EPTCS 210

Proceedings of the

First International Workshop on Hammers for Type Theories

Coimbra, Portugal, July 1, 2016

Edited by: Jasmin Christian Blanchette and Cezary Kaliszyk

Published: 17th June 2016
DOI: 10.4204/EPTCS.210
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Table of Contents i

Preface ii

Generic Automation for the Coq Proof Assistant: Design and Principles 1
Pierre Corbineau

Proof Generation in Propositional Intuitionistic Logic Based upon Automata Theory 2
Aleksy Schubert and Maciej Zielenkiewicz

Extending Nunchaku to Dependent Type Theory 3
Simon Cruanes and Jasmin Christian Blanchette

Goal Translation for a Hammer for Coq (Extended Abstract) 13
Łukasz Czajka and Cezary Kaliszyk

Extending SMTCoq, a Certified Checker for SMT (Extended Abstract) 21
Burak Ekici, Guy Katz, Chantal Keller, Alain Mebsout, Andrew J. Reynolds and Cesare Tinelli

Towards the Integration of an Intuitionistic First-Order Prover into Coq 30
Fabian Kunze

Preface

This volume of EPTCS contains the proceedings of the First Workshop on Hammers for Type Theories (HaTT 2016), held on 1 July 2016 as part of the International Joint Conference on Automated Reasoning (IJCAR 2016) in Coimbra, Portugal.

HOLyHammer for HOL Light and HOL4, Sledgehammer for Isabelle/HOL, and other similar tools can have a huge impact on user productivity. These integrate automatic theorem provers (including SMT solvers) with proof assistants. However, users of proof assistants based on type theories, such as Agda, Coq, Lean, and Matita, currently miss out on this convenience. The expressive, constructive logic is often seen as an insurmountable obstacle, but large developments, including the CompCert compiler, typically postulate the classical axioms and use dependent types sparingly.

The workshop features four regular papers, three regular presentations, and two invited talks by Pierre Corbineau (Verimag, France) and Aleksy Schubert (University of Warsaw, Poland).

We would like to thank the authors for submitting papers of high quality to these proceedings, the program committee and external reviewers for diligently reviewing the submissions, and the local organizers of IJCAR 2016 for their help in organizing HaTT 2016.

6 June 2016

Jasmin Christian Blanchette
Cezary Kaliszyk

Organization

Program Committee

- Jesper Bengtson, IT University of Copenhagen
- Frédéric Besson, Inria
- Jasmin Christian Blanchette (co-chair), Inria & MPII Saarbrücken
- Arthur Charguéraud, Inria
- Leonardo de Moura, Microsoft Research
- Jean-Christophe Filliâtre, CNRS
- Liana Hadarean, Oxford University
- Cătălin Hriţcu, Inria
- Cezary Kaliszyk (co-chair), University of Innsbruck
- Chantal Keller, Université Paris-Sud
- Assia Mahboubi, Inria
- Claudio Sacerdoti Coen, University of Bologna
- Laurent Théry, Inria
- Cesare Tinelli, The University of Iowa
- Josef Urban, Czech Technical University in Prague

External Reviewers

- Chad Brown, Czech Technical University in Prague
- Stéphane Graham-Lengrand, École polytechnique

Generic Automation for the Coq Proof Assistant: Design and Principles

Pierre Corbineau
Verimag, Université Grenoble Alpes, France

Proof-editing in the Coq proof assistant is conducted using a wide variety of procedures called tactics. Several of these tactics host automated proof-search procedures addressing generic or specific logical problems.

Generic automation tactics try to provide help without relying on the existence of a specific theory or axiom, whereas specialised tactics address logical problems expressed in specific object-level theories such as linear arithmetic, rings, fields, . . .

In this talk, we will focus on several examples of generic automation procedures. We will first describe how they work, and then show how they can interact with each other and other Coq features. Finally we will discuss their usefulness and weaknesses, and the pertinence of the generic approach.

