Concrete Semantics with Coq and CoqHammer

Łukasz Czajka[1], Burak Ekici[2], and Cezary Kaliszyk[2][0000−0002−8273−6059]

[1] University of Copenhagen, Copenhagen, Denmark
luta@di.ku.dk
[2] University of Innsbruck, Innsbruck, Austria
burak.ekici,cezary.kaliszyk@uibk.ac.at

Abstract The "Concrete Semantics" book gives an introduction to imperative programming languages accompanied by an Isabelle/HOL formalization. In this paper we discuss a re-formalization of the book using the Coq proof assistant (version 8.7.2). In order to achieve a similar brevity of the formal text we extensively use CoqHammer[3], as well as Coq Ltac-level automation. We compare the formalization efficiency, compactness, and the readability of the proof scripts originating from a Coq re-formalization of two chapters from the book.

1 Introduction

Formal proofs allow today most precise descriptions and specifications of computer systems and programs. Such precision is very important both for human learning and for machine knowledge management. Formalization accompanied courses allow students to investigate the topic to an arbitrary level of detail, and naturally offer very precise exercises of the topic [7]. Formalization attached to mathematical knowledge allows algorithms to the knowledge semantically and permits learning machine translation to, from, and between datasets [5]. This becomes even more important with multi-translation, where the availability of the same text in multiple languages improves the computer-understanding and ability to translate between each two [4].

In this short paper we translate parts of the Concrete Semantics book by Nipkow and Klein to Coq. To do so, we improve the CoqHammer [9] automation to be able to handle the more advanced use-cases, improve the legibility of the reconstructed proofs and compare the proof style and other differences in between the two. The project is in some ways similar to the "Certified Programming with Dependent Types" book [2], however we attempt to avoid dependent types and more advanced constructions to build both an easier material for students and a more precise dataset for bootstrapping an automated translation between proof corpora in the style of [5].

2 Concrete Semantics with Isabelle/HOL

The Concrete Semantics book [7] by Nipkow and Klein is made of two parts. The first part introduces how to write functional programs, inductive definitions

[3] release: https://github.com/lukaszcz/coqhammer/releases/tag/v1.0.8-coq8.7

and how to reason about their properties in Isabelle/HOL's structured proof language. While the second part is devoted to formal semantics of programming languages using the "small" imperative `IMP`[4] language as the instance. This part more concretely examines several topics in a wide range varying from operational semantics, compiler correctness to Hoare Logic. The proofs presented in this part are not given in Isabelle/HOL's structured language. However, such a formalization accompanies the paper proofs via the provided links usually given in section beginnings.

In this work we attempt to reformalize in Coq some subset of the Isabelle/HOL theories that accompanies the second part of the book. As illustrated in Chapter 4, we aim at catching the same level of automation in Coq thus approximating the proof texts to the original ones in terms of length. To do so, we use automated reasoning techniques discussed in Chapter 3.

3 Coq and Coq Automation

The Coq proof assistant is based on the Calculus of Inductive Constructions. The main difference from proof assistants based on higher-order logic is the presence of dependent types. Coq also features a rich tactic language Ltac, which allows to write specialised proof automation tactics. Some standard automation tactics already available in Coq are:

- `intuition`: implements a decision procedure for intuitionistic propositional calculus based on the contraction-free sequent calculi LJT* of Roy Dyckhoff.
- `firstorder`: extends `intuition` to a proof search tactic for first-order intuitionistic logic.
- `auto` and `eauto`: implement a Prolog-like backward proof search procedure.

The CoqHammer [9,3] plugin extends Coq automation by a number of other useful and generally more powerful tactics similarly to that available in Isabelle [1]. Its main tactic `hammer` combines machine learning and automated reasoning techniques to discharge goals automatically. It works in three phases:

1. **Premise selection** uses machine learning techniques to choose a subset of the accessible lemmas that are likely useful for the goal.
2. **Translation** of the goal and the preselected lemmas to the input formats of first-order automated theorem provers (ATPs) such as Vampire [6] or Eprover [8], and running the ATPs on the translations.
3. **Reconstruction** uses the information obtained from a successful ATP run to re-prove the goal in the logic of Coq. Upon success the `hammer` tactic should be replaced with the reconstruction tactic displayed in the response window. The success of the reconstruction tactic does not depend on any time limits nor external ATPs, therefore it is machine-independent.

