University of Bergen
Department of Informatics

Formalizing a problem in dependent type theory and extracting a certified program from the proof of its specification

Author: Andreas Salhus Bakseter

Supervisors: Marc Bezem, Håkon Robbestad Gylterud

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

May, 2023

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer 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.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Acknowledgements

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer 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
Saturday 6th May, 2023

Contents

List of Figures

List of Tables

Listings

Chapter 1

Background

1.1 Formalizing mathematics

1.1.1 Proofs

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 [8]. Such proofs rely heavily on the reader's intuition and often omit logical steps to make them easier to understand for humans [5]. 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 [6], and even the claim justified by it 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 is reduced [5]. The only possible errors in formalized proofs are false assumptions and/or flawed verification software.

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, or we can create our own data types which allows us to represent e.g. clauses in logic.

1.2.1 Propositions as types

add source, see LateX comment

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 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, 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.

1.2.2 Dependent types

Dependent types allow us to define more rigorously types which depend on terms.

An example to illustrate this is the definition of a vector:

```
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 `vector A` (`S n`), where the value of `n` is the length of the vector given to the constructor as an argument. 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 *)
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 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. 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 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* (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 langauges OCaml and Haskell [12]. 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 [10].

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 1.3: 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_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 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 (is this correct?) [3]. 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 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 [11]. Ltac is not the only proof langauge, with another example being *SSReflect* [4].

1.3.2 Other proof assistants

Agda

Agda is a depdently typed functional programming language based on Martin-Löf's intuitionistic type theory. Unlike Coq, Agda does not use tactics. [1] However, by using proposition as types, Agda can also function as a proof assistant.

Isabelle

Lean

Lean is proof assistant, automated theorem prover and dependently typed functional programming language. Lean can be instantiated using either CIC or Martin-Löf's intuitionistic type theory. [7]

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 a variety programming languages, such as Haskell and OCaml (as is the case for Coq). [12]

(if we want to elaborate, cite this `https://www.irif.fr/~letouzey/download/extraction2002.pdf` and this `https://www.irif.fr/~letouzey/download/letouzey_extr_cie08.pdf`)

Chapter 2

The Case in Question

2.1 Overview

We have used the Coq proof assistant to formalize parts of the proofs of the following paper, Bezem and Coquand [2]. 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 for use in checking loops in Coq's type system. [9]

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.

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

A term over V has the form $x_1 + k_1 \vee \ldots \vee x_m + k_m$, where $x_i \in V$ and $k_i \in \mathbb{N}$.

A relation is an equation $s = t$, where s and t are terms over V. A constraint, like $x = y^+$ (with $x, y \in V$), expresses a relation between the generators (variables) x and y.

Horn clauses are propositional clauses $A \to b$, 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*.

For each constraint $s = t$, we generate $m + n$ clauses by replacing join by conjunction and implication. We define $S_{s=t}$ as the set of these clauses and S_C as the union of all $S_{s=t}$, with $s = t$ as a constraint in C.

Predecessor clauses are derived from the axiom $x \vee x^+ = x^+$ and have the form $x+k+1 \to x + k$, where $(x \in V, k \in \mathbb{N})$

We define closure under shifting upwards as follows: $A \to b$ is in the set of clauses, then so must $A+1 \to b+1$ 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 $\overline{S_C} \mid W$ 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 $f : V \to N^\infty$ is a model ...

2.2.1 Lemma 3.1

Lemma 3.1 states that given $f : V \to N^\infty$, and a clause $A \to b$, let P be the problem whether or not $A + k \to b + k$ is satisfied by f for all $k \in N$. Then P is decidable. [2, p. 3]

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 $k \in N$. Lemma 3.1 is also crucial for making case distinctions in further proofs, since we know that any S_C is finite.

2.2.2 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, p. 3]

2.2.3 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. [2, p. 3-4]

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. 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. Examples of such trade-offs are simplicity of the implementation, ease of use, and performance.

3.1 Simplifications

We have made some simplifications to our formalization for the sake of time.

3.1.1 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

3.1.2 Omission of formal proof of Lemma 3.3

We have included a formulation of this lemma, but not a 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; the limitations of this simplification will be explained in more detail in section 5.3.

3.1.3 Incomplete proofs for some purely logical lemmas

We have chosen to not waste too much time fully completing the proofs of some purely logical lemmas, which are mainly used as intermediate steps in the proof of Theorem 3.2. As we will se later in this section, this is mainly due to very complex or impossible-to-prove lemmas in our set implementation being trivial in informal mathematics. The proofs of these lemmas are however not very interesting, and they do not contribute to the correctnes of the implementation nor the results of extracting code from the Coq formalization.

3.2 Modeling finite sets in Coq

Sets in mathematics are seeminlgy simple structures; a set is a collection of elements. 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 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 nor 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:

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

Listing 3.2: Inductive def. 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** 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 asserts whether a 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`. 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.
```

