

UNIVERSITY OF BERGEN
DEPARTMENT OF INFORMATICS

**A Case Study in Dependent Type Theory:
Extracting a Certified Program from the
Formal Proof of its Specification**

Author: Andreas Salhus Bakseter

Supervisors: Marc Bezem, Håkon Robbestad Gylterud



UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

May, 2023

Abstract

Proofs are an important part of mathematics, but they are not without their flaws. Proofs are written by humans, and humans make mistakes. In this thesis, we explore the use of proof assistants to construct formal versions of informal proofs, and to extract certified programs from these formal proofs. We study a specific case of two problems from dependent type theory, solved by Bezem and Coquand in [1]. Firstly, we ask ourselves: is the informal proof of Theorem 3.2 from [1] correct? Theorem 3.2 is a complex enough proof such that mistakes are possible. Using the Coq proof assistant, we formalize the proof of Theorem 3.2 from [1], to gain more confidence in the correctness of the informal proof. Secondly, is the process of formalization worth the effort? Transforming an informal proof into a formal proof is not necessarily an easy or straightforward process, and there are several approaches to formalization of a proof. Lastly, is the the certified program extracted from the formal proof of Theorem 3.2 useful? The algorithm extracted from the proof has practical applications in some fields of computer science, and we explore the usefulness of the algorithm in a real-world setting.

Acknowledgements

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Andreas Salhus Bakseter

Thursday 25th May, 2023

Contents

1	Background	1
1.1	Proofs & formalization	1
1.2	Type theory	2
1.2.1	Propositions as types	2
1.2.2	Dependent types	2
1.3	Proof assistants	3
1.3.1	Coq	4
1.3.2	Other proof assistants	5
1.3.3	Extraction of programs from verified proofs	6
2	The Case in Question	7
2.1	Overview	7
2.2	Relevant parts of the paper	8
2.2.1	Lemmas and theorems	10
3	Approach & Design Choices	11
3.1	Simplifications for the sake of time	11
3.1.1	Incomplete formal proofs for some purely logical lemmas	11
3.1.2	Leaving out the formal proof of Lemma 3.3	12
3.1.3	Proof of minimality	12
3.2	Modeling finite sets in Coq	12
3.2.1	List & ListSet	13
3.2.2	MSetWeakList	16
3.2.3	Ensembles	17
3.3	Choice of implementation of sets	18
4	Implementation of Logical Notions	19
4.1	Data types	19
4.2	Semantic functions and predicates	22

4.2.1	The function <code>sub_model</code>	22
4.2.2	The functions <code>sub_vars_improvable</code> and <code>sub_forward</code>	23
4.2.3	The function <code>geq</code>	24
4.2.4	The predicate <code>ex_lfp_geq</code>	24
4.3	The main proofs	26
4.3.1	The predicate <code>pre_thm</code>	26
4.3.2	Lemma 3.3	26
4.3.3	Theorem 3.2	27
4.4	Extraction to Haskell	29
5	Examples & Results	31
5.1	Examples using the extracted Haskell code	31
5.1.1	Defining examples for extraction in Coq	31
5.1.2	Necessary alterations to the extracted Haskell code	33
5.1.3	Example output	33
5.2	Real world example	34
5.3	Extension of Listing 5.3 that fails	35
6	Evaluation	39
6.1	Correctness of informal proof	39
6.2	Worthwhileness of formalization	39
6.3	Usefulness of extracted algorithm	40
7	Related & Future Work	41
7.1	Related work: universe consistency checking in Coq	41
7.2	Future work	41
7.2.1	Completing proof of remaining logical lemmas	41
7.2.2	Formal proof of Lemma 3.3	42
7.2.3	Proving minimality of model generated by Theorem 3.2	42
8	Conclusion	43
	Bibliography	45
A	Full proof of Theorem 3.2 in Coq	48

Listings

1.1	<code>vector</code> in Coq, using dependent types	3
1.2	Examples of vectors in Coq	3
1.3	Example of Gallina syntax	4
1.4	Example of Ltac syntax	5
3.1	Proposition for minimal model	12
3.2	Inductive definition of list type in Coq	13
3.3	Decidability proof for string equality in Coq	14
3.4	<code>set_mem</code> lemma in <code>ListSet</code> module	15
3.5	Easy proof of lemma in <code>ListSet</code> module	15
3.6	Impossible proof of lemma in <code>ListSet</code> module	15
3.7	Strict subset relation on lists using <code>incl</code>	16
3.8	Set of atoms in <code>MSetWeakList</code> module	17
3.9	Union in Ensembles module	18
4.1	<code>Atom</code> and <code>Clause</code> in Coq	19
4.2	<code>Ninfty</code> and <code>Frontier</code> in Coq	20
4.3	<code>atom_true</code> and <code>clause_true</code> in Coq	20
4.4	<code>shift_atom</code> and <code>shift_clause</code> in Coq	21
4.5	<code>all_shifts_true</code> in Coq	21
4.6	The function <code>sub_model</code> in Coq	22
4.7	The function <code>sub_vars_improvable</code>	23
4.8	The function <code>sub_forward</code>	23
4.9	Pointwise comparing frontiers with <code>geq</code> in Coq	24
4.10	<code>ex_lfp_geq</code> in Coq, using both <code>Prop</code> and <code>Set</code>	24
4.11	A universe inconsistency in Coq	25
4.12	<code>pre_thm</code> in Coq	26
4.13	Lemma 3.3 in Coq	27
4.14	Theorem 3.2 in Coq	27
4.15	Extraction of Coq definitions to Haskell	29
5.1	Type signature of <code>thm_32</code> in Coq	31

5.2	Type signature of <code>pre_thm</code> in Coq	31
5.3	<code>thm_32</code> example	32
5.4	<code>thm_32</code> example output	34
5.5	Universe hierarchy in <code>Main.v</code>	35
5.6	Universe hierarchy as clauses	35
5.7	<code>thm_32</code> example extended	36
5.8	Output of <code>thm_32</code> example extended	36
5.9	Output of <code>thm_32</code> example extended, with debug	37
5.10	False claim in <code>thm_32</code> proof	37
A.1	Full proof of Theorem 3.2 in Coq	48

Chapter 1

Background

1.1 Proofs & formalization

When solving mathematical problems, we often use proofs to either **justify** a claim or to **explain** why the claim is true. We can distinguish between two types of proofs; *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 [10]. Such proofs rely heavily on the reader's intuition and often omit logical steps to make them easier to understand for humans [6]. As these proofs grow larger and more complex, they become harder for humans to follow, which can ultimately lead to errors in the proofs' logic. This might cause the whole proof to be incorrect [7], and even the claim justified by 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 formal proofs extremely verbose, but the amount of logical errors is reduced [6]. The only possible errors in formalized proofs are false assumptions and/or flawed verification software. When we talk about *formalization*, we mean the process of translating an informal proof into a formal proof.

1.2 Type theory

Type theory groups mathematical objects with similar properties together by assigning them a "type". Similarly 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, or we can create our own data types which allows us to represent e.g., clauses in logic.

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 a type, in this case, of the proposition as a type. For example, to prove a proposition P which states "all natural numbers are the sum of four squares", we must construct a value of the type P that shows that this is true for all natural numbers. Such a value is a function that for any input n returns a proof that n is the sum of four squares, that is, return four numbers a, b, c, d and a proof that $n = a^2 + b^2 + c^2 + d^2$. 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. Often, the proof can be used to compute something, i.e., the numbers a, b, c, d mentioned above [20].

1.2.2 Dependent types

Dependent types allow us to define more rigorously types that depend on values. This is important for propositions as types. For example, the proposition P in Section 1.2.1 uses $n = a^2 + b^2 + c^2 + d^2$, which is a type depending on five values n, a, b, c, d . Since Coq has support for dependent types, we can give an example to illustrate what this means. We consider the type `vector`, which is a type polymorphic (the elements of the list can be of any same type) finite list with a fixed length, which is the value on which the type depends, a natural number.

```

Inductive vector (A : Type) : nat → Type :=
| Vnil : vector A 0
| Vcons : forall (h : A) (n : nat), vector A n → vector A (S n).