J.C. Blanchette and C. Kaliszyk:
Hammers for Type Theories (HaTT'16)
EPTCS 210, 2016, pp. 1–1, doi:10.4204/EPTCS.210.1

Proof Generation in Propositional Intuitionistic Logic Based upon Automata Theory

Aleksy Schubert and Maciej Zielenkiewicz
Institute of Informatics, University of Warsaw, Poland

The process of proof construction in constructive logics corresponds very naturally to runs of a certain kind of automata. This idea was used as a presentation method in recent book on lambda calculi with types by Barendregt, Dekkers, and Statman. However, this idea also gives the opportunity to bring the refined techniques of automata theory to proof generation in constructive logics.

In the talk a model of automata will be presented that can handle proof construction in full intuitionistic first-order logic. The automata are constructed in such a way that any successful run corresponds directly to a cut-free proof in the logic. This makes it possible to discuss formal languages of proofs and the closure properties of the automata and their connections with the traditional logical connectives.

It turns out that one can devise two natural notions of automata. The first one that is able to recognise the language of all the normal forms and one that is able to recognise only proofs in so called total discharge form. This difference will be discussed during the talk as well as a number of decision problems around the automata. Of course, the emptiness problem for automata in their most general presentation is undecidable, but a number of interesting decidable cases will be presented during the talk.

The languages of proofs discussed so far are languages of cut-free proofs. However, proofs in proof assistants are usually constructed with help of lemmas and the cut rule is used there extensively. An automata theoretic approach to proofs with cuts will also be discussed during the talk.

J.C. Blanchette and C. Kaliszyk:
Hammers for Type Theories (HaTT'16)
EPTCS 210, 2016, pp. 2–2, doi:10.4204/EPTCS.210.2

Extending Nunchaku to Dependent Type Theory

Simon Cruanes
Inria Nancy – Grand Est, France
simon.cruanes@inria.fr

Jasmin Christian Blanchette
Inria Nancy – Grand Est, France
Max-Planck-Institut für Informatik, Saarbrücken, Germany
jasmin.blanchette@inria.fr

Nunchaku is a new higher-order counterexample generator based on a sequence of transformations from polymorphic higher-order logic to first-order logic. Unlike its predecessor Nitpick for Isabelle, it is designed as a stand-alone tool, with frontends for various proof assistants. In this short paper, we present some ideas to extend Nunchaku with partial support for dependent types and type classes, to make frontends for Coq and other systems based on dependent type theory more useful.

1 Introduction

In recent years, we have seen the emergence of "hammers"—integrations of automatic theorem provers in proof assistants, such as Sledgehammer and HOLyHammer [7]. As useful as they might be, these tools are mostly helpless in the face of an invalid conjecture. Novices and experts alike can enter invalid formulas and find themselves wasting hours (or days) on an impossible proof; once they identify and correct the error, the proof is often easy. To discover flaws early, some proof assistants include counterexample generators to debug putative theorems or specific subgoals in an interactive proof. When formalizing algebraic results in Isabelle/HOL, Guttmann et al. [21] remarked that

> Counterexample generators such as Nitpick complement the ATP [automatic theorem proving] systems and allow a proof and refutation game which is useful for developing and debugging formal specifications.

Nunchaku is a new fully automatic counterexample generator for higher-order logic (simple type theory) designed to be integrated into several proof assistants. It supports polymorphism, (co)algebraic datatypes, (co)recursive functions, and (co)inductive predicates. The tool is undergoing considerable development, and we expect that it will soon be sufficiently useful to mostly replace Nitpick [8] for Isabelle/HOL. The source code is freely available online.[1]

A Nunchaku frontend in a proof assistant provides a **nunchaku** command that can be invoked on conjectures to debug them. It collects the relevant definitions and axioms, translates them to higher-order logic along with the negated conjecture, invokes Nunchaku, and translates any model found to higher-order logic. We have developed a frontend for Isabelle/HOL [32]. We are also working on a frontend for the set-theoretic TLA$^+$ Proof System [18] and plan to develop frontends for other proof assistants.