Listing 3.4: `set_mem` lemma from `ListSet`

`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`

The downside is that even easy and seamingly 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`

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. 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. 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. 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.

3.2.2 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 an equality relation, decidability of this relation and the fact that this relation is an equivalence relation. 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.

3.2.3 Ensembles

Yet 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.

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 seen in ??, 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 model functions of the form $f : V \to N^{\infty}$. 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: `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) ⇒
    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 : 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 [2, p. 3], since this lemma enables us to check whether or not a clause is satisfied by a frontier for any shift of `k : nat`. We will use this property later to determine if a set of clauses is a valid model.

```
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 function assigning values to the variables (frontier), 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`. 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  ⇒
    (* 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 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 `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: Pointwise comparing frontiers with `geq` in Coq

4.2.3 The predicate `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_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: `ex_lfp_geq` in Coq, using both `Prop` and `Set`

One thing to note here is that we can define this lemma either as having the type `Prop` or as having the type `Set`. When using `Prop`, we define it as a proposition, using standard first-order logic syntax.

When using `Set`, 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`. explain this

The reason for defining ex_lfp_geq using Set, 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 Set to avoid universe inconsistencies when performing extraction. explain this, cite extraction papers maybe

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 (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 proposition 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

4.3.2 Lemma 3.3

Lemma 3.3 from the paper [2] 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

4.3.3 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 4.11: 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) ex_lfp_geq Cs (nodup string_dec V) (nodup string_dec V) f.

Since n $= 0$, we get that the length of V is 0, and hence we get a new goal ex_lfp_geq Cs [] [] f. This is proven by the lemma ex_lfp_geq_empty, which states that forall Cs f, 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

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 $= 0$, 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

insert brief explanation here (from proof overview)...

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 ⇒ "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.

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, `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

When extracting thm_32 to Haskell, Coq creates a Haskell function that takes as input every variable that is used in the definition of thm_32. This is what the type signature of such a function looks like in Haskell:

```
thm_32 :: Prelude.Integer → Prelude.Integer → (Set Clause0) →
          (Set Prelude.String) → (Set Prelude.String) → Frontier →
          Ex_lfp_geq → Ex_lfp_geq
thm_32 n m cs v w f x =
   {- ... -}
```

Listing 5.1: Haskell extraction of thm_32

This all looks familiar: n, m are two natural numbers which are used for induction in the proof of the theorem, cs is the set of clauses, v, w are the set of variables and f is the frontier. We also see an additional argument of type Ex_lfp_geq, and that the return type of the function also has this type. In the extraction, Ex_lfp_geq has the following definition:

```
type Ex_lfp_geq_S = Frontier
type Ex_lfp_geq = Ex_lfp_geq_S
```

Listing 5.2: `Ex_lfp_geq` in Haskell

Looking back at the Coq definition of `ex_lfp_geq`, it defined a proposition that stated that there exists a `g : Frontier`, such that the proposition holds. If such a `g` exists, then the `g` itsself is evidence (proof) that the proposition holds. Thus, the type of the proof of `ex_lfp_geq` is just the type of `Frontier`.

We can now start to define a computable example using `thm_32`. 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, making much of the Haskell code hard to read.