[4] `IMP` is a standard Turing complete imperative language involving the mutable global state as a computational side effect. The reason why this language has been selected is just that it has enough expressive power to be Turing complete.

The CoqHammer tool provides various reconstruction tactics. Among others, the tactics `hobvious` and `hsimple` perform proof search via the `yelles` tactic (see the last item below) using the information returned from the successful ATP runs after a constant unfolding and hypothesis simplification. Also, CoqHammer comes with tactics written entirely in Ltac. These tactics do not depend on any external tool, and are not informed about available lemmas in the context:

- `sauto` – a "super" version of the standard Coq tactics `auto` and `intuition`. It tries to simplify the goal and possibly solve it without performing much of actual proof search beyond what `intuition` already does. It is designed in such a way as to terminate in a short time in most circumstances. One can customize it by adding rewrite hints to the `yhints` database.
- `scrush` – essentially a combination of `sauto` and `ycrush`. The `ycrush` tactic tries various heuristics and performs some limited proof search. Usually stronger than `sauto`, but may take a long time if it cannot find a proof. In contrast to `sauto`, `ycrush` does not perform rewriting using the hints in the `yhints` database. One commonly uses `ycrush` after `sauto`.
- `yelles n` – performs proof search up to depth n; slow for n larger than 3-4.

4 Case Studies

In this section, we illustrate a set of goals that are discharged using the Coq automation techniques, presented in Section 3, together with a comparison to their original versions, in an Isabelle/HOL formalization, as presented in the Concrete Semantics book. Notice that the examples in this section are given broadly, with no background details. The point to emphasize here is that we can actually achieve a similar brevity of the formal text in terms of proof lengths using proof automation in Coq. The examples are given in code snippets that have Coq text on the left and Isabelle/HOL text on the right side of the minipages.

Note also that we translated thelemma statements into Coq directly from Isabelle/HOL theory files, and proved them using mostly the standard tactics coming with CoqHammer, with only minimal use of more sophisticated custom Ltac tactics, and practically no hints from Coq hint databases. Therefore the translation is not quite automatic but fairly straightforward.

The example given in the below code snippet comes from the Hoare Logic. Leaving the technical details aside, it basically says that a precondition `{P}` of some Hoare triple can be strengthened into `{P'}` if `{P'}` entails `{P}`. This is actually one of the corollaries of the consequence (called `conseq` in our Coq formalization) rule of Hoare Logic. Notice that, in this snippet, `hoaret` is the Coq inductive predicate representing Hoare triples which corresponds to the notation "$\vdash_t$" on Isabelle/HOL side.

```
Lemma strengthen_pre:
∀ (P P' Q: assn) c, (entails P' P)
→ hoaret P c Q → hoaret P' c Q.
Proof. hobvious Empty (@conseq) (@entails) Qed.
```

lemma strengthen-pre:
⟦ ∀s. P' s ⟶ P s; $\vdash_t$ {P} c {Q} ⟧
⟹ $\vdash_t$ {P'} c {Q}
by (metis conseq)

Upon a call, the CoqHammer tool gets a proof returned by one of the employed ATPs, and discharges the goal using its reconstruction tactic `hobvious`

parametrized with the empty set of hypotheses from the goal context, the rule `conseq` and the definition `entails`. Indeed, this is very similar to what happens in Isabelle/HOL proof of the same fact. The proof is simply made of a call to the `metis` tactic with the `conseq` rule as the argument.

Another but slightly more complicated example that stems from the Hoare Logic (using the same notation as the previous one) is given in the below code snippet. This lemma is a version of the partial correctness of the `while` rule enriched with a measure function `f` which is supposed to decrease in each loop iteration so as to guarantee the loop termination.

```
Lemma While_fun: ∀ b P Q c
(f: state → nat), (∀ n: nat, hoaret
(fun s ⇒ P s ∧ bval s b = true∧
n = f s) c (fun  s ⇒ P s ∧ f s < n))
→ hoaret P (While b c)
(fun s ⇒ P s ∧  bval s b = false).
Proof. pose While; pose conseq;
unfold entails in *; yelles 3. Qed.
```

lemma While-fun:
⟦ ⋀n::nat. ⊢$_t$ {λs. P s ∧ bval b s ∧ n = f s} c {λs. P s ∧ f s < n}⟧
⟹ ⊢$_t$ {P} WHILE b DO c {λs. P s ∧ ¬bval b s}
by (rule Hoare-Total.While [**where** T=λs n. n = f s, simplified])

