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

Andreas Salhus Bakseter

Department of Informatics
University of Bergen

June 28, 2023

Overview

1. Background
2. The Case
3. Approach & Design Choices
4. Implementation
5. Examples & Results
6. Related & Future Work
7. Evaluation & Conclusion

Background – Proofs

- important part of mathematics, used to justify a claim or to explain why the claim is true
- **informal proof**: written in a natural language, for humans.
  "Any natural number $n \in \mathbb{N}$ is either even or odd."
- **formal proof**: written in a formal language, for computers.

```
Theorem even_or_odd :
  forall n : nat,
  exists k : nat,
    n = 2 * k ∨ n = 2 * k + 1
```

- **formalization**: informal $\Rightarrow$ formal
- **verification**: check correctness of formal proof

Background – (Dependent) Type theory

- A foundation of mathematics
- Every mathematical object has a type, $a : A$
- We must know the construction of every element of a type
- Provides us with rules of inference for manipulating objects
- **Dependent types**:
  - Types can depend on objects
  - Gives us more expressive power
  - Using Σ-types and Π-types we can model existential and universal quantification

Background – Propositions as types

- Propositions can also be viewed as types, with their elements being proofs
- Since we know the construction of any element of a type, we know the construction of any proof of a proposition
- Also means that proofs are programs, i.e. they can be executed (as long as no axioms are used)

Background – Proof Assistants

- Software tools for constructing & verifying formal proofs
- Based on type theory, uses propositions as types
- Coq, Agda, Lean, etc.
- Often supports program extraction

The Case – Research questions

Two problems in lattice theory solved by Bezem & Coquand. We want to answer the following questions:

1. Are the results from Bezem & Coquand correct?
2. Is it feasible to formalize non-trivial proofs as in Bezem & Coquand, given our experience & time constraints?
3. Is the formalization process worth the effort, i.e. what do we gain from it?

The Case – Relevant parts of the paper

- **join-semilattices**: partially ordered set where any two elements have a least upper bound, called their join, denoted $\vee$
- **inflationary endomorphism**: function that maps an element to itself or to a greater element in the ordered set, denoted $\_^+$
- **semilattice presentation p1**: Set V of variables and set C of constraints. For a semilattice term $t \in V$ and $k \in \mathbb{N}$, $t + 0 = t$ and $t + 1 = t^+$. A term over V has the form $x_1 + k_1 \vee \ldots \vee x_m + k_m$, where $x_i \in V$ and $k_i \in \mathbb{N}$.

The Case – Relevant parts of the paper

- **Horn clauses**: Propositional clauses $A \to b$ with a non-empty body of atoms A and a conclusion b. Atoms are of the form $x + k$, where $x \in V$ and $k \in \mathbb{N}$.
- **semilattice presentation p2**: A constraint over V has the form $s = t$, where s and t are terms over V. For a constraint s = t, we can generate Horn clauses by replacing join by conjunction, denoted by "," and $\leq$ by implication. The set of all generated clauses from a constraint is denoted by $S_{s=t}$. We define S_C as the union of all sets of clauses generated by constraints in C.

The Case – Relevant parts of the paper

Example of generating clauses from constraints:

1. $a \vee b = c^+ \xrightarrow{\text{axiom}} a \vee b \leq c^+$ and $c^+ \leq a \vee b$.

   1.1 $a \vee b \leq c^+ \xrightarrow{\text{generate}} a, b \to c^+$

   1.2 $c^+ \leq a \vee b \xrightarrow{\text{axiom}} c^+ \leq a$ and $c^+ \leq b$.

      1.2.1 $c^+ \leq a \xrightarrow{\text{generate}} c^+ \to a$

      1.2.2 $c^+ \leq b \xrightarrow{\text{generate}} c^+ \to b$

We are then left with $S_{a \vee b = c^+} = \{a, b \to c^+, c^+ \to a, c^+ \to b\}$.

The Case – Relevant parts of the paper

- Horn clause provability $\Longleftrightarrow$ semilattice provability
- **closure under shifting upwards**: if $A \to b$ is in a set of clauses, then so is $A + 1 \to b + 1$, where $A + 1$ is every atom in A shifted upwards by 1.
- $\overline{S_C}$: the smallest subset of S_C that is closed under shifting upwards.
- $\overline{S_C} \mid W$: the set of clauses in $\overline{S_C}$ with only variables from W.
- $\overline{S_C} \downarrow W$: the set of clauses in $\overline{S_C}$ with conclusion over W.
- **frontier**: a model of a set of clauses $\overline{S_C}$ is defined by a function $f : V \to \mathbb{N}^\infty$, where $\mathbb{N}^\infty$ is the set of natural numbers extended with ∞.

