version of the Coq function $\leq$, which uses `Prop` and is not inherently decidable without additional lemmas.

We can also define functions that "shift" the number value of atoms or whole clauses by some amount `n : nat`.

```
Definition shift_atom (n : nat) (a : Atom) : Atom :=
  match a with
  | (x & k) ⇒ (x & (n + k))
end.

Definition shift_clause (n : nat) (c : Clause) : Clause :=
  match c with
  | conds ⇝ conc ⇒
    (map (shift_atom n) conds) ⇝ (shift_atom n conc)
  end.
```

Listing 4.4: Def. of `shift_atom` and `shift_clause` in Coq

Using these definitions, we can now define an important property that will be used later; whether a set of clauses is true for any shift of `n : nat`.

```
Definition all_shifts_true (c : Clause) (f : Frontier) : bool :=
  match c with
  | (conds ⇝ conc) ⇒
      match conc with
      | (x & k) ⇒
          match f x with
          | infty ⇒ true
          | fin n ⇒ clause_true (shift_clause (n + 1 − k) c) f
          end
      end
  end.
```

Listing 4.5: Def. of `all_shifts_true`

4.3 Model

4.3.1 `sub_model`

Given any set of clauses and a function assigning values to the variables, we can determine if this gives us a valid model (reword?).

We translate this propery to Coq as the recursive function `sub_model`. We have two additional arguments `V` and `W`; these are the set of variables (strings) from the set of clauses, and all changed variables (expand on this), respectively. The function `vars_set_atom` simply returns all the variables used in a set of atoms as a set of strings.

```
Fixpoint sub_model (Cs : set Clause) (V W : set string) (f : Frontier) : bool :=
  match Cs with
  | []          ⇒ true
  | (l ⇝ (x & k)) :: t  ⇒
    (negb (set_mem string_dec x W) ||
     negb (
       fold_right andb true
         (map (fun x ⇒ set_mem string_dec x V) (vars_set_atom l))
     ) ||
     all_shifts_true (l ⇝ (x & k)) f
    ) && sub_model t V W f
  end.
```

Listing 4.6: Def. of `sub_model`

4.3.2 `geq`

We want to determine whether all the values assigned to a set of variables from one frontier are greater than or equal to all the values assigned to a set of variables from another frontier. The values are of the type `Ninfty`, and the function only returns true if **all** the values from the first frontier are greater than the values from the second frontier.

```
Fixpoint geq (V : set string) (g f : Frontier) : bool :=
  match V with
  | []          ⇒ true
  | h ::  t  ⇒
    match g h with
    | infty ⇒ geq t g f
    | fin n ⇒
        match f h with
        | infty ⇒ false
        | fin k ⇒ (k ≤ ? n) && geq t g f
        end
    end
  end.
```

Listing 4.7: Def. of `geq`

4.3.3 ex_lfp_geq

We can now combine sub_model and geq to construct a lemma stating that there exists a frontier g that is a model of the set of clauses Cs and is greater than or equal to another frontier f.

```
Definition ex_lfp_geq_P (Cs : set Clause) (V W : set string) (f : Frontier) : Prop :=
  exists g : Frontier, geq V g f = true ∧ sub_model Cs V W g = true.

Definition ex_lfp_geq_T (Cs : set Clause) (V W : set string) (f : Frontier) : Type :=
  sig (fun g : Frontier ⇒ prod (geq V g f = true) (sub_model Cs V W g = true)).

(* we can also use Set, this def. is equivalent to the def. above *)
Definition ex_lfp_geq_S (Cs : set Clause) (V W : set string) (f : Frontier) : Set :=
  sig (fun g : Frontier ⇒ prod (geq V g f = true) (sub_model Cs V W g = true)).
```

Listing 4.8: Multiple defs. of ex_lfp_geq

One thing to note here is that we can define this lemma either as a Prop or as a Type. When using Prop, we define it as a proposition, using standard FOL syntax.