```

Listing 1.1: `vector` in Coq, using dependent types

This definition gives us a type with two constructors:

- `Vnil` has the type of `vector A 0`, and represents the empty vector.
- `Vcons` has type of `forall (h : A) (n : nat) vector A (S n)`, where `h` is the head of the vector and `n` is the length of the vector, where both are given to the constructor as arguments. This makes the type of a *vector* *depend* on its length.

In this scenario, the length of a *vector* is fixed by the argument `n : nat` and the term `vector A n → vector A (S n)`. Any definition of a *vector* must adhere to this term, and is checked at compile time. An example of a valid and invalid definition is:

```

(* valid definition; (S 0) equal to 1 *)
Definition vec_valid : vector string 2 :=
  Vcons string "b" 1 (Vcons string "a" 0 (Vnil string)).
(* invalid definition; (S 0) not equal to 2 *)
Fail Definition vec_invalid : vector string 2 :=
  Vcons string "b" 2 (Vcons string "a" 0 (Vnil string)).

```

Listing 1.2: Examples of vectors in Coq

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 rigorously types which depend on values, thus enhancing the expressivity of simple types. The former is a crucial aspect of *proof assistants*, while the latter gives us more expressive power when constructing proofs using a proof assistant. An example of the expressive power of dependent types is the fact that we can define predicates that depend on the value of a term, e.g. a predicate that checks if a number is even. The purpose of a proof assistant is to get computer support to construct and verify a formal proof mechanically.

1.3.1 Coq

Coq is based on the higher-order type theory *Calculus of Inductive Constructions* (CIC), 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 languages OCaml and Haskell [14]. Coq implements a specification language called *Gallina*, which allows us to define logical objects within Coq. These objects are typed to ensure their correctness (is quote too direct?), and the typing rules used are from CIC [12].

This is an example of the syntax of Gallina:

```
Inductive nat : Type :=  
  | 0  
  | S : nat → nat.  
  
Definition lt_n_S_n :=  
  (fun n : nat ⇒ le_n (S n)) : forall n : nat, n < S n.
```

Listing 1.3: 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_n (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 intuitive 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 a kind of shorthand syntax for defining Gallina terms [4]. Using Ltac, we can rewrite the proof from 1.3 as such:

```

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 1.4: 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 separated 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 [13]. Ltac is not the only proof language, with another example being *SSReflect* [5].

1.3.2 Other proof assistants

Agda

Agda is a dependently typed functional programming language based on Martin-Löf's intuitionistic type theory. Unlike Coq, Agda does not use tactics. However, by using proposition as types, Agda can also function as a proof assistant [2].

Isabelle

Lean

Lean is a dependently typed functional programming language and theorem prover. The goal of Lean is to establish a connection between the automated and interactive processes of proving theorems. Like Coq, Lean is also based on the type theory of CIC [3].

Higher-Order Logic

HOL is a family of several theorem provers, based on the concept of higher-order logic.

1.3.3 Extraction of programs from verified proofs

By the the notion of propositions as types, we can use a proof assistant to prove the correctness of a program. However, we can also extract a program from a proof of its correctness. This type of code extraction is a common feature of proof assistants. The extracted program is guaranteed to be correct by the type system of the proof assistant, and the resulting code can be extracted to several programming languages, such as Haskell and OCaml (as is the case for Coq) [14].

Chapter 2

The Case in Question

2.1 Overview

Recall the three questions posed in the abstract. Firstly, we want to use the Coq proof assistant to formalize parts of the proofs of the following paper, by Bezem and Coquand [1]. This paper solves two problems that occur in dependent type systems where typings depend on universe-level constraints, and the two problems can be summarized as follows:

1. The uniform word problem for an equational theory of semilattices with one inflationary endomorphism can be solved in polynomial time.
2. Loop-checking in a type system with universe-level constraints can be done in polynomial time.

From [1], we get a definition of a uniform word problem for an equational theory T :

The uniform word problem for an equational theory T is to determine, given a finite set E of relations between generators, whether a given relation is provable from E in T .

In our case, the equational theory T is the theory of join-semilattices with one inflationary endomorphism. This problem is shown in [1] to be solved in polynomial time, and provability in our case of a semilattice theory is shown to be equivalent to provability in Horn clauses. We will explain in more detail in the next subsection what join-semilattices inflationary endomorphisms and Horn clauses are. Loop-checking, in our case, means determining if a set of constraints on some set of variables generate an infinite loop between constraints. This is important for algorithms such as a type-checking algorithm where type universe levels have to be determined, and the algorithm has to be decidable. We have focused on formalizing the proof of Theorem 3.2 from [1]. Since this proof is complex enough that mistakes are possible, it is a good candidate for formalization. By formalizing the proof and then verifying it using Coq, we can be sure that it is correct.

Secondly, we are also interested in finding out whether or not the process of formalizing the proof can be completed in a reasonable amount of time, and with a reasonable amount of effort. Is the promise of a complete verification of the proof worth the effort of formalizing it?

Finally, by using an advanced feature of Coq, we can also extract an algorithm from the proof, where this algorithm is certified to produce correct results. As an aside, this algorithm has direct applications to the formalization and verification of the Coq proof assistant itself, as Coq employs a similar algorithm to resolve universe-level constraints in its type system.

2.2 Relevant parts of the paper

We will paraphrase some key concepts from [1] that are needed to understand our implementation and formalization.

We will call join-semilattices with inflationary endomorphisms simply semilattices. A *join-semilattice* is a partially ordered set in which any two elements have a least upper bound, called their join. The *join* of a semilattice is a binary operation \vee that is commutative, associative and idempotent. An *inflationary endomorphism*, in our case, is a function that maps an element to itself or to a greater element in the ordered set. We denote the inflationary endomorphism as $^+$, for every lattice element x , x^+ is also called the successor of x . The fact that x is an endomorphism means that we get the following properties: $(x \vee y)^+ = x^+ \vee y^+$ and $x \vee x^+ = x^+$.

A *semilattice presentation* consists of a set V of *generators* (also called *variables*) and a set C of *relations* (also called *constraints*). For a semilattice t and $k \in \mathbb{N}$, $t + k$ denotes the k -fold successor of t , thus making $t + 0 = t$ and $t + 1 = t^+$. A term over V has the form $x_1 + k_1 \vee \dots \vee x_m + k_m$, where $x_i \in V$ and $k_i \in \mathbb{N}$. Every semilattice expression is equal to a term over V .

Horn clauses are propositional clauses $A \rightarrow b$, with a non-empty body A and conclusion b . The atoms are of the form $x + k$, where $x \in V, k \in \mathbb{N}$. We call this special form of Horn clauses simply *clauses*.

A relation is an equation $s = t$, where s and t are terms over V . A constraint (relation), like $a \vee b = c^+$ (with $a, b, c \in V$), expresses a relation between the generators (variables) a, b, c . For each constraint $s = t$, we generate clauses by replacing join by conjunction, denoted by \wedge , and \geq by implication. E.g. for the constraint $a \vee b = c^+$, we first derive $a \vee b \geq c^+$ and $c^+ \geq a \vee b$. From $a \vee b \geq c^+$, we generate the clause $a, b \rightarrow c^+$. From $c^+ \geq a \vee b$, we derive $c^+ \geq a$ and $c^+ \geq b$. We then generate the clause $c^+ \rightarrow a$ from the former, and the clause $c^+ \rightarrow b$ from the latter. This generated set of clauses is denoted by $S_{s=t}$, which in our example makes it $S_{a \vee b = c^+} := \{a, b \rightarrow c^+, c^+ \rightarrow a, c^+ \rightarrow b\}$. We define S_C as the union of all $S_{s=t}$, where $s = t$ is a constraint in C . From the axiom $x \vee x^+ = x^+$, we can derive the following clauses: $x, x^+ \rightarrow x^+$, $x^+ \rightarrow x^+$ and $x^+ \rightarrow x$. We are only interested in the last one, which we call a *predecessor clause*. A predecessor clauses ensures that any atom $x + k$ that is true in a model, also has $x + k - 1$ true in that model.

In [1, Theorem 2.2, p. 3] it is proven that provability in the semilattice theory is equivalent to Horn clause provability, making it so that we can continue with the latter in the rest of this thesis.

We define closure under shifting upwards as follows: if $A \rightarrow b$ is in the set of clauses, then so must $A + 1 \rightarrow b + 1$ be. $A + 1$ denotes the set of atoms of the form $a + 1$, where $a \in A$. Given a finite semilattice presentation (V, C) , and a subset $W \subseteq V$, we denote by $\overline{S_C}$ the smallest set of clauses that is closed under shifting upwards, by $\overline{S_C} \upharpoonright W$ the set of clauses in $\overline{S_C}$ mentioning only variables in W , and by $\overline{S_C} \downarrow W$ the set of clauses in $\overline{S_C}$ with conclusion over W .

A function $f : V \rightarrow \mathbb{N}^\infty$ specifies a downward closed set of atoms, namely $\{v + k \mid v \in V, k \in \mathbb{N}, k \leq f(v)\}$, where \mathbb{N}^∞ is the set of natural numbers extended with ∞ . This set contains all atoms $v + k$ if $f(v) = \infty$. The sets specified by such functions are models of $\overline{S_C}$.

2.2.1 Lemmas and theorems

Lemma 3.1 ([1, p. 3]) *Given $f : V \rightarrow \mathbb{N}^\infty$, and a clause $A \rightarrow b$, let P be the problem whether or not $A + k \rightarrow b + k$ is satisfied by f for all $k \in \mathbb{N}$. Then P is decidable.*

The proof of Lemma 3.1 demonstrates that the problem P is decidable, meaning we can indeed write an algorithm that determines whether or not the problem holds for all $k \in \mathbb{N}$. Lemma 3.1 is also crucial for making case distinctions in further proofs, since we know that any S_C is finite.

Theorem 3.2 ([1, p. 3]) *Let (V, C) be a finite semilattice presentation. For any $f : V \rightarrow \mathbb{N}^\infty$ we can compute the least $g \geq f$ that is a model of $\overline{S_C}$.*

Theorem 3.2 has a special case that is solved by an additional lemma:

Lemma 3.3 ([1, p. 3]) *Let (V, C) be a finite semilattice presentation. Let W be a strict subset of V such that for any $f : W \rightarrow \mathbb{N}^\infty$ we can compute the least $g \geq f$ that is a model of $\overline{S_C} \upharpoonright W$. Then for any $f : V \rightarrow \mathbb{N}^\infty$ with $f(V - W) \subseteq \mathbb{N}$, we can compute the least $h \geq f$ that is a model of $\overline{S_C} \downarrow W$.*

Chapter 3

Approach & Design Choices

When translating an informal proof or specification to a formal proof, one often has to decide how to model certain mathematical objects and their properties, and which parts of a proof that can be simplified or left out entirely. Our formalization effort has been focused mostly on Theorem 3.2 and its proof. We have left aside two concepts from [1]:

- the relation between lattices and Horn clauses
- complexity theoretic claims about the problems

These are sections 2 and 4 from [1], respectively. We have also made some simplifications to various parts of our formalization, which we will explain in more detail in the next subsection.

3.1 Simplifications for the sake of time

3.1.1 Incomplete formal proofs for some purely logical lemmas

We have chosen to not spend too much time fully completing the formal Coq proofs of some purely logical lemmas, which are mainly used as intermediate steps in the proof of Theorem 3.2. When we say "purely logical", we mean that they do not contribute anything to the constructive part of the proof, and are not used in the extracted algorithm in any way (we will expand on how most logical lemmas are removed by extraction in

Section 4.2.4). They are purely logical in the sense that they only serve to prove the correctness of the proof. As we will see later in this section, this is mainly due to very complex or impossible-to-prove formal proofs of lemmas in our set implementation being trivial when proving the same lemma informally. The formal proofs of these lemmas are however not very interesting, which is why we have not spent too much time on them.

3.1.2 Leaving out the formal proof of Lemma 3.3

We have included a formulation of this lemma, but not a formal proof. 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. This simplification is sufficient for a surprising large number of problems, but not all; the limitations of this simplification will be explained in more detail in Section 5.3.

3.1.3 Proof of minimality

In our proof of Theorem 3.2, we have chosen to omit proving the minimality of the model generated by the algorithm. Our algorithm does however generate a minimal model, but we have not proven that it does.

To prove that the model generated by our algorithm is minimal, we would have had to include the following proposition in our definition of Theorem 3.2:

```
forall h : Frontier, sub_model Cs V V h → geq h f.
```

Listing 3.1: Proposition for minimal model

We will explain the semantics of this proposition in more detail in Chapter 4.

3.2 Modeling finite sets in Coq

Sets in mathematics are seemingly simple structures; a set is a collection of elements. A 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 naive definition of a set, not

taking into account the complexity of this subtle notion, different set theories, powerful axioms, and so on.

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 whether or not 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. The inductive definition of a list from Coq’s standard library is as follows [15]:

```
Inductive list (A : Type) : list A :=
| nil : list A
| cons : A → list A → list A.
```

Listing 3.2: Inductive definition of list type 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 take into account the two important properties of *no duplicates* and *no order* into our definition of lists. In Coq, there are several ways to do this.

3.2.1 List & ListSet

As stated previously, Coq gives us a traditional definition of a list in the **List** module¹ of the standard library [17]. 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 asserts whether a

¹The definition of `list` is actually located in the module `Init.Datatypes`, while everything else relating to lists is in the **List** module.

list has duplicates or not. `nodup` is a function that takes in a list and returns a list with the same elements, but without duplicates. In other words, a list for which `NoDup` holds. These two can be used to better represent finite sets as lists, since we gain additional information about whether the list has duplicates or not. Coq does not however inherently understand how to compare elements when checking a list for duplicates in `nodup`. Hence we have to provide a proof that the equality of the underlying type of the list is decidable. 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 3.3: Decidability proof for string equality in Coq

Proofs as in Listing 3.3 are often given for the standard types in Coq such as `nat`, `bool` and `string`. As such, they can just be passed 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.

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` [18]. 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. As with `nodup`, these functions all require a proof of decidability of equality for the underlying type of the set. One thing to note is that all these functions use `bool` instead of `Prop`, and all require a decidability proof, which make the functions themselves decidable.

The module also gives us some lemmas to transform the boolean (type `bool`) set-operation functions into propositions (type `Prop`), and vice versa. An example to illustrate this is the following lemma on `set_mem`:

```

Lemma set_mem_correct1 {A : Type} (dec : forall x y : A, {x = y} + {x <> y}) :
  forall (x : A) (l : set A), set_mem dec x l = true → set_In x l.
Proof.
  (* ... *)
Qed.

```

Listing 3.4: `set_mem` lemma in `ListSet` module

`set_In` is just an alias for `In` from the **List** module, which is a proposition that is very common in many lemmas from the standard library. Lemmas such as the example above are very useful when reasoning about boolean functions such as `set_mem` in proofs, as transforming them into propositions makes them easier to work with and often enables us to use existing lemmas from the standard library.

Many of these boolean 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:

```

Lemma set_union_l_nil {A : Type} (dec : forall x y : A, {x = y} + {x <> y}) :
  forall l : set A, set_union dec l [] = l.
Proof.
  destruct l; reflexivity.
Qed.

```

Listing 3.5: Easy proof of lemma in `ListSet` module

The downside is that even easy and seemingly trivial proofs that reason about the other argument are frustratingly hard (or impossible) to prove, for example:

```

Lemma set_union_nil_l {A : Type} (dec : forall x y : A, {x = y} + {x <> y}) :
  forall l : set A, set_union dec [] l = l.
Proof.
  (* ... *)
Qed.

```

Listing 3.6: Impossible proof of lemma in `ListSet` module

What makes this proof impossible is that the order of elements in `set_union dec [] l` is not the same as in `l` (due to how `set_union` is implemented), and since equality on lists care about order, we cannot prove this lemma. However, there are ways to circumvent this 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. One example of such a "trick" is this definition of a strict subset relation on lists:

```
Definition strict_subset {A : Type} (s1 s2 : set A) :=
  incl s1 s2 ∧ ¬(incl s2 s1).
```

Listing 3.7: Strict subset relation on lists using `incl`

`incl` in this case is list inclusion, i.e. every element of `s1` is also in `s2`. Instead of defining a strict subset as a list that is a subset of another list and where the lists are not equal, we instead define it as a list that includes every element of another list and where some element of the other list is not in the first list. If we use these kinds of tricks and construct our proofs carefully, **ListSet** is a viable implementation for our purposes. There might however be cases where the order of the elements in the lists come into play (e.g. such as in Listing 3.6), 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, which is the case in our implementation; we need sets of atoms, clauses, strings, etc.

3.2.2 MSetWeakList

The Coq standard library also gives us another implementation of sets, **MSetWeakList** [19]. 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 satisfy some basic properties about the type we want to create a set of, namely an equality relation, decidability of this relation and the fact that this relation is an equivalence relation. The "output" from the functor is a new module containing functions and lemmas about set operations, with our input type being the type of the elements of the set. An example definition for a module of a set of atoms would be as follows:

```

Module AtomEq : DecidableType.