The Case – Lemmas & Theorems

- **Lemma 3.1**: given a base clause, it is decidable whether or not all shifts of that clause is true in a frontier
- **Theorem 3.2**: we can compute the least model of a set of clauses $\overline{S_C}$
- **Lemma 3.3**: given some assumptions, we can compute the least model of $\overline{S_C} \downarrow W$ from the least model of $\overline{S_C} \mid W$

Approach & Design Choices – Simplifications

1. Incomplete purely logical formal proofs
2. Leaving out formal proof for Lemma 3.3
3. Proof of minimality of least model

Approach & Design Choices – Modeling finite sets in Coq

- `List` & `ListSet`
  - uses lists, familiar and easy, inductive
  - has order and duplicates, can be combated
  - type polymorphic, requires decidable equality
- `MSetWeakList`
  - a lot more complicated, module functor
  - requires boilerplate code for each type
- `Ensembles`
  - uses inductive propositions, i.e.: $x \in A \Rightarrow x \in A \cup B$
  - cannot reason about set size, ergo useless for us

We went with `List` & `ListSet`.

Implementation – Data types

- **Atom**: `string` → `nat` → `Atom`, e.g. `"x" & 0`
- **Clause**: `set Atom` → `Atom` → `Clause`, e.g. `["x" & 0; "y" & 1] ~> "z"& 2`
- **Ninfty**: either a natural number `fin n` or infinity `infty`
- **Frontier**: `string` → `Ninfty`

Implementation – Data types

- **check if satisfied by frontier**: `atom_true`, `clause_true`
- **shift by n upwards**: `shift_atom`, `shift_clause`
- **satisfied for all shifts**: `all_shifts_true`
  - decidable by Lemma 3.1
  - will be used later to determine model

Implementation – Functions & predicates

- `sub_model Cs V W f : bool`
  For each clause in `Cs`, checks three conditions:
  1. variable in conclusion of clause is not in `W`
  2. some variable in the premise of the clause is not in `V`
  3. the clause is true for all shifts

  If any of these are true for all clauses, `f` is a model of `Cs`.
  Can handle all of $\overline{S_C}$, $\overline{S_C} \mid W$ and $\overline{S_C} \downarrow W$.
- `sub_forward Cs V W f : set string * Frontier`
  For each clause in `Cs`, checks the same as `sub_model`, and adds variable in conclusion to set of "improvable" variables if 1,2,3 are all false. Returns tuple where first value is the set of improvable variables, and the second value is a new frontier that is incremented by one for each variable in the set.

Implementation – Functions & predicates

- `geq V g f : bool`
  Checks if frontier `g` is greater than or equal to frontier `f` for all variables in `V`.
- `ex_lfp_geq Cs V W f : Set :=`
  `exists g : Frontier, geq V g f ∧ sub_model Cs V W g.`

Implementation – Theorem 3.2

```
Theorem thm_32 :
  forall n m Cs V W f,
    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).
```

Implementation – Theorem 3.2 proof overview

- Proof by primary induction on `n` (length of `V`), and secondary induction on `m` (length of `V - W`). Both base cases and induction step of `n` relatively easy, induction step of `m` by far most complex.
- In induction step of `m`, we distinguish between three cases:
  1. $W = \emptyset$
  2. $W = V$
  3. $\emptyset \subset W \subset V$, apply primary and secondary induction hypotheses, and apply Lemma 3.3

Extraction and output examples

- We can extract formal proofs to Haskell code, and run Theorem 3.2 on a set of clauses.
- $\boldsymbol{a \to b + 1, b + 1 \to c + 2}$:

```
Example Cs := [ ["a" & 0] ~> "b"& 1; ["b" & 1] ~> "c"& 2 ]
thm_32 Cs "a"⇒ Fin 0
thm_32 Cs "b"⇒ Fin 1
thm_32 Cs "c"⇒ Fin 2
```

- $\boldsymbol{a \to a + 1}$:

```
Example Cs_loop := [ ["a"& 0] ~> "a"& 1 ]
thm_32 Cs_loop "a"⇒ Infty
```

Real-world example & limitations

- We can use Theorem 3.2 algorithm to check type universe level consistency in Coq, which Coq does when type-checking.
- The Coq Command `Print Universes` gives us a list of universe constraints, can be translated to our syntax. In testing this produced correct type universe levels for over 5000 constraints.
- Algorithm does not always work due to Lemma 3.3 not formally proven, e.g. fails when extending first example with `Example Cs := [ (*...*)["c"& 2] ~> "d"& 3 ].`

Related work

Ongoing effort by Matthieu Sozeau, Coq Team to implement version of algorithm for use in universe consistency checking in Coq. Main focus on speed, and only supports clauses of the form $x + k \geq y$, where $k \in \{0, 1\}$.

Future work

1. Complete proofs of remaining logical lemmas
2. Formal proof of Lemma 3.3
3. Prove minimality of model in Theorem 3.2

Evaluation & Conclusion

1. Are the results correct?
   - we have not formalized every part of results, and we have simplified some proofs
   - but, with a full formal proof of Theorem 3.2, we can be very confident that the results are correct
2. Is it feasible to formalize such complex proofs?
   - by our effort, yes
   - could be done faster and/or better with more experience
3. Is the formalization process worth the effort?
   - we gain a usable prototype
   - can be used as a starting point for more efficient implementations