When using Type, we define it as a type, using the sig type constructor in place of exists. We also use the prod type constructor to represent the conjunction of two propositions.

Another thing to note is the difference between Lemma and Definition.

The former is used to define a proposition, while the latter is (usually) used to define a type or a non-recursive function. In this case, we could actually use Definition instead of Lemma, but not the other way around.

The reason for defining ex_lfp_geq as a Type, is that we can then use Coq's extraction feature to generate Haskell code from the Coq definitions. Since ex_lfp_geq plays a central part in the proof of the main theorem, it must be defined as a Type (or as a Set!) to avoid universe inconsistencies when performing extraction.

4.4 The Main Proofs

We have now laid the groundwork for the formalization of theorem 3.2 from the paper [1]. We preceed the definition of theorem 3.2 with two additional definitions, which helps us simplify its definition and the proof of the theorem itself.

4.4.1 pre_thm

Since (in our case) the formal definitions of lemma 3.3, which will be expanded on shortly, and theorem 3.2 share some structure, we define a propsition `pre_thm`:

```
Definition pre_thm (n m : nat) (Cs : set Clause) (V W : set string) (f : Frontier) :=
  incl W V →
  Datatypes.length (nodup string_dec V) ≤ n →
  Datatypes.length
    (set_diff string_dec
      (nodup string_dec V)
      (nodup string_dec W)
    ) ≤ m ≤ n →
  ex_lfp_geq Cs (nodup string_dec W) (nodup string_dec W) f →
  ex_lfp_geq Cs (nodup string_dec V) (nodup string_dec V) f.
```

Listing 4.9: Def. of `pre_thm`

4.4.2 Lemma 3.3

Lemma 3.3 from the paper [1] is used in the proof of theorem 3.2 to solve fill inn explanation here...

We define it using `pre_thm` as follows:

```
Lemma lem_33 :
  forall Cs : set Clause,
  forall V W : set string,
  forall f : Frontier,
    (forall Cs' : set Clause,
     forall V' W' : set string,
     forall f' : Frontier,
     forall m : nat,
      pre_thm (Datatypes.length (nodup string_dec V) - 1) m Cs' V' W' f'
    ) ->
    incl W V ->
    ex_lfp_geq Cs (nodup string_dec W) (nodup string_dec W) f ->
    ex_lfp_geq Cs (nodup string_dec V) (nodup string_dec W) f.
Proof.
  (* ... *)
Qed.
```

Listing 4.10: Lemma 3.3 in Coq

4.4.3 Theorem 3.2

We can now formulate theorem 3.2 using `pre_thm`:

```
Theorem thm_32 :
  forall n m : nat,
  forall Cs : set Clause,
  forall V W : set string,
  forall f : Frontier,
    pre_thm n m Cs V W f.
Proof.
  (* ... *)
Qed.
```

Listing 4.11: Theorem 3.2 in Coq

The proof of theorem 3.2 is based on a double induction on `n` and `m`.

Base case of `n`

The first base case is simple. We want to prove

`ex_lfp_geq Cs (nodup string_dec V) (nodup string_dec V) f`.

If we unfold the definition of `ex_lfp_geq`, we see that we must prove

`{g : Frontier | geq (nodup string_dec V) f g = true}`

and

`sub_model Cs (nodup string_dec V) (nodup string_dec V) f = true`.

The syntax of the first goal is a bit strange, but it is simply a proposition that states that there exists a `g : Frontier` such that `geq (nodup string_dec V) f g = true`. This is easily proven by assuming `g = f` and using the lemma `geq_refl`. The reason for the different syntax is that we are using the `Set` universe which does not have the same syntax as the `Prop` universe, where we could have written

`exists g : Frontier, geq (nodup string_dec V) f g = true`.

The second goal is proven by the fact that the length of `V` is less than or equal to `n`, which in this case is 0. This means that `V` is empty, and we therefore have the new goal of `sub_model Cs [] [] f = true`. This is proven by the lemma `sub_model_W_empty`, which states that `forall Cs V f, sub_model Cs V [] f = true`.