The Coq proof is found by the Ltac implemented tactic `yelles` which performs a proof search until a user specified depth has been reached. In our concrete example, we give it some guidance by using the primitive Coq tactic `pose` with `while` and `conseq` rules as arguments, adding them to the context (or simply generalizing them), together with unfolding the definition of `entails`. This way, the tactic finds a proof at the proof search depth 3. Isabelle/HOL proof of the same statement follows similar lines. It uses a simplification of the `while` rule with the measure function being `λs n. n = f s`. Just notice that our Coq tactic `yelles` is clever enough on this goal to find the measure function automatically.

A third example is about semantics of the `IMP` language. The lemma shown in the below snippet states that one can deduce the big-step semantics of any terminating `IMP` program from its small-step semantics. Observe that, in this snippet, the Coq notations "`⟹`" and "`⟶ *`" respectively represent the inductive predicates for the (transitive closure of) `IMP` big-step and small-step semantics. The single difference on Isabelle/HOL side is that we have "⇒" standing for big-step semantics.

```
Lemma lem_small_to_big: ∀ p s,
p ⟶* (Skip, s)  → p ⟹ s.
Proof. enough (∀ p p', p ⟶* p' →
∀ s, p' = (Skip, s)  → p ⟹ s) by scrush.
intros p p' H. induction H; sauto.
hsimple AllHyps (@lem_small1_big_continue)
Empty. Qed.
```

lemma small-to-big:
cs →* (SKIP,t) ⟹ cs ⇒ t
apply (induction cs (SKIP,t) rule: star.induct)
apply (auto intro: small1-big-continue)
done

The Coq proof of this lemma proceeds by an induction on the (transitive closure of) small-step semantics after introducing an helper statement (asserted by the pure Coq tactic `enough` and proven by the Ltac tactic `scrush`) into the goal context. Then, it calls the Ltac tactic `sauto` to do some preprocessing for the CoqHammer call. The base case `p = (Skip, s)` is trivially solved by `sauto`. For the inductive case, namely `∀s, p' = (Skip, s) → p' ==> s`, we call CoqHammer and get the goal solved by an application of the reconstruction tactic `hsimple` which uses all hypotheses in the goal context (that's why we introduce a new one at the beginning) and the helper lemma called `lem_small1_big_continue` with no definitions unfolded. This is again very similar to the Isabelle/HOL

proof of the fact in hand. The proof uses the induction principle on the transitive closure of the small-step IMP semantics and then applies the helper lemma lem_small1_big_continue. Below we give three more examples that we think interesting in the sense that all cases appear on Coq side are discharged fully automatically. And the text size is fairly close to the one of Isabelle/HOL.

```
Lemma exec_n_exec: ∀ n P c1 c2,
exec_n P c1 n c2 → exec P c1 c2.
Proof. induction n; intros; destruct H.
- scrush.
- pose @star_step;
hobvious (@H, @IHn)(@Star.star_step)
(@Compiler.exec). Qed.

Lemma exec_exec_n: ∀ P c1 c2,
exec P c1 c2 →∃ n, exec_n P c1 n c2.
Proof. intros; induction H.
- ∃ 0; scrush.
- pose exec_Suc; scrush. Qed.

Lemma exec_eq_exec_n: ∀ P c1 c2,
exec P c1 c2 ↔ ∃ n, exec_n P c1 n c2.
Proof. pose exec_exec_n;
pose exec_n_exec; scrush. Qed.
```

```
lemma exec-n-exec:
  P ⊢ c →^n c ⟹ P ⊢ c →* c
  by (induct n arbitrary: c) (auto intro: star.step)

lemma exec-exec-n:
  P ⊢ c →* c ⟹ ∃n. P ⊢ c →^n c
    by (induct rule: star.induct) (auto intro:
exec-Suc)

lemma exec-eq-exec-n:
  (P ⊢ c →* c) = (∃n. P ⊢ c →^n c)
  by (blast intro: exec-exec-n exec-n-exec)
```