  (* what is our base type? *)
  Definition t := Atom.

  Definition eq := (* ... *)

  Lemma eq_equiv :
    Equivalence eq.
  Proof.
    (* proof that eq is an equivalence relation *)
  Qed.

  Lemma eq_dec :
    forall x y : t, {eq x y} + {~ eq x y}.
  Proof.
    (* proof that eq is decidable *)
  Qed.

End AtomEq.

(* apply functor Make to module AtomEq *)
Module AtomSet := Make AtomEq.

```

Listing 3.8: Set of atoms in `MSetWeakList` module

The proofs of both lemmas above are trivial, but require a substantial amount of boilerplate code. For every type we want to use as an element in a set, we have to go through this entire process. In **List** and **ListSet**, we just had to pass in the proof of the decidability of 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, given limited knowledge of Coq.

3.2.3 Ensembles

Yet another implementation of sets is given by the **Ensembles** module which defines the structure of a set as inductive propositions [16]. These inductive propositions assert

some fact about if an element is in the set or not. An example that illustrates this is the definition of `Union`:

```
Inductive Union (B C:Ensemble) : Ensemble :=
| Union_introl : forall x:U, In B x → In (Union B C) x
| Union_intror : forall x:U, In C x → In (Union B C) x.
```

Listing 3.9: `Union` in **Ensembles** module

We see that the first constructor of `Union` asserts that if an element is in `B`, then it is in the union of `B` and `C`. **Ensembles** 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.

3.3 Choice of implementation of sets

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**.

Chapter 4

Implementation of Logical Notions

4.1 Data types

As discussed in Chapter 2, we want to represent clauses as a set of atoms as premises and a single atom as a conclusion. We implement this in Coq using two types, `Atom` and `Clause`.

```
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: `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 definitionally equal 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 represent functions of the form $f : V \rightarrow \mathbb{N}^\infty$, which as we saw in Section 2.2 are functions that are models of a set of clauses. We implement this in Coq using two types, `Ninfty` and `Frontier`. `Ninfty` has two constructors; `infty` represents ∞ , and `fin n` represents a natural number n . `Frontier` is a function from a string (variable) to `Ninfty`. In the context of our formalization, we will refer to a function of this type as a *frontier*.

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

Definition Frontier := string → Ninfty.
```

Listing 4.2: `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
    (* se explantation for ≤ ? below *)
    | fin n => k ≤ ? n
    end
  end.

Definition clause_true (c : Clause) (f : Frontier) : bool :=
  match c with
  | (conds ~> conc) =>
    (* every atom in conds is true *)
    if fold_right andb true (map (fun a => atom_true a f) conds)
    then (atom_true conc f)
    else true
  end.
```

Listing 4.3: `atom_true` and `clause_true` in Coq

The infix function $\leq ?$ is the boolean (and hence 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 $n : \text{nat}$.

```

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: `shift_atom` and `shift_clause` in Coq

Using these definitions, we can now define an important property that is possible by Lemma 3.1 [1, p. 3], since this lemma enables us to check whether or not a clause is satisfied by a frontier for any shift of $k : \text{nat}$. We will use this property later to determine if a frontier is a valid model for a set of clauses.

```

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: `all_shifts_true` in Coq

4.2 Semantic functions and predicates

4.2.1 The function `sub_model`

Given any set of clauses and a frontier (function assigning values to the variables), we can determine if the frontier is a model of the set of clauses, i.e. whether all shifts of all clauses are satisfied by the frontier.

We translate this property to Coq as the recursive function `sub_model`. As we saw in Section 2.2, we distinguish between three different sets of clauses when talking about frontiers as models; $\overline{S_C}$, $\overline{S_C} \mid W$ and $\overline{S_C} \downarrow W$. These are represented by `sub_model Cs V V f`, `sub_model Cs W W f` and `sub_model Cs V W f`, respectively.

We have two additional arguments `V` and `W`: `V` is the the set of variables (strings) from the set of clauses `Cs`, and `W` is a subset of `V` such that for each variable there is a clause in `Cs` that can be used to generate a new atom. The function `vars_set_atom` used below 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 =>
    (* conclusion not in W *)
    (negb (set_mem string_dec x W) ||
    (* some premise not in V *)
    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: The function `sub_model` in Coq

4.2.2 The functions `sub_vars_improvable` and `sub_forward`

The three conditions of the disjunction in Listing 4.6 are what determines if a clause is satisfied by a frontier. If these conditions are not met, we want to repeatedly increment the value of the variable, and check again if the clause is satisfied. We start by first determining such "improvable" variables with the function `sub_vars_improvable`:

```

Fixpoint sub_vars_improvable (Cs : set Clause) (V W : set string) (f : Frontier)
  : set string :=
match Cs with
| []      => []
| (l ~> (x & k)) :: t =>
  if 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
  then sub_vars_improvable t V W f
  else set_add string_dec x (sub_vars_improvable t V W f)
end.

```

Listing 4.7: The function `sub_vars_improvable`

We then define a new function `sub_forward` that take these improvable variables and a frontier `f`, and defines a new frontier `f'` that increments the value that `f` assigns to each variable. The return type of `sub_forward` is a pair of a set of strings and a frontier, where the set of strings is the set of improvable variables.

```

Definition sub_forward (Cs : set Clause) (V W : set string) (f : Frontier)
  : set string * Frontier :=
let X := sub_vars_improvable Cs V W f in
  (* Sinfy increases a Fin value by 1, and leaves Infy unchanged *)
  let f' := fun v => if set_mem string_dec v X then Sinfy (f v) else f v
  in (X, f').

```

Listing 4.8: The function `sub_forward`

4.2.3 The function `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 `Nifty`, 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 4.9: Pointwise comparing frontiers with `geq` in Coq

4.2.4 The predicate `ex_lfp_geq`

We can now combine `sub_model` and `geq` to construct a predicate 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_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 4.10: `ex_lfp_geq` in Coq, using both `Prop` and `Set`

One thing to note here is that we can define this predicate either with `Prop` or with `Set`. When defined with `Prop`, we define it as a logical predicate, using standard first-order logic syntax. When defined with `Set`, we define it as a type, using the `sig` type constructor in place of `exists`.

fill inn some dep. type theory from Per-Martin article, also tie in to chap. 1

The reason for defining `ex_lfp_geq` with `Set`, is that we can then use Coq’s extraction feature to generate Haskell code from the Coq definitions. As briefly mentioned in Section 1.2.1, a proof can often be used to compute something. In this case, we want to compute the actual frontier `g` that satisfies the above predicate. Coq distinguishes between logical objects (objects in `Prop`) and informative objects (objects in `Set`) [9, p. 1-2]. When extracting, Coq will remove as many logical objects as possible, meaning a proposition defined in `Prop` would simply be collapsed to `()` (the unit type) when extracted to Haskell [8, p. 8]. Logical objects are only used to ensure correctness when constructing a proof in Coq, and are not needed when actually computing something using the extracted code. Informative objects, on the other hand, are kept when extracting, and can be used to compute something. This is why we define `ex_lfp_geq` with `Set`.

