EPiC Series in Computing
Volume 94, 2023, Pages 11–22
Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning

An Interactive SMT Tactic in Coq using Abductive Reasoning*

Haniel Barbosa[1], Chantal Keller[2], Andrew Reynolds[3], Arjun Viswanathan[3], Cesare Tinelli[3], and Clark Barrett[6]

[1] Universidade Federal de Minas Gerais, Brazil
hbarbosa@dcc.ufmg.br
[2] Laboratory of Formal Methods, Université Paris-Saclay, France
ckeller@lmf.cnrs.fr
[3] University of Iowa, USA
{andrew-reynolds,arjun-viswanathan,cesare-tinelli}@uiowa.edu
[4] Stanford University, USA
barrettc@stanford.edu

Abstract

A well-known challenge in leveraging automatic theorem provers, such as satisfiability modulo theories (SMT) solvers, to discharge proof obligations from interactive theorem provers (ITPs) is determining which axioms to send to the solver together with the conjecture to be proven. Too many axioms may confuse or clog the solver, while too few may make a theorem unprovable. When a solver fails to prove a conjecture, it is unclear to the user which case transpired. In this paper we enhance SMTCoq — an integration between the Coq ITP and the cvc5 SMT solver — with a tactic called `abduce` aimed at mitigating the uncertainty above. When the solver fails to prove the goal, the user may invoke `abduce` which will use abductive reasoning to provide facts that will allow the solver to prove the goal, if any.

1 Introduction

Interactive theorem provers (ITPs) [8], or *proof assistants*, are used, among other things, to construct proofs of logical properties in various hardware and software verification tasks. Such proofs are reliable due to the prover's small, highly trustworthy proof kernel; but they are often also tedious, involving elaborate sub-proofs of even simple facts that could easily be proved by automated reasoning (AR) tools such as solvers for Satisfiability Modulo Theories (SMT) [6]. Using SMT solvers to automate the proving of basic proof subgoals in proof assistants is, however, problematic in principle since SMT solvers are generally rather complex systems. They typically have very large code-bases, whose correctness is more difficult to trust than that

*This work was partially funded by Amazon Web Services, the Stanford Center for Automated Reasoning, and NSF grant #2019348.

of ITP kernels. Therefore, using results from an SMT solver in a proof assistant amounts to significantly extending its *trust-base*.

SMTCoq [4] is a plug-in for the Coq [35] proof assistant that provides a trustworthy integration into Coq of selected SMT solvers. Using one of SMTCoq's tactics, a Coq user is able to discharge goals via an SMT solver without expanding Coq's trust-base. SMTCoq does this by requiring the external solver to provide a proof certificate for any goal G the solver claims to have proven. This certificate is used, roughly speaking, to construct automatically a proof of G within Coq, fully obviating the need to trust the external solver.

A typical workflow with SMTCoq proceeds as follows. The Coq user calls the `smt` tactic — provided by SMTCoq — on a goal G, which asks the SMT solver to prove the validity of (an encoding of) G. Optionally, the user can pass a set H of additional facts from Coq to the SMT solver as arguments to the tactic, to be used as hypotheses for G. In other words, the user can ask the SMT solver to prove that H entails G.

Outside of a selection of function and predicate symbols from SMTCoq's supported types (`Bool`, `Z`, and a custom type for arrays and bit-vectors), all symbols in a goal or its hypotheses are considered *uninterpreted* by the SMT solver. If the solver succeeds in proving G, it also sends a proof certificate, which SMTCoq then uses to derive a proof of G in Coq's logic via a *computational reflection* process, in which SMT proof terms are defined in Coq's programming language and lifted to Coq terms by a proof of correctness [4]. If the SMT solver finds G to be invalid, it may additionally return a counter-example, which is presented to the Coq user as a witness of G's invalidity. At that point, the user needs to determine whether G is truly invalid in the Coq context or the hypothesis for G provided to the solver did not capture the properties of the uninterpreted symbols of G in the Coq formalization that are needed to prove the goal. A very simple example of this situation would be an arithmetic goal like $\forall x, 0 \leq \mathsf{square}(x) \leq \mathsf{square}(x+1)$, containing applications of a square function symbol defined in Coq as expected but uninterpreted in the SMT solver. An SMT solver supporting linear integer arithmetic would find this goal invalid unless it received as hypotheses formulas that capture the non-negativity and the monotonicity of the square function.