```
Example Cs := [
  ["a" & 0] ⇝ "b" & 1;
  ["b" & 1] ⇝ "c" & 2
].
Example f := frontier_fin_0.
Example vars' := nodup string_dec (vars Cs).

Example thm_32_example :=
  thm_32
    (Datatypes.length vars')
    (Datatypes.length vars')
    Cs
    vars'
    []
    f.
```

Listing 5.3: `thm_32` example

In Coq, the type of `thm_32_example` is `pre_thm` partially applied to all the arguments given. If we would like to execute this program entirly in Coq, we would need to give a proof of each of the assumptions in `pre_thm`, and the resulting type of the output would be the type of the proof of `ex_lfp_geq`, again partially applied to all the arguments given.

"partially applied to the arguments given" might not be the best explanation here? what I'm trying to say is that the type of thm_32_example has less arrows than pre_thm since we have given it some arguments

Since Coq elimiates many of the logical parts of the proof when extracting (cite coq extraction papers), we can avoid the tedious task of proving all the assumptions of `pre_thm` by simply using the extracted Haskell function for `thm_32_example`, which as we saw previously only needs a `Frontier` as input (since we have given every other argument necessary), and returns a `Frontier` as output. This frontier should be the same as the one given as input, i.e. `f` in this example. We can then apply the extracted Haskell function `thm_32_example` to `f`, and we receive a `Frontier` as output. This `Frontier` can then be applied to any string to get the resulting value of the string, 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 definition of the lemma in the extracted code, but will immidiately crash the program if the extracted function representing the lemma is ever 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 `Infty` 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 comes with the Haskell compiler GHC.

```
ghci> (thm_32_example f) "a"
Fin 0
ghci> (thm_32_example f) "b"
Fin 1
ghci> (thm_32_example f) "c"
Fin 2
ghci> (thm_32_example f) "x"
Fin 0
```

Listing 5.4: `thm_32` example output

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 universe levels in the type system of Coq.

explain more about type universes in Coq, see LateX comment for possible citation

5.3 Limitations

Chapter 6

Evaluation

Chapter 7

Related & Future Work

Chapter 8

Conclusion

Bibliography

[1] Ulf Norell Ana Bove, Peter Dybjer. A Brief Overview of Agda – A Functional Language with Dependent Types.
URL: `https://www.cse.chalmers.se/~ulfn/papers/tphols09/tutorial.pdf`. Accessed: 2023-05-01.

[2] 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`.

[3] 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`. Accessed: 2023-03-21.

[4] 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.

[5] Thomas C. Hales. Formal Proof. *Notices of the American Mathematical Society*, 55 (11):1370, 2008.
URL: `https://www.ams.org/notices/200811/200811FullIssue.pdf`. Accessed: 2023-03-21.

[6] 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.

[7] Jeremy Avigad Floris van Doorn Jakob von Raumer Leonardo de Moura, Soonho Kong. The lean theorem prover. In *25th International Conference on Automated Deduction (CADE-25), Berlin, Germany*, 2015.
URL: `https://leanprover.github.io/papers/system.pdf`. Accessed: 2023-05-01.

[8] 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.

[9] Matthieu Sozeau. Universes loop checking with clauses. **URL:** `https://github.com/coq/coq/pull/16022`. Accessed: 2023-05-01.

[10] The Coq Team. Calculus of Inductive Constructions, . **URL:** `https://coq.github.io/doc/v8.9/refman/language/cic.html#calculusofinductiveconstructions`. Accessed: 2023-05-01.

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

[12] 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