The main lemma exec_eq_exec_n, using the other two above, is broadly about the symbolic compilation of IMP programs into a low level language based on a stack machine. It specifically says that one can speak about the n step instruction executions instead of reflexive transitive closure of single step executions. Note that the Coq predicates exec_n and exec are respectively standing for n step instruction, and transitive closure of single step instruction. These are denoted as "_ ⊢ _ → ∗ _" and "_ ⊢ _ →^ n _" in Isabelle/HOL text respectively. The Coq proof from left to right (exec_n_exec) of the equivalence is based on an induction over n and the other direction (exec_exec_n) is based on an induction over transitive closure of single step executions. The base cases of both induction steps are trivially solved by the tactic scrush. The inductive case of the former proof is a CoqHammer call which discharges the goal using the reconstruction tactic hobvious. It uses two hypotheses (H and the induction hypothesis IHn) from the goal context with the lemma called star_step (coming from Star.v), and the definition exec. The inductive case of the latter is just made of an scrush application with a guide reminding that the goal is a variant of the lemma exec_Suc.

Again, these proofs follow very similar lines with those of Isabelle/HOL of the same facts. The proof of exec_n_exec induces on n and uses the definition star_step to discarge the goal. Similarly, exec_exec_n is proven by an induction on the transitive closure of single step executions followed by the application of the exec_Suc fact.

5 Conclusion

We have reproven 101 lemmas from the Isabelle/HOL theories Star, AExp, BExp, ASM, Com, Big_Step, Hoare, Small_Step, Compiler and Compiler2 in Coq; heavily using the automation techniques described in previous sections.

	# of lines	# of words	# of tactics	# of hammer calls	time(secs)
Isabelle/HOL	2806	11278	544	not verifiable	31
Coq	3493	19292	1190	468	149

As shown in the above table, the number of Coq tactics we used to get the same lemmas proven is almost twice in number, as opposed to Isabelle/HOL, but about half of which benefits from the automation techniques that CoqHammer comes with. This can be seen as an improvement given that the Isabelle/HOL tactics are more compound than the "simple" Coq tactics.

The `coqc 8.7.2` needs 149 seconds to compile the translated source and Isabelle 2017 needs 31 seconds to build the corresponding theories on an Intel Core i7-7600U machine. We attribute the difference mostly to the fact that all used Isabelle tactics are written in ML, while most Coq ones use Ltac.

We plan to build on this work by proving more lemmas coming from different theories of the book and by improving the level of automation, thus decreasing the number of words, in the already proven goals. Please see:

https://github.com/lukaszcz/COQ-IMP

for the proofs done so far.

Acknowledgments This work has been supported by the Austrian Science Fund (FWF) grant P26201, the European Research Council (ERC) grant no. 714034 *SMART* and the Marie Skłodowska-Curie action *InfTy*, program H2020-MSCA-IF-2015, number 704111.

References

1. J. C. Blanchette, D. Greenaway, C. Kaliszyk, D. Kühlwein, and J. Urban. A learning-based fact selector for Isabelle/HOL. *J. Autom. Reasoning*, 57(3):219–244, 2016.
2. A. Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.
3. Ł. Czajka and C. Kaliszyk. Goal translation for a hammer for Coq (extended abstract). In J. Blanchette and C. Kaliszyk, editors, *International Workshop on Hammers for Type Theories (HaTT'16)*, volume 210 of *EPTCS*, pages 13–20, 2016.
4. D. Dong, H. Wu, W. He, D. Yu, and H. Wang. Multi-task learning for multiple language translation. In *ACL (1)*, pages 1723–1732. The Association for Computer Linguistics, 2015.
5. C. Kaliszyk, J. Urban, J. Vyskočil, and H. Geuvers. Developing corpus-based translation methods between informal and formal mathematics. In S. Watt, J. Davenport, A. Sexton, P. Sojka, and J. Urban, editors, *7th Conference on Intelligent Computer Mathematics (CICM'14)*, volume 8543 of *LNCS*, pages 435–439. Springer, 2014.
6. L. Kovács and A. Voronkov. First-order theorem proving and Vampire. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 1–35, 2013.
7. T. Nipkow and G. Klein. *Concrete Semantics - With Isabelle/HOL*. Springer, 2014.
8. S. Schulz. System Description: E 1.8. In K. McMillan, A. Middeldorp, and A. Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*. Springer, 2013.
9. Łukasz Czajka and C. Kaliszyk. Hammer for Coq: Automation for dependent type theory. *J. Autom. Reasoning*, 61(1-4):423–453, 2018.