Another important thing to we need to keep in mind when doing proofs with extraction in mind is to avoid *universe inconsistencies*. We will not elaborate much on what type universes are, but we will give a brief explanation of how they work in Coq. In Coq, all objects of type `Prop` live in the universe `Set`, and all objects of type `Set` live in the universe `Type`. The universe hierarchy extends infinitely past `Type`, but we will not need to go further than that. When defining a type, we can only refer to objects in universes lower than the universe of the type we are defining. Here is an example to illustrate this:

```
(* type signature of sig *)
sig : forall A : Type, (A → Prop) → Type

(* Prop ≤ Set, ok *)
Definition invalid_def (A : Prop) : Set :=
  sig (fun x : A ⇒ x = x).

(* Set ≤ Prop, universe inconsistency! *)
Fail Definition invalid_def (A : Set) : Prop :=
  sig (fun x : A ⇒ x = x).
```

Listing 4.11: A universe inconsistency in Coq

4.3 The main proofs

We have now laid the groundwork for the formalization of Theorem 3.2. We precede the definition of Theorem 3.2 with two additional definitions, which helps us simplify its definition and the proof of the theorem itself.

4.3.1 The predicate `pre_thm`

Since our formal definitions of Lemma 3.3, which will be expanded on shortly, and Theorem 3.2 share some structure, we define (with `Set`) a predicate `pre_thm`:

```
Definition pre_thm (n m : nat) (Cs : set Clause) (V W : set string) (f : Frontier)
  : Set :=
  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 4.12: `pre_thm` in Coq

4.3.2 Lemma 3.3

Lemma 3.3 is used in the proof of Theorem 3.2 to solve a special case, as we saw in Section 2.2. We also need this lemma to be able to prove Theorem 3.2 in Coq. As explained in Section 3.1.2, we will only give a formal definition of the lemma in Coq, leave out the proof. By using the Coq command `Admitted`, we can leave out a full proof of a lemma, but still be able to use the lemma in other Coq proofs. We define Lemma 3.3 in Coq 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.
Admitted.

```

Listing 4.13: Lemma 3.3 in Coq

One thing to note is that when applying Lemma 3.3 in the formal proof of Theorem 3.2, the first assumption of the lemma is solved immediately by applying the primary induction hypothesis (the induction on n , also called IH_n in Coq).

4.3.3 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 4.14: Theorem 3.2 in Coq

The proof of Theorem 3.2 is based on a primary induction on n and a secondary induction on m .

Base case of n

The first base case is simple. We want to prove

$$(1) \text{ ex_lfp_geq } Cs \text{ (nodup string_dec } V) \text{ (nodup string_dec } V) f.$$

Since $n = 0$, we get that the length of V is 0, and hence we get a new goal $\text{ex_lfp_geq } Cs [] [] f$.

This is proven by the lemma `ex_lfp_geq_empty`, which states that $\text{forall } Cs f, \text{ex_lfp_geq } Cs [] [] f$.

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

$$\text{ex_lfp_geq } Cs \text{ (nodup string_dec } V) \text{ (nodup string_dec } V) f.$$

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

$$\text{forall } Cs \text{ } V \text{ } W \text{ } f, \text{incl } V \text{ } W \rightarrow \text{forall } f, \text{ex_lfp_geq } Cs \text{ } W \text{ } W \text{ } f \rightarrow \text{ex_lfp_geq } Cs \text{ } V \text{ } V \text{ } f.$$

We give this lemma the arguments of Cs , `nodup string_dec V` and `nodup string_dec W`.

This generates two new goals:

$$(1) \text{ incl (nodup string_dec } V) \text{ (nodup string_dec } W)$$

$$(2) \text{ ex_lfp_geq } Cs \text{ (nodup string_dec } W) \text{ (nodup string_dec } W) f$$

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

$$\text{Datatypes.length (set_diff string_dec (nodup string_dec } V) \text{ (nodup string_dec } W))} \leq m \leq n.$$

Since $m = 0$, 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

$$\text{forall dec } V \text{ } W, \text{set_diff dec } V \text{ } W = [] \leftrightarrow \text{incl } V \text{ } W.$$

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

The inductive case of m by far the largest part of the proof (see lines 17-199 of Listing A.1). As it is very complex and does not follow the paper too closely, we will go through a more "high-level" explanation of the reasoning we followed when constructing the formalized proof. We will sometimes use mathematical notation to explain the proof instead of Coq code, for the sake of being terse and clear.

Assume $W \subseteq V$, $|V| \leq n + 1$, $|V - W| \leq m + 1 \leq n + 1$. If $|V - W| \leq m$ then we can apply the secondary induction hypothesis, and we are done. Hence, we can assume $|V - W| = m + 1$, meaning $|W| \leq n$. To prove $\text{ex_lfp_geq } \text{Cs } V \text{ } V \text{ } f$, we assert $(W, g) = \text{sub_forward } \text{Cs } V \text{ } V \text{ } f$. This gives us three cases to consider:

1. $W = \emptyset$, and we are done.
2. $W = V$, meaning all variables in V map to ∞ , and we are done.
3. $\emptyset \subset W \subset V$. Then $|W| \leq n$ and $|V - W| \leq m$. So by the primary induction hypothesis, we get $\text{ex_lfp_geq } \text{Cs } W \text{ } W \text{ } f$. Then by the second induction hypothesis, we get $\text{ex_lfp_geq } \text{Cs } V \text{ } V \text{ } f$, and we are done.

4.4 Extraction to Haskell

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

```
Extraction Language Haskell.

Extract Constant map  $\Rightarrow$  "Prelude.map".
Extract Constant fold_right  $\Rightarrow$  "Prelude.foldr".

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

Listing 4.15: 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.

By default, Coq will give its own implementation of any functions used, instead of using Haskell's native implementations. If we want, we can specify what native Haskell functions should be used when extracting a Coq function. In the example code above, we specify that when extracting, Haskell's `Prelude.map` and `Prelude.foldr` should be used for the Coq functions `map` and `fold_right`.

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

5.1 Examples using the extracted Haskell code

5.1.1 Defining examples for extraction in Coq

It is easiest to define as much of the example as possible in Coq and then extract it to Haskell, since Coq heavily prioritizes code correctness over readability when extracting. This makes much of the Haskell code hard to read due to unconventional syntax, e.g. using a recursion operator insted of calling a function recursively. First, we can look at the type of `thm_32` in Coq to see what arguments we need to give it. We do this by using the Coq command `Check`.

```
thm_32
: forall (n m : nat) (Cs : set Clause) (V W : set string) (f : Frontier),
  pre_thm n m Cs V W f
```

Listing 5.1: Type signature of `thm_32` in Coq

We see that the type of `thm_32` is a proof of `pre_thm`, with the quantified variables `n`, `m`, `Cs`, `V`, `W`, `f`. We can now look at the type signature of `pre_thm`.

```
pre_thm
: nat →
  nat → set Clause → set string → set string → Frontier → Set
```

Listing 5.2: Type signature of `pre_thm` in Coq

We see that `pre_thm` returns an object in `Set`. This object is the proposition defined in Listing 4.12, where all the variables are the arguments given to `pre_thm`, i.e. the variables quantified in `thm_32`. This proposition states some logical assumptions, and concludes with a definition of `ex_lfp_geq`. By the notion of propositions as types, this proposition is a function that transforms a proof of the assumptions into a proof of the conclusion. Since the conclusion is a definition of `ex_lfp_geq` instantiated with some variables, we get a proof of `ex_lfp_geq` for these same variables. Looking at the definition of `ex_lfp_geq`, it defines a proposition stating that there exists a `g : Frontier` such that the proposition holds. If such a `g` exists, then the `g` itself is evidence (proof) that the proposition holds. Hence, a proof of `ex_lfp_geq` is simply a `Frontier` that satisfies the conditions in `ex_lfp_geq`.

With all this information, we can now define an example of `thm_32` in Coq.

```
(* represents the clauses: a → b + 1, b + 1 → c + 2 *)
Example Cs := [
  ["a" & 0] ~> "b" & 1;
  ["b" & 1] ~> "c" & 2
].

(* all-zero frontier *)
Example thesis_ex_1_f := frontier_fin_0.

(* extract only variables from clauses *)
Example vars' := nodup string_dec (vars Cs).