Inductive case of `n`

We start the inductive case of `n` by doing a new induction on `m`.

Base case of `m`

The first base case is similar to the first base case of `n`. We again want to prove

```
ex_lfp_geq Cs (nodup string_dec V) (nodup string_dec V) f.
```

We now apply the lemma `ex_lfp_geq_incl`, which states that

```
forall Cs V W f, incl V W → forall f, ex_lfp_geq Cs W W f → ex_lfp_geq Cs V V f.
```

We give this lemma the arguments of `Cs`, `nodup string_dec V` and `nodup string_dec W`. This generates to new goals,

(1) `incl (nodup string_dec V) (nodup string_dec W)`

and

(2) `ex_lfp_geq Cs (nodup string_dec W) (nodup string_dec W) f.`

The goal (1) is proven by using a hypothesis that states that

```
Datatypes.length (set_diff string_dec (nodup string_dec V) (nodup string_dec W)) ≤ m ≤ n.
```

Since `m` is 0, this means that the set difference of `V` and `W` is empty. We can now apply the lemma `set_diff_nil_incl` on this hypothesis, which states that

```
forall dec V W, set_diff dec V W = [] ↔ incl V W.
```

This gives us a hypothesis identical to our goal (1), and therefore proves it.

The goal (2) is proven by an existing hypothesis.

Inductive case of `m`

Too long :)

4.5 Extraction to Haskell

Using Coq's code extraction feature, we can extract Haskell code from our Coq definitions.

```
Extraction Language Haskell.

Extract Constant map ⇒ "Prelude.map".
Extract Constant fold_right ⇒ "Prelude.foldr".

Extraction "/home/user/path/to/code/ex.hs"
  thm_32
  lem_33.
```

Listing 4.12: Extraction of Coq definitions to Haskell

Coq will automatically determine definitions which depend on one another when doing extraction. In the example above, we would not have needed to specify `lem_33` to be extracted, since `thm_32` already depends on it.

In the next chapter we will go more into detail about the results of the extraction, and the results of the Haskell code ran on some example input.

Chapter 5

Examples & Results

Chapter 6

Evaluation

Chapter 7

Conclusion

Bibliography

[1] Marc Bezem and Thierry Coquand. Loop-checking and the uniform word problem for join-semilattices with an inflationary endomorphism. *Theoretical Computer Science*, 2022. ISSN 0304-3975. doi: https://doi.org/10.1016/j.tcs.2022.01.017.
URL: `https://www.sciencedirect.com/science/article/pii/S0304397522000317`.

[2] Enrico Tassi Georges Gonthier, Assia Mahboubi. The ssreflect proof language.
URL: `https://coq.inria.fr/refman/proof-engine/ssreflect-proof-language.html`. Accessed: 2023-03-21.

[3] Thomas C. Hales. Formal proof. *Notices of the American Mathematical Society*, 55 (11):1370, 2008.
URL: `https://www.ams.org/notices/200811/200811FullIssue.pdf`.

[4] Alfred Horn. On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic*, 16(1):14–21, 1951. doi: 10.2307/2268661.

[5] Roxanne Khamsi. Mathematical proofs are getting harder to verify, 2006.
URL: `https://www.newscientist.com/article/dn8743-mathematical-proofs-getting-harder-to-verify`. Accessed: 2023-18-01.

[6] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*, volume 1 of *Software Foundations*. Electronic textbook, 2022.
URL: `https://softwarefoundations.cis.upenn.edu/lf-current/index.html`. Version 6.2.

[7] The Coq Team. Coqide, .
URL: `https://coq.inria.fr/refman/practical-tools/coqide.html`. Accessed: 2023-03-21.

[8] The Coq Team. A short introduction to coq, .
URL: `https://coq.inria.fr/a-short-introduction-to-coq`. Accessed: 2023-01-18.

Appendix A

Coq examples