This short paper discusses some of the issues that must be addressed to make frontends for Coq [4] and other systems based on dependent type theory (e.g., Agda, Lean, and Matita) applicable beyond their simple type theory fragment. We plan to elaborate and implement the approach in a Coq frontend, as part of the Inria technological development action "Contre-exemples utilisables par Isabelle et Coq."

[1] https://github.com/nunchaku-inria/nunchaku

J.C. Blanchette and C. Kaliszyk:
Hammers for Type Theories (HaTT'16)
EPTCS 210, 2016, pp. 3–12, doi:10.4204/EPTCS.210.3

2 Overview of Nunchaku

Nunchaku is the spiritual successor to Nitpick but is designed as a stand-alone OCaml program, with its own input language. Whereas Nitpick generates a succession of finite problems for increasing cardinalities, Nunchaku translates its input to one first-order logic program that targets the finite model finding fragment of CVC4 [2], a state-of-the-art SMT (satisfiability modulo theories) solver. Using CVC4 as a backend allows Nunchaku to reason efficiently about arithmetic constraints and (co)algebraic datatypes [36] and to detect unsatisfiability in addition to satisfiability. Support for other backends, including Kodkod [43] (used by Nitpick) and Paradox [16], is in the works. We also plan to integrate backends based on code execution and narrowing, as provided by Quickcheck for Isabelle/HOL [10], to further increase the likelihood of finding counterexamples.

Nunchaku's input syntax is inspired by that of proof assistants based on higher-order logic (e.g., Isabelle/HOL) and by typed functional programming languages (e.g., OCaml). The following problem gives a flavor of the syntax:

```
data nat := Zero | Suc nat.

pred even : nat → prop :=
  even Zero;
  ∀n. odd n ⇒ even (Suc n)
and odd : nat → prop :=
  ∀n. even n ⇒ odd (Suc n).

val m : nat.

goal even m ∧ ¬ (m = Zero).
```

The problem defines a datatype (nat) and two mutually recursive inductive predicates (even and odd), it declares a constant *m*, and it specifies a goal to satisfy ("*m* is even and nonzero"). For counterexample generation, the negated conjecture must be specified as the Nunchaku goal. For the example above, Nunchaku outputs the model

```
val even := λ(n : nat). IF n = Zero ∨ n = Suc (Suc Zero) THEN true ELSE ?__ n.
val odd  := λ(n : nat). IF n = Suc Zero THEN true ELSE ?__ n.
val m    := Suc (Suc Zero).
```

The output is a finite fragment of an infinite model. The notation '?__' is a placeholder for an unknown value or function. To most users, the interesting part is the interpretation of *m*; but it may help to inspect the partial model of even and odd to check if they have the expected semantics.

Given an input problem, Nunchaku parses it before applying a sequence of translations, each reducing the distance to the target fragment. In our example, the predicates even and odd are translated to recursive functions, then the recursive functions are encoded to allow finite model finding, by limiting their domains to an unspecified finite fragment. If Nunchaku finds a model of the goal, it translates it back to the input language, reversing each phase.

The translation pipeline includes the following phases (adapted from a previous paper [37]):

Type inference infers types and checks definitions;

Type skolemization replaces $\exists\alpha.\ \varphi[\alpha]$ with $\varphi[\tau]$, where τ is a fresh type;

Monomorphization specializes polymorphic definitions on their type arguments and removes unused definitions;

Elimination of equations translates multiple-equation definitions of recursive functions into a single nested pattern matching;

Specialization creates instances of functions with static arguments (i.e., an argument that is passed unchanged to all recursive calls);

Polarization specializes predicates into a version used in positive positions and a version used in negative positions;

Unrolling adds a decreasing argument to possibly ill-founded predicates;

Skolemization introduces Skolem symbols for term variables;

Elimination of (co)inductive predicates recasts a multiple-clause (co)inductive predicate definition into a recursive equation;

λ-Lifting eliminates λ-abstractions by introducing named functions;

Elimination of higher-order constructs substitutes SMT-style arrays for higher-order functions;

Elimination of recursive functions encodes recursive functions to allow finite model finding;

Elimination of pattern matching rewrites pattern-matching expressions using datatype discriminators and selectors;

Elimination of assertions encodes ASSERTING operator using logical connectives;

CVC4 invocation runs CVC4 to obtain a model.