Contribution We describe a new capability of SMTCoq, encapsulated in the new tactic `abduce`, for those cases where the SMT solver finds the goal to be invalid under the given hypotheses H. The new tactic exploits the ability of the cvc5 [5] SMT solver to produce *abducts* for a goal G invalid under H, i.e., formulas that are consistent with H and entail G [33] when added to H. With this tactic, we envision a more interactive session between the Coq user and the SMT solver, where the solver might help answer the question of what information is missing, if any, to prove the goal. The tactic returns one or more abducts as suggestions for the user on how to strengthen the hypotheses for that invalid goal. If a disjunction of some of the provided abducts is provable in Coq, the user can first prove it and then pass it as an additional hypothesis to the `smt` tactic, knowing that at that point the original goal will be proved by the SMT solver.

The rest of the paper is structured as follows. In Section 2, we introduce the formal setting of our problem space, summarize the tools involved, and present related work; in Section 3, we give an example to motivate our tactic. Then, in Section 4, we present the `abduce` tactic, our extension to SMTCoq. Finally, in Section 5, we present some initial experiments, and we conclude with directions of future work in Section 6.

2 Background

Our logical setting is that of classical many-sorted first-order logic with equality, the base logic of SMT [6]. In contrast, Coq is based on the Calculus of Inductive Constructions, a constructive higher-order logic with dependent types [32]. SMTCoq resolves this mismatch by considering, in effect, only quantifier-free goals of the form $\phi = \mathit{True}$ where ϕ is a term of type `Bool`, as opposed to terms of type `Prop`, the designated type for formulas in Coq. This allows SMTCoq to use SMT solvers thanks to a faithful encoding of such goals in the logic of SMT.

We restate some central definitions and notation here. We use standard symbols for connectives and quantifiers within formulas, and represent *True* by $\top$, and *False* by $\perp$. As an abuse of notation, we also use $\perp$ for the empty clause. We use standard notions of signature, formula and interpretation. A theory T is a pair consisting of Σ, a signature; and I, a non-empty set of Σ-interpretations, called the *models* of T. A Σ-formula ϕ is T*-satisfiable* if there exists a model in I that satisfies it, and is T*-unsatisfiable* otherwise. A set Γ of formulas T*-entails* a formula ψ, written $\Gamma \models_T \psi$, if every model of T that satisfies all the formulas in Γ is also a model of ψ. A Σ-formula ϕ is *weaker (in T)* than a formula ψ if $\{\psi\} \models_T \phi$.

2.1 Coq

Coq is a theorem prover with trusted computing base (TCB) that is relatively smaller than those of AR tools, offering strong guarantees about properties proved within this TCB. Coq implements the Curry-Howard isomorphism where properties — stated as logical formulas — are also types, and can be proven by constructing terms of the corresponding type. Via so-called *conversion rules* [28], a proof term in Coq can have two different types as long as they are computationally equivalent. The Coq type-checker plays the role of the guarantor of its TCB. While a user can provide a term of the right type to Coq as a proof, Coq offers an interface to construct proof terms via scripts called *tactics.* Tactics range from single, one-word invocations of previously proven theorems to complicated scripts involving nested case-splittings that may involve inductive reasoning.