(* (partially) apply arguments to thm_32 *)
Example thesis_ex_1 :=
  thm_32
  (Datatypes.length vars')
  (Datatypes.length vars')
  Cs
  vars'
  []
  thesis_ex_1_f.
```

Listing 5.3: `thm_32` example

Since Coq eliminates many of the logical parts of the proof when extracting, [8, p. 8], we can avoid the tedious task of proving all the assumptions of `pre_thm` by simply using

the extracted Haskell function for `thesis_ex_1`. The final assumption of `pre_thm`, namely `ex_lfp_geq Cs (nodup string_dec W) (nodup string_dec W) f`, will however not be removed by Coq, since it is an object in `Set`. This assumption is trivially true for any `f`, since `W` is an empty list in our example. Since Coq simplifies the type of `ex_lfp_geq` to `Frontier`, we can just include the same frontier `f` as in `thesis_ex_1` to the extracted Haskell version of `thesis_ex_1`, as a replacement for a proof of the above trivially true assumption. We then receive a `Frontier` as output, which is the result of the theorem; the frontier that is a model of a set of clauses `Cs`. This frontier can then be applied to any string representing a variable to get the value of that variable in the model, which should be either a natural number or infinity.

5.1.2 Necessary alterations to the extracted Haskell code

Since we have not given a proof of Lemma 3.3, Coq will include a "placeholder" definition of the lemma in the extracted code. This definition will simply call the `error` function, which will crash the program when called. We circumvent this by replacing the extracted definition of `lem_33` with the identity function for any frontier. We will look at some examples where this workaround is not sufficient in Section 5.3.

If we want to actually read the output from the extracted functions, we also need to derive a `Show` instance for `Ninfty`. What this means is that we need to define a function `show :: Ninfty → String`, which will be used by Haskell to convert a `Ninfty` to a `String`. This can be done by simply adding the line `deriving Prelude.Show` to the definition of `Ninfty`, which will make Haskell just print the constructors of `Ninfty`, which will be either `Fin n` for some natural number `n`, or `Infy` for infinity.

5.1.3 Example output

We can now run the example from Listing 5.3 using `GHCi`, which is an interactive Haskell interpreter that is included with `GHC`, the standard Haskell compiler.


```

ghci> (thesis_ex_1 thesis_ex_1_f) "a"
Fin 0
ghci> (thesis_ex_1 thesis_ex_1_f) "b"
Fin 1
ghci> (thesis_ex_1 thesis_ex_1_f) "c"
Fin 2
ghci> (thesis_ex_1 thesis_ex_1_f) "x"
Fin 0

```

Listing 5.4: `thm_32` example output

The expression `(thesis_ex_1 thesis_ex_1_f)` produces a function that is the minimal model of `Cs`. When given a string value (variable) from the set of clauses, the function will compute the value of that variable. When given any other variable, the function will return the value that the original frontier given as input would return for that variable, which is always `Fin 0` in this case (since in our example the frontier is `frontier_fin_0`, which is a constant function that always returns `Fin 0`).

5.2 Real world example

As stated previously, the algorithm described Theorem 3.2 is being tested for use in checking loops and determining type universe levels in the type system of Coq. We briefly explained type universes in the case of Coq in Section 4.2.4. The error that occurred in the example of a universe inconsistency in Listing 4.11 was detected by Coq's own universe-constraint checking algorithm, but we can also use our own algorithm to do the same thing. *is this true?*

Using the Coq command `Print Universes` we can see the current state of the universe hierarchy for any Coq file. Doing this at the bottom of our file `Main.v`, after our proof of Theorem 3.2, we get the following output (truncated):

```

Set < Coq.setoid_ring.BinList.1
  < Coq.Numbers.BinNums.1

...

≤ Coq.Lists.List.247
≤ Coq.Lists.List.886

```

Listing 5.5: Universe hierarchy in `Main.v`

This output has over 5000 lines, so we have truncated it to only show the first and last few lines. We can translate these constraints into a set of clauses `Cs` as such:

```

Definition Cs := [
  ["Set" & 0] ~> "Coq.setoid_ring.BinList.1" & 1;
  ["Coq.setoid_ring.BinList.1" & 1] ~> "Coq.Numbers.BinNums.1" & 2;
  ...
  ["Coq.Lists.List.247" & 247] ~> "Coq.Lists.List.886" & 886
].

```

Listing 5.6: Universe hierarchy as clauses

We can then run our algorithm on these clauses to get a minimal model of `Cs`. This minimal model is a function that assigns to each type universe a natural number, which corresponds to its universe level. It is minimal in the sense that no other mapping from type universes to universe levels that satisfies the constraints in `Cs` has a lower universe level for any given type universe. Even though we have not given a formal proof of Lemma 3.3, this "real world" example still gives a correct minimal model for the type universe constraints from `Coq`.

5.3 Extension of Listing 5.3 that fails

There are some cases where the extracted Haskell code will crash. To show this, we can extend the set of clauses from Listing 5.3 as such:

```

Example Cs := [
  ["a" & 0] ~> "b" & 1;
  ["b" & 1] ~> "c" & 2;
  (* add new clause *)
  ["c" & 2] ~> "d" & 3
].

Example thesis_ex_2_f := frontier_fin_0.
Example vars' := nodup string_dec (vars Cs).

Example thesis_ex_2 :=
  thm_32
  (Datatypes.length vars')
  (Datatypes.length vars')
  Cs
  vars'
  []
  thesis_ex_2_f.

```

Listing 5.7: `thm_32` example extended

If we try to run the extracted Haskell code for this example, we will always encounter a runtime exception:

```

ghci> (thesis_ex_2 thesis_ex_2_f) "a"
*** Exception: absurd case
CallStack (from HasCallStack):
  error, called at ex.hs:13:3 in main:Ex

```

Listing 5.8: Output of `thm_32` example extended

The exception "absurd case" is what happens when a logical assumption in the proof is not correct, which is the case here. To see why this happens, we can do some minor alterations to the extracted Haskell code, and print out the values of some variables during the execution of the algorithm. What we want to show is every primary and secondary induction call (`IHn` and `IHm` respectively), and the values of the variables `n`, `m`, `V` and `W` at the time of those calls. Doing so, we get the following output (truncated for brevity):

```

ghci> (thesis_ex_2 thesis_ex_2_f) "a"
IHn: n = 3, m = 4, V = ["a","b","c","d"], W = []

...

IHn: n = 0, m = 1, V = ["a","b","c","d"], W = []
IHm: n = 1, m = 0, V = [], W = []
IHm: n = 3, m = 0, V = ["b","c","d"], W = ["d"]
*** Exception: absurd case
CallStack (from HasCallStack):
  error, called at ex.hs:13:3 in main:Ex

```

Listing 5.9: Output of `thm_32` example extended, with debug

The debug line we are most interested in is the last one, which is the last secondary induction call before the exception. If we refer to lines 133-137 of Listing A.1, we see that the proof asserts (and proves) the following claim:

```

Datatypes.length
  (set_diff string_dec V (set_union string_dec (nodup string_dec W) U)) <
Datatypes.length
  (set_diff string_dec V (nodup string_dec W)) ≤ S m.

```

Listing 5.10: False claim in `thm_32` proof

This claim is false, as we can see by looking at the values of `V`, `W` and `m` at the time of the last induction call. We replace the variables in the claim with their values, and reduce (parts of) the claim to the following:

```

Datatypes.length (set_diff string_dec V (nodup string_dec W)) ≤ S m
⇒ Datatypes.length (set_diff string_dec ["b","c","d"] ["d"]) ≤ S 0.
⇒ Datatypes.length ["c","b"] ≤ S 0.
⇒ 2 ≤ 1.

```

The final claim is obviously not true, hence why the Haskell code crashes with "absurd case". But why does this happen at all? Shouldn't the Coq proof have failed at the point

where the claim was asserted? This error is due to the fact that we have not given a formal proof of Lemma 3.3. Since we do not use Lemma 3.3 to calculate a new frontier in the algorithm but instead just redefine Lemma 3.3 as the identity function, this has consequences for later steps in the algorithm. This example in particular is "constructed to fail"; as we saw in the previous section, the algorithm does work for many cases even without a formal proof of Lemma 3.3.

Chapter 6

Evaluation

6.1 Correctness of informal proof

As we have shown in our formalization and by the results of the extraction, the informal proof of Theorem 3.2 from [1] appears to be correct. As mentioned in Section 3.1, we have not fully formalized nor verified the entirety of the informal proofs, and as such we cannot be 100% sure that the informal proof of Theorem 3.2 is correct. However, we have verified the most important parts of Theorem 3.2 without encountering any problems, and we can thus be very confident that the informal proof is correct.

6.2 Worthwhileness of formalization

The formalization of the proof of Theorem 3.2 from [1] took us about 10 months of work. This includes the time spent on learning Coq and the time spent on the formalization of the proof. We believe that the formalization of the proof of Theorem 3.2 was worth the effort, as it has both given us a better understanding of the proof, and a formal proof that we can be very confident is correct.

6.3 Usefulness of extracted algorithm

As seen in Section 5.2, the extracted algorithm is useful in practice. However, the efficiency is not very good (is it? should check). There are also cases where the extracted algorithm crashes, as seen in Section 5.3. However, it should not be a surprise that not every case is handled by the algorithm, as we have not given a formal proof of Lemma 3.3. It is actually quite surprising that the extracted algorithm works as well as it does (when given "real world" input, that is).

Chapter 7

Related & Future Work

7.1 Related work: universe consistency checking in Coq

There is an ongoing effort [11] to implement a version of the algorithm from [1] for use in universe consistency checking in Coq. This version of the algorithm is graph-based, does not include joins, and only support constraints (clauses) of the form $x + k \geq y$ with $k \in \{0, 1\}$. The main focus is on speed, which explains the limited expressivity. There are plans to extend the current implementation with more expressivity later on.

7.2 Future work

7.2.1 Completing proof of remaining logical lemmas

As mentioned in Section 3.1.1, we have not fully completed some of the intermediate lemmas used directly in the proof of Theorem 3.2. We have done some work on making sure these are purely logical steps, and that they should have no effect on the final result. Solving these fully would however strengthen the argument for the correctness of the formal, and hence informal, proof of Theorem 3.2.

7.2.2 Formal proof of Lemma 3.3

As mentioned in Section 3.1.2, we have not given a full formal proof of Lemma 3.3. As we also saw in Section 5.3, constructing a formal proof of this lemma would fix some cases where the extracted algorithm fails with an exception. Further work on a complete formalization of Theorem 3.2 would certainly benefit from a formal proof of Lemma 3.3.

7.2.3 Proving minimality of model generated by Theorem 3.2

As mentioned in Section 3.1.3, our proof of Theorem 3.2 does not include a claim that the model generated is minimal. As later explained in the same subsection, we would only need to include one additional assumption in our proof of Theorem 3.2 to be able to correctly claim that the model generated is minimal. We chose not to do this as it had no effect on the results produced by the extracted algorithm, only the claim for correctness.

Chapter 8

Conclusion

In this thesis, we have presented a detailed exploration into formalizing the proofs from two problems in dependent type theory, and extracting a certified program from their verified formal proofs using Coq. We have investigated the utility of proof assistants and dependent type theory in providing a structured approach to proofs and formalization, which ultimately led to a practical application, in the form of a certified correct program.

We started by introducing relevant theoretical concepts, including propositions as types and dependent types. We then used these to explore the Coq proof assistant, and compared it to other available tools.

The case presented in Chapter 2 provided an opportunity to utilize these theoretical concepts in a practical setting. Simplifications were made to fit the time constraints of the project. Since reasoning about sets were a sizeable part of the theorem we formalized, a good implementation of sets was important. Various implementations were considered, and we made a choice that best fit our needs. We then presented the "building blocks" of our formalization, in the form of various data types, functions, predicates, lemmas and theorems.

The practical relevance of the formalization effort was demonstrated through the extraction of our Coq proofs into a certified Haskell program. We gave an example of how the program could be run to solve a simple case, but also discussed the limitations of the algorithm. We also gave a more sizable "real-world" example, which again demonstrated the practicality of our work.

We then assessed how through our formalization effort, we gained more confidence in the correctness of the original informal proofs posed in [1]. Furthermore, the extracted

algorithm was evaluated for its usefulness, highlighting the practical benefits of our formalization. We then explained how our work could be extended in the future, namely by completing the proofs of remaining logical lemmas, providing a formal proof of Lemma 3.3, and proving the minimality of the model generated by Theorem 3.2.

In conclusion, we have in this thesis successfully presented a case study in dependent type theory, where we formalized an informal proof using the Coq proof assistant, and extracted a useful certified program from this formal proof.

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] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda – a functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 73–78, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-03359-9.
- [3] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (system description). In *25th International Conference on Automated Deduction (CADE-25), Berlin, Germany*, 2015.
URL: <https://leanprover.github.io/papers/system.pdf>.
- [4] D. Delahaye. A Tactic Language for the System Coq. In *Proceedings of Logic for Programming and Automated Reasoning (LPAR), Reunion Island*, volume 1955 of *Lecture Notes in Computer Science*, pages 85–95. Springer-Verlag, November 2000.
URL: [https://www.lirmm.fr/%7Edelahaye/papers/ltac%20\(LPAR%2700\).pdf](https://www.lirmm.fr/%7Edelahaye/papers/ltac%20(LPAR%2700).pdf).
- [5] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. The SSREFLECT proof language.
URL: <https://coq.inria.fr/refman/proof-engine/ssreflect-proof-language.html>. Accessed: 2023-03-21.
- [6] Thomas C. Hales. Formal Proof. *Notices of the American Mathematical Society*, 55 (11):1370, 2008.
URL: <https://www.ams.org/notices/200811/200811FullIssue.pdf>.

- [7] 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.
- [8] P. Letouzey. Coq Extraction, an Overview. In A. Beckman, C. Dimitracopoulos, and B. Löwe, editors, *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*, volume 5028 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008.
URL: https://www.irif.fr/~letouzey/download/letouzey_extr_cie08.pdf. Accessed: 2023-05-09.
- [9] Pierre Letouzey. A New Extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002*, volume 2646 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.
URL: <https://hal.science/hal-00150914/document>. Accessed: 2023-05-09.
- [10] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hrițcu, 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.
- [11] Matthieu Sozeau. Universes loop checking with clauses.
URL: <https://github.com/coq/coq/pull/16022>. Accessed: 2023-05-01.
- [12] The Coq Team. Calculus of Inductive Constructions, .
URL: <https://coq.github.io/doc/v8.9/refman/language/cic.html#calculusofinductiveconstructions>. Accessed: 2023-05-01.
- [13] The Coq Team. CoqIDE, .
URL: <https://coq.inria.fr/refman/practical-tools/coqide.html>. Accessed: 2023-03-21.
- [14] The Coq Team. A short introduction to Coq, .
URL: <https://coq.inria.fr/a-short-introduction-to-coq>. Accessed: 2023-01-18.
- [15] The Coq Team. Library Coq.Init.Datatypes, .
URL: <https://coq.inria.fr/library/Coq.Init.Datatypes.html>. Accessed: 2023-05-11.
- [16] The Coq Team. Library Coq.Sets.Ensembles, .
URL: <https://coq.inria.fr/library/Coq.Sets.Ensembles.html>. Accessed: 2023-05-11.

- [17] The Coq Team. Library Coq.Lists.ListSet, .
URL: <https://coq.inria.fr/library/Coq.Lists.ListSet.html>. Accessed: 2023-05-11.
- [18] The Coq Team. Library Coq.Lists.List, .
URL: <https://coq.inria.fr/library/Coq.Lists.List.html>. Accessed: 2023-05-11.
- [19] The Coq Team. Library Coq.MSets.MSetWeakList, .
URL: <https://coq.inria.fr/library/Coq.MSets.MSetWeakList.html>. Accessed: 2023-05-11.
- [20] Philip Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, nov 2015. ISSN 0001-0782. doi: 10.1145/2699407.
URL: <https://doi.org/10.1145/2699407>. Accessed: 2023-05-01.

Appendix A

Full proof of Theorem 3.2 in Coq