Although our examples use datatypes and well-founded (terminating) recursion, Nunchaku also supports codatatypes and productive corecursion. In addition to finite values, cyclic α-regular codatatype values can arise in models (e.g., the infinite stream $1, 0, 9, 0, 9, 0, 9, \ldots$) [36].

While most of Nunchaku's constructs are fairly conventional, one is idiosyncratic and plays a key role in the translations described here: The ASSERTING operator, written t ASSERTING φ, attaches a formula φ—the *guard*—to a term t. It allows the evaluation of t only if φ is satisfied. The construct is equivalent to IF φ THEN t ELSE UNREACHABLE in other specification languages (e.g., the Haskell Bounded Model Checker [14]). Internally, Nunchaku pulls the ASSERTING guards outside of terms into the surrounding logical context, carefully distinguishing positive and negative contexts.

Nunchaku can only find classical models with functional extensionality, which are a subset of the models of constructive type theory. This means the tool, together with the envisioned encoding, will be sound but incomplete: All counterexamples will be genuine, but no counterexamples will be produced for classical theorems that do not hold intuitionistically. We doubt that this will seriously impair the usefulness of Nunchaku in practice.

3 Encoding Recursive Functions

When using finite model finding to generate counterexamples, a central issue is to translate infinite positive universal quantifiers in a sound way. The situation is hopeless for arbitrary axioms or hypotheses, but infinite quantifiers arising in well-behaved definitions can be encoded soundly. We describe Nunchaku's encoding of recursive functions [37], because it is one of the most crucial phases of the translation pipeline and it illustrates the ASSERTING construct in a comparatively simple setting.

Consider the following factorial example:

rec $\mathsf{fact} : \mathsf{int} \to \mathsf{int} :=$
  $\forall n.\ \mathsf{fact}\ n = (\textsc{if}\ n \leq 0\ \textsc{then}\ 1\ \textsc{else}\ n * \mathsf{fact}\ (n-1)).$

val $m : \mathsf{int}.$

goal $\mathsf{fact}\ m > 100.$

(We conveniently assume that Nunchaku has a standard notion of integer arithmetic, as provided by its backend CVC4.) The encoding restricts quantification on fact's domain to an unspecified, but potentially finite, type α_{fact} that is isomorphic to a subset of fact's argument type and introduces projections $\gamma_{\mathsf{fact}} : \alpha_{\mathsf{fact}} \to \mathsf{int}$ and ASSERTING guards throughout the problem, as follows:

val $\mathsf{fact} : \mathsf{int} \to \mathsf{int}.$

axiom $\forall(a : \alpha_{\mathsf{fact}}).\ \mathsf{fact}\ (\gamma_{\mathsf{fact}}\ a) = (\textsc{if}\ \gamma_{\mathsf{fact}}\ a \leq 0\ \textsc{then}\ 1$
  $\textsc{else}\ \gamma_{\mathsf{fact}}\ a * (\mathsf{fact}\ (\gamma_{\mathsf{fact}}\ a - 1)\ \textsc{asserting}\ \exists(b : \alpha_{\mathsf{fact}}).\ \gamma_{\mathsf{fact}}\ b = \gamma_{\mathsf{fact}}\ a - 1)).$

val $m : \mathsf{int}.$

goal $(\mathsf{fact}\ m\ \textsc{asserting}\ \exists(b : \alpha_{\mathsf{fact}}).\ \gamma_{\mathsf{fact}}\ b = m) > 100.$

The guards are propagated outward until they reach a polarized context, at which point they can be asserted using standard connectives:

val $\mathsf{fact} : \mathsf{int} \to \mathsf{int}.$

axiom $\forall(a : \alpha_{\mathsf{fact}}).\ \mathsf{fact}\ (\gamma_{\mathsf{fact}}\ a) = (\textsc{if}\ \gamma_{\mathsf{fact}}\ a \leq 0\ \textsc{then}\ 1\ \textsc{else}\ \gamma_{\