When external tools are used for providing automation in Coq, care must be taken so that the TCB is not extended. One way of ensuring this is to re-implement the external tools within Coq and prove them correct [31]; another is to use the external tool as a guide and reconstruct its proofs within Coq [15] (tools that perform such a reconstruction are called *hammers* and exist for other ITPs as well [9, 10]). A different route is to use *proof by reflection*, which allows one to automatically construct proofs in Coq from proofs generated by external provers. Part of this process involves a *deep embedding* of the external prover's logic into Coq's logic. This is done by representing external formulas and proof rules in Coq as data structures in its embedded programming language Gallina [27], and defining their formal semantics in Coq itself. Once this is done, one can write in Coq and prove correct once and for all a *proof checker* for externally generated proofs. This is a decision procedure that checks the correctness of external *proof certificates*, i.e., representations of proofs generated by the external prover. By computational reflection, a Coq goal G can then be considered proven whenever the proof checker approves an externally provided proof certificate for G. One of the earliest tactics to use external SMT solvers in Coq via reflection this way is the `kettle` tactic [14], which was able to do reasoning over equality and linear integer arithmetic. The one we extend in this work is SMTCoq [4].

2.2 SMTCoq

SMTCoq is a Coq plugin that achieves a skeptical cooperation between the Coq proof assistant and SAT and SMT solvers. The majority of SMTCoq's machinery provides a way to computationally reflect a proof certificate from an external solver into a Coq proof term, as described in the previous subsection. This includes a checker for these certificates, and a proof of correctness of this checker in terms of Coq's logic, supported by many efficient data structures to improve scalability.

The goals that SMTCoq can deal with are restricted to a subset of the first-order fragment of Coq's logic. The recent Sniper tool [11] extends the applicability of SMTCoq by providing a number of composable, small-scale transformations from more general Coq goals into the logic fragment supported by SMTCoq. While SMTCoq can handle solver proofs with *holes* in them by presenting the holes as new subgoals to the user, interaction between the user and the external solver is limited — SMTCoq's tactics are considered push-button provers that can either succeed in proving the subgoal or fail. In this work, we address this shortcoming.

2.3 Abduction

Given a set of formulas H (taken as a conjunction), a goal G, and a theory T, an abduct is a formula ϕ such that (1) $H \wedge \phi$ is T-satisfiable and (2) $H \wedge \phi \models_T G$. Abduction has applications in program verification and static analysis, including: loop invariant generation [24, 18]; specification inference [36, 2]; and compositional analysis [13, 19], among others [17, 21].

Several tools that perform abductive reasoning have also been developed over the years. Echenim et al. [25, 22] modify the superposition calculus to present an abductive algorithm for prime implicate generation in the theory of equality. GPiD [23] is a tool for abduction built on top of SMT solvers. EXPLAIN [16] also leverages SMT technology to perform abduction — in this case, the Mistral [20] SMT solver is used. CAPI [26] uses abduction in descriptive logics to provide explanations for observations that do not hold.

cvc5 performs abductive reasoning via syntax-guided synthesis (SyGuS) [3]. Thus, in addition to the semantic conditions (1) and (2) listed above, it constrains abducts to range over formulas generated by a user-provided context-free grammar R. The grammar input is optional, with the default being the grammar that generates the entire language of the theory T.

The solver is driven by a basic CEGIS [34] procedure: candidate abducts ϕ, formulas in the language generated by R that satisfy the consistency requirement (1) above, are validated by checking whether $H \wedge \phi \models_T G$. Valuations (of the free symbols of $H \wedge \phi$ and G) that invalidate this entailment, by satisfying $H \wedge \phi$ and falsifying G, are collected and used to guide the search for a solution: future candidates ϕ that are satisfied by any of those valuations are immediately discarded as they are guaranteed to fail the entailment check. cvc5 refines this basic CEGIS procedure with various optimizations and symmetry-breaking strategies which eliminate redundant solutions and accelerate the search.

In the context of SMTCoq, a more general abduct is preferable to a less general one since the latter is less likely to be provable in Coq. In light of this, we have modified cvc5 to generate a *sequence* of abducts for the same problem so that their disjunction is typically weaker in T than the individual abducts. This has the effect of producing a progressively more general (disjunctive) abducts at the cost of additional computation. This cost can be controlled by the user by specifying the length of the abduct sequence.

3 Motivating Example

Suppose our Coq development contains a binary function `f` of type `Z → Z → Z` (where `Z` is Coq's integer type) and many facts about `f`.

Example 3.1. We can invoke SMTCoq through the `smt` tactic as follows.

```
Goal forall (x y : Z), x = y → f x (y + x) = f y (22 * y − 20 * x).
Proof. smt. Qed.
```

In Example 3.1, the SMT solver is able to prove a simple fact about `f` without using user-provided axioms since it is able to prove the goal using equational reasoning along with linear arithmetic. Now, consider a more interesting goal, whose validity depends on the specific behavior of `f`.

Example 3.2. The following proof cannot be closed, since the tactic fails.

```
Goal forall (x y z : Z), x = y + 1 → (f y z) = f z (x − 1).
Proof. smt.
```

The solver gives a counterexample witnessing the failure of the proof, shown to the user:

$$\mathtt{f} \mapsto \lambda\, \mathtt{x}, \mathtt{y} \implies \mathtt{x}, \qquad \mathtt{x} \mapsto 1, \qquad \mathtt{y} \mapsto 0, \qquad \mathtt{z} \mapsto 1$$

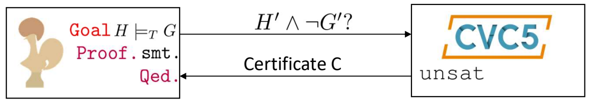
It is possible that the solver failed because the goal is indeed invalid. However, considering that the solver does not have access to a definition of `f` or an axiomatization of its properties, it is also possible that the solver is missing one of those additional facts from Coq. Instead of trying to determine which one this might be (one that is falsified by this counter-example), the user may invoke cvc5's abduction capability to get a suggestion from the solver.

4 The `abduce` Tactic

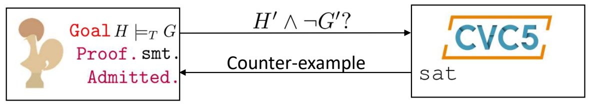
A goal in SMTCoq is a logical formula to be proven from a (possibly empty) set of premises H in some theory T. So, we can identify it with the goal of proving the entailment $H \models_T G$. SMTCoq converts H and G to some corresponding SMT formulas H' and G', phrases the entailment between them as an implication, and sends the negation of this implication to the SMT solver. Thus, the goal of showing that $H \models_T G$ holds is reduced to that of showing that $H' \wedge \neg G'$ is T-unsatisfiable. For any particular Coq goal supported by SMTCoq and sent to the SMT solver, there are three possible outcomes: (*i*) the solver proves the goal, by finding $H' \wedge \neg G'$ to be T-unsatisfiable; (*ii*) it disproves the goal, by finding $H' \wedge \neg G'$ to be T-satisfiable; (*iii*) it produces an "unknown" answer because it ran out of resources. An acceptable certificate for outcome (*i*) is a proof of unsatisfiability, a formal proof that derives $\perp$ from $H' \wedge \neg G'$. An acceptable certificate for outcome (*ii*) is a *counterexample*, a valuation of the (free) variables of $H' \wedge \neg G'$ that satisfies H' and falsifies G'.

Example 3.1 is an illustration of outcome (*i*), and Example 3.2 demonstrates outcome (*ii*). Figures 1a and 1b show the interaction between Coq and the SMT solver for both situations.

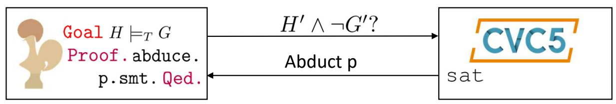
With our new `abduce` tactic in SMTCoq, a Coq user can ask cvc5 for abducts that would entail a currently failing goal. An integer argument allows the user to request a particular number of independent abducts, with the guarantee that each abduct separately entails the goal (guaranteeing, as a consequence, that their disjunction also entails the goal) along with the hypotheses. The tactic invokes cvc5's SyGuS-based abduction solver as discussed in Section 2.3.



(a) cvc5 finds query to be unsatisfiable



(b) cvc5 finds query to be satisfiable



(c) cvc5 returns an abduct

Figure 1: Interaction of SMTCoq with the SMT solver. $H = \{H_1, H_2, \ldots, H_n\}$ is the set of hypotheses sent to the solver.

Example 4.1. Consider Example 3.2 from Section 3. We can use the `abduce` tactic on this goal, since `smt` fails.

```
Goal forall (x y z : Z), x = y + 1 → (f y z) = f z (x − 1).
Proof. (* smt. Fails with counter-example *) abduce 3.
```

This presents three abducts to the user: `z = y`; `z + 1 = x`; and `f z y = f y z`. The third abduct might suggest to the user that cvc5 would prove the goal if it was told that the function is commutative or, more specifically, that it is commutative over `y` and `z`. If one of our previously-proven facts about `f` is:

`comm_f : ∀ m n, f m n = f n m`

the user can easily instantiate it in Coq for the necessary variables. A subsequent call to the `smt` tactic with this instantiated fact in scope would successfully close the proof. We can now complete the proof.

```
Goal forall (x y z : Z), x = y + 1 → (f y z) = f z (x − 1).
Proof. intros. assert (f z y = f y z). { apply comm_f. } smt. Qed.
```

The `intros` tactic introduces `x`, `y` and `z`, and the hypothesis `x = y + 1` into the scope of the proof. `assert` is a way to locally introduce a fact into scope, and we use it to state the chosen abduct. The abduct is easy to prove by an application of `comm_f`. At that point, the `smt` tactic can successfully prove the current goal `(f y z) = f z (x - 1)` from the (automatically collected) local hypotheses `x = y + 1` and `f z y = f y z`.

We point out that in cases where SMTCoq disproves the goal (outcome (*ii*) above), the `abduce` tactic can provide a more general explanation of the failure than a counterexample. Counterexamples are single points over which the entailment $H \models_T G$ fails whereas an abduct represents a general sufficient condition for the provability of the goal that the user might be able to prove and then provide to the SMT solver from the current Coq context. Since there are a large number of these additional hypotheses that might help in proving a given goal, it is impractical to send all of them along with the goal. Abduction is then a way for the SMT solver to tell the user what else it needs. Figure 1c illustrates this case.

5 Evaluation

In this section, we present a preliminary case study on applying the `smt` and `abduce` tactics in the Coq library `Zorder` [29], with the goal of simplifying the proofs in it.[1] The library contains theorems about order predicates over Coq's `Z` (integer) type. While this library is deprecated, its lemmas are still available in the Coq core libraries.

Our study demonstrates the utility of the `smt` tactic and provides a proof-of-concept use case for interacting with the SMT solver via the `abduce` tactic in an IDE for Coq. We ran all experiments on CoqIDE version 8.16.1 in a system with 16 GB RAM, running Ubuntu 20.04. Our experimental set-up is as follows. Within the `Zorder` Coq file, we import SMTCoq as a plug-in, and for each goal, we first try the `smt` tactic, which attempts to solve the goal using a combination of the SMT solver CVC4 [7] and veriT[12], both of which are well integrated in SMTCoq. We define the goal as an `smt` *success* if it can be fully solved by the SMT solver (with no additional hypotheses). In cases where the solver finds the goal to be invalid, we repeatedly call `abduce` n with $n = 1, 2, 3 \dots$ and a 20 second timeout per call, stopping as soon as we find a suitable abduct or the solver times out for some n. Recall that `abduce` n asks for n abducts, each of which independently entail the goal. We classify the goal as an `abduce` *success*, if a call to `abduce` produces (within the given timeout) an *easily provable* abduct which, once added locally, allows `smt` to prove the goal. Otherwise, we classify the goal as a *timeout*. For the purposes of this experiment we consider an abduct to be easily provable if it is provable from the Coq context by just unfolding once any applications of the integer successor or predecessor functions (`Z.succ` or `Z.pred`, respectively) in the abduct.[2]

Our results are presented in Figure 3. From the 93 goals in the file, 30 goals contain non-linear arithmetic, a theory currently unsupported by SMTCoq; 3 goals relate to decidability in Coq, which cannot be proved by an SMT solver; and 1 contains predicates unrecognized by SMTCoq. From the remaining 59 goals, we found 33 (55.9%) `smt` successes, and 26 candidates for abduction, half of which were `abduce` successes.

All goals found invalid by the SMT solver were so because they contained either Coq's integer successor or integer predecessor functions, `Z.succ` and `Z.pred`. When successful, the abduction solver was able to suggest either definitions of `Z.succ` and `Z.pred`, or properties satisfied by them in Coq. Both forms of abducts could be proven locally by unfolding the definitions of those functions, and applying some basic properties of inequalities over integers. We further automate this process by calling `smt` on the unfolded sub-goal. For example, consider goal `Znot_le_succ` in Figure 2a ($\sim$ represents logical negation in Coq). `time` is used to output the duration of the tactic being run, along with its regular output. The tactic `abduce` is designed to fail when it successfully finds the abducts and to print the abducts as part of its error message. The call to the tactic is commented out in the figure. We report its output in a comment as well. An alternative way to view this tactic is presented in Figure 2b. The SMT solver fails to prove the goal as given, but the abduct returned by the `abduce` tactic suggests that all the user needs to do in this case is to unfold the definition of `Z.succ`.

Admittedly, this simple example may not seem very compelling since the user might have guessed from the start that the definition of `Z.succ` is needed for the SMT solver to prove the goal. Moreover, there is an alternative automated solution provided by Sniper [11] whose `snipe` tactic is able to identify function definitions relevant to the goal and send them to the SMT solver. However, for more complicated functions, providing hypotheses capturing relevant

[1] Instructions and resources needed to reproduce our experiments can be found at `https://homepage.divms.uiowa.edu/~viswanathn/lpar23/`

[2] A more principled experiment would use a less stringent notion of easily provable formula.

```
Lemma Znot_le_succ n : ~ Z.succ n <= n.
Proof. (* time abduce 1. *)
     (* The solver finds the goal to be invalid; the abduce call runs
        for 1.072 secs and returns the abduct 1 + n <= (Z.succ n) *)
     assert (1 + n <= (Z.succ n)). { unfold Z.succ. smt. } smt.
Qed.
```

(a) Example goal proven using `smt` and `abduce`

```
Lemma Znot_le_succ n : ~ Z.succ n <= n.
Proof. (* time abduce 1. *) unfold Z.succ. smt. Qed.
```

(b) An alternative interaction with abduction

Figure 2: Interactions with SMTCoq using the `abduce` tactic.

Goals	**Invalid goals**	`smt` successes	`abduce` successes	**Timeouts**
59	26	33	13	13

Figure 3: Summary of results of using `abduce` in `Zorder`

properties of the function, as in the case of function `f` from Section 3, may be more effective than providing their definitions since proving such properties in the external prover may require inductive reasoning, something SMT solvers are not generally capable of. So the `abduce` tactic can be seen as a complement to `snipe` in helping the user prove goals. Although we allowed the tactic 20 seconds to find a useful abduct, all 14 successful calls were made within 8 seconds. In fact, 9 of them took less than 2 seconds. Note that there were 13 successful goals but 14 invocations of `abduce` because one of the goals required two calls, one for `Z.succ` and one for `Z.pred`.

Using the same test set, we also confirmed some of our hypotheses about the default grammar to provide, and the configuration with which to call the abduction solver. The first was to remove logical disjunction and the if-than-else (ITE) operator from the grammar. Such operators are not crucial since the user can recover disjunctive information by asking for more than one abduct. We found that eliminating these operators did yield more successful abducts. Second, we tested the ability of cvc5's abduction solver to generate conjunctive solutions quickly through unsat-core learning [33]. We found that, although the solver was much faster in generating solutions with this configuration, in almost all cases, at least one of the conjuncts was too specific, rendering the entire solution useless as it was not entailed by the Coq context. For instance, with this option enabled, one of the abducts for `Znot_le_succ` from Figure 2a is (`&&` denotes conjunction):

```
n <= (Z.succ n) && (not (Z.succ n) = n) && (Z.succ -2) = n && n = -1
```

We can see that the first conjunct is a useful abduct in isolation, whereas the full conjunction clearly does not hold for the successor function.

6 Conclusion and Future Work

We have extended SMTCoq by adding the interactive tactic `abduce` to its set of proof tactics. When cvc5 fails to prove a goal valid, this tactic presents an alternative to returning (possibly spurious) counterexamples. In such cases, the abductive capabilities of cvc5 can present the user with additional assumptions that would make the goal provable. With tools such as hammers [9, 15] that deal with integrating AR tools into proof assistants, a good hypothesis selection strategy is important to avoid either overloading the AR tools with too many facts, or conversely, supplying it with insufficient facts to prove the goal [1, 30]. With `abduce`, we allow the AR tool to be part of the premise selection process.

We plan to have this tactic available in the next official release of SMTCoq. (It is currently available via a developer branch.) Moving forward, there are many ways in which the interaction with the abduction solver could be improved. Currently, we use a default grammar for abducts that has proved to be efficient from some experimentation with the abduction solver. A combination of allowing grammar selection by the user and using automatic methods to reduce the language generated by the grammar would improve the quality of the generated abducts.

Sending quantified hypotheses to the solver is problematic since SMTCoq has limited support for quantifiers, and because quantified assertions slow the SMT solver down. By producing ground abducts, and relying on (i) manual quantifier instantiation of lemmas by the user; and (ii) utilities such as Coq's `Search` vernacular to find the relevant lemmas, the `abduce` tactic offers a way around this issue. While Example 4.1 suggests how this may be done, we aim to test this ability in a larger Coq development setting, where `abduce` can be used within complex proofs, possibly with numerous cases to discharge, and in tandem with other tactics.

References

[1] Jesse Alama, Tom Heskes, Daniel Kühlwein, Evgeni Tsivtsivadze, and Josef Urban. Premise selection for mathematics by corpus analysis and kernel methods. *J. Autom. Reason.*, 52(2):191–213, 2014.

[2] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. Maximal specification synthesis. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 789–801. ACM, 2016.

[3] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8, 2013.

[4] Michael Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A modular integration of sat/smt solvers to Coq through proof witnesses. In *Proceedings of the First International Conference on Certified Programs and Proofs*, CPP'11, page 135–150, Berlin, Heidelberg, 2011. Springer-Verlag.

[5] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[6] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 305–343. Springer International Publishing, Cham, 2018.

[7] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT,*

USA, July 14-20, 2011. Proceedings, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[8] Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[9] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending sledgehammer with SMT solvers. *J. Autom. Reason.*, 51(1):109–128, 2013.

[10] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards QED. *J. Formaliz. Reason.*, 9(1):101–148, 2016.

[11] Valentin Blot, Denis Cousineau, Enzo Crance, Louise Dubois de Prisque, Chantal Keller, Assia Mahboubi, and Pierre Vial. Compositional pre-processing for automated reasoning in dependent type theory. In Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve Zdancewic, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*, pages 63–77. ACM, 2023.

[12] Thomas Bouton, Diego Caminha B. De Oliveira, David Déharbe, and Pascal Fontaine. Verit: An open, trustable and efficient SMT-solver. In *Proceedings of the 22nd International Conference on Automated Deduction*, CADE-22, page 151–156, Berlin, Heidelberg, 2009. Springer-Verlag.

[13] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 289–300. ACM, 2009.

[14] Adam Chlipala and George C. Necula. Cooperative integration of an interactive proof assistant and an automated prover. In *Proceedings of the 6th International Workshop on Strategies in Automated Deduction*, 2006.

[15] Lukasz Czajka and Cezary Kaliszyk. Hammer for Coq: Automation for dependent type theory. *Journal of Automated Reasoning*, 61, 06 2018.

[16] Isil Dillig and Thomas Dillig. Explain: A tool for performing abductive inference. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 684–689. Springer, 2013.

[17] Isil Dillig, Thomas Dillig, and Alex Aiken. Automated error diagnosis using abductive inference. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 181–192. ACM, 2012.

[18] Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. Inductive invariant generation via abductive inference. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 443–456. ACM, 2013.

[19] Isil Dillig, Thomas Dillig, Boyang Li, Kenneth L. McMillan, and Mooly Sagiv. Synthesis of circular compositional program proofs via abduction. *Int. J. Softw. Tools Technol. Transf.*, 19(5):535–547, 2017.

[20] Isil Dillig, Thomas Dillig, Kenneth L. McMillan, and Alex Aiken. Minimum satisfying assignments for SMT. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 394–409. Springer, 2012.

[21] Thomas Dillig, Isil Dillig, and Swarat Chaudhuri. Optimal guard synthesis for memory safety. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 491–507.

Springer, 2014.

[22] Mnacho Echenim and Nicolas Peltier. A superposition calculus for abductive reasoning. *J. Autom. Reason.*, 57(2):97–134, 2016.

[23] Mnacho Echenim, Nicolas Peltier, and Yanis Sellami. A generic framework for implicate generation modulo theories. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*, volume 10900 of *Lecture Notes in Computer Science*, pages 279–294. Springer, 2018.

[24] Mnacho Echenim, Nicolas Peltier, and Yanis Sellami. Ilinva: Using abduction to generate loop invariants. In Andreas Herzig and Andrei Popescu, editors, *Frontiers of Combining Systems - 12th International Symposium, FroCoS 2019, London, UK, September 4-6, 2019, Proceedings*, volume 11715 of *Lecture Notes in Computer Science*, pages 77–93. Springer, 2019.

[25] Mnacho Echenim, Nicolas Peltier, and Sophie Tourret. Prime implicate generation in equational logic. *J. Artif. Intell. Res.*, 60:827–880, 2017.

[26] Fajar Haifani, Patrick Koopmann, Sophie Tourret, and Christoph Weidenbach. Connection-minimal abduction in *EL* via translation to FOL. In Jasmin Blanchette, Laura Kovács, and Dirk Pattinson, editors, *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*, volume 13385 of *Lecture Notes in Computer Science*, pages 188–207. Springer, 2022.

[27] Gérard Huet. The Gallina specification language: A case study. In Rudrapatna Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science*, pages 229–240, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[28] CNRS Inria and contributors. Conversion rules in Coq - Coq documentation. https://coq.github.io/doc/master/refman/language/core/conversion.html.

[29] CNRS Inria and contributors. Library Coq.Zarith.Zorder. https://coq.github.io/doc/v8.13/stdlib/Coq.ZArith.Zorder.html.

[30] Daniel Kühlwein, Twan van Laarhoven, Evgeni Tsivtsivadze, Josef Urban, and Tom Heskes. Overview and evaluation of premise selection techniques for large theory mathematics. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning*, pages 378–392, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[31] Stéphane Lescuyer. *Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq. (Formalisation et developpement d'une tactique reflexive pour la demonstration automatique en coq)*. PhD thesis, University of Paris-Sud, Orsay, France, 2011.

[32] Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In Bruno Woltzenlogel Paleo and David Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical logic and foundations)*. College Publications, January 2015.

[33] Andrew Reynolds, Haniel Barbosa, Daniel Larraz, and Cesare Tinelli. Scalable algorithms for abduction via enumerative syntax-guided synthesis. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I*, volume 12166 of *Lecture Notes in Computer Science*, pages 141–160. Springer, 2020.

[34] Armando Solar-Lezama. The sketching approach to program synthesis. In Zhenjiang Hu, editor, *Programming Languages and Systems*, pages 4–13, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[35] Laurent Théry, Pierre Letouzey, and Georges Gonthier. *Coq*, pages 28–35. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[36] Haiyan Zhu, Thomas Dillig, and Isil Dillig. Automated inference of library specifications for source-sink property verification. In Chung-chieh Shan, editor, *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013.*

Proceedings, volume 8301 of *Lecture Notes in Computer Science*, pages 290–306. Springer, 2013.