```
1 Theorem thm_32 :
2   forall n m : nat,
3   forall Cs : set Clause,
4   forall V W : set string,
5   forall f : Frontier,
6     pre_thm n m Cs V W f.
7 Proof.
8   unfold pre_thm. induction n as [|n IHn].
9   - intros. apply le_0_r in H0.
10    apply length_zero_iff_nil in H0.
11    rewrite H0. apply ex_lfp_geq_empty.
12  - induction m as [|m IHm]; intros.
13    + apply (ex_lfp_geq_incl Cs (nodup string_dec V) (nodup string_dec W));
14      try assumption. destruct H1. apply le_0_r in H1.
15      apply length_zero_iff_nil in H1.
16      apply set_diff_nil_incl in H1. assumption.
17    + inversion H1.
18      apply le_lt_eq_dec in H3. destruct H3.
19      * apply (IHm Cs V W f); try assumption. lia.
20      * apply (ex_lfp_geq_nodup_iff) in H2.
21        assert (Datatypes.length (nodup string_dec W) ≤ n).
22        {
23          eapply (set_diff_succ string_dec) in H; try apply H3.
24          apply succ_le_mono. eapply le_trans.
25          apply H.
26          - eapply le_trans in H0. apply H0. apply le_refl.
27          - apply e.
28        }
```

```

29   assert (ex_lfp_geq Cs (nodup string_dec W) (nodup string_dec W) f).
30   {
31     apply (IHn n Cs W [] f);
32     try assumption; try apply incl_nil_1.
33     — eapply (set_diff_succ string_dec) in H; try apply H3.
34       simpl. rewrite set_diff_nil. apply conj; lia. apply e.
35     — unfold ex_lfp_geq. exists f. split.
36       apply geq_refl. apply sub_model_W_empty.
37   }
38   assert (H': incl W V) by assumption.
39   apply (nodup_incl2 string_dec) in H.
40   apply (lem_33 Cs V (nodup string_dec W) f) in H.
41   elim H. intros h [H8 H9].
42   destruct
43     (sub_forward
44       Cs
45       (nodup string_dec V)
46       (nodup string_dec V)
47       h) as [U h'] eqn:Hforward.
48   assert
49     (sub_forward
50       Cs
51       (nodup string_dec V)
52       (nodup string_dec V)
53       h = (U, h')) by assumption.
54   assert
55     (sub_forward
56       Cs
57       (nodup string_dec V)
58       (nodup string_dec V)
59       h = (U, h')) by assumption.
60   rewrite nodup_rm in H9.
61   apply
62     (sub_forward_incl_set_diff
63       Cs
64       h
65       h'
66       (nodup string_dec V)
67       (nodup string_dec W)
68       U) in H9;

```



```

69     try apply Hforward.
70     inversion Hforward. apply sub_forward_incl in Hforward.
71     destruct U as [[u U']] eqn:Hu.
72     - - apply sub_forward_empty in H7.
73         destruct H7. unfold ex_lfp_geq. exists h.
74         split; assumption.
75     - - destruct
76         (incl_dec
77         string_dec
78         V
79         (nodup
80         string_dec
81         (set_union string_dec (nodup string_dec W) U))).
82     ++ unfold ex_lfp_geq. exists (update_infty_V V f). split.
83         ** apply geq_nodup_true. apply geq_update_infty_V.
84         ** rewrite ← sub_model_nodup. apply sub_model_update_infty_V.
85     ++ assert (incl (nodup string_dec (set_union string_dec W U)) V).
86         {
87             apply nodup_incl2. eapply incl_set_union_trans.
88             assumption. apply nodup_incl in Hforward.
89             rewrite Hu. assumption.
90         }
91     assert
92         (Datatypes.length
93         (nodup string_dec (set_union string_dec W U)) <
94         Datatypes.length
95         (nodup string_dec V)).
96     {
97         assert (Datatypes.length
98         (nodup string_dec (set_union string_dec W U)) ≤
99         Datatypes.length
100         (nodup string_dec V)).
101         {
102             eapply NoDup_incl_length. apply NoDup_nodup.
103             apply nodup_incl. assumption.
104         }
105         apply le_lt_eq_dec in H13. destruct H13; try assumption.
106         assert
107             (strict_subset
108             (nodup string_dec (set_union string_dec W U))

```

```

109         (nodup string_dec V)).
110     {
111         unfold strict_subset. split.
112         — apply nodup_incl. assumption.
113         — unfold not. intros. apply n0.
114         apply nodup_incl2 in H13.
115         assert
116             (incl
117                 (nodup string_dec (set_union string_dec W U))
118                 (nodup string_dec V)).
119         {
120             apply nodup_incl. assumption.
121         }
122         rewrite (incl_set_union_nodup_l string_dec). assumption.
123     }
124     apply (strict_subset_lt_length string_dec).
125     unfold strict_subset in H13.
126     destruct H13. unfold strict_subset. split.
127     — apply nodup_incl in H13.
128         apply nodup_incl2 in H13. assumption.
129     — unfold not. intros. apply n0.
130         rewrite (incl_set_union_nodup_l string_dec).
131         apply nodup_incl. assumption.
132     }
133     assert
134         (Datatypes.length
135             (set_diff string_dec V (set_union string_dec (nodup string_dec W) U)) <
136             Datatypes.length
137             (set_diff string_dec V (nodup string_dec W)) ≤ S m).
138     {
139         apply conj.
140         — rewrite Hu. apply set_diff_incl_lt_length;
141             try assumption. apply nodup_incl2. assumption.
142             rewrite ← set_diff_nodup_l in H9. assumption.
143         — rewrite ← set_diff_nodup_r.
144             rewrite ← set_diff_nodup_eq. rewrite e. lia.
145     }
146     apply (ex_lfp_geq_monotone Cs (nodup string_dec V) h' f).
147     eapply (IHm Cs V (nodup string_dec (set_union string_dec W U)) h');
148     try assumption.

```

```

149      ** apply conj; try lia. inversion H14.
150      apply le_lt_eq_dec in H16. destruct H16;
151      rewrite nodup_rm; rewrite set_diff_nodup_eq in *;
152      rewrite ← length_set_diff_set_union_nodup_l;
153      lia.
154      ** apply (IHn n Cs (nodup string_dec (set_union string_dec W U)) [] h');
155      try apply incl_nil_l.
156      - - - assert (Datatypes.length (nodup string_dec V) ≤ Datatypes.length V).
157      {
158          apply NoDup_incl_length. apply NoDup_nodup.
159          apply nodup_incl2. apply incl_refl.
160      }
161      rewrite nodup_rm. rewrite set_diff_nodup_eq in *;
162      try rewrite set_union_nodup_l in *; lia.
163      - - - apply conj; try lia.
164      assert
165          (Datatypes.length
166              (set_diff
167                  string_dec
168                  (nodup string_dec (set_union string_dec W U))
169                  [])) ≤
170          Datatypes.length
171              (nodup string_dec (set_union string_dec W U)).
172      apply (set_diff_nil_length string_dec).
173      eapply le_trans.
174      rewrite set_diff_nodup_eq. apply H15.
175      apply lt_le_pred in H13.
176      eapply le_trans. apply H13.
177      assert
178          (Datatypes.length (nodup string_dec V) ≤
179              Datatypes.length V).
180      {
181          apply NoDup_incl_length. apply NoDup_nodup.
182          apply nodup_incl2. apply incl_refl.
183      }
184      assert
185          (pred (Datatypes.length (nodup string_dec V)) ≤
186              pred (Datatypes.length V)).
187      {
188          apply Nat.pred_le_mono. assumption.

```

```

189         }
190         apply le_n_Sm_pred_n_m in H0. assumption.
191         - - - unfold ex_lfp_geq. exists h'.
192         split. apply geq_refl.
193         apply sub_model_W_empty.
194         ** eapply geq_trans with h; try assumption.
195         rewrite ← H12.
196         apply geq_Sinfy_f2.
197         - - unfold pre_thm. intros. apply (IHn m0 Cs' V' W' f');
198             try assumption; try lia.
199         - - rewrite nodup_rm. assumption.
200 Qed.

```

Listing A.1: Full proof of Theorem 3.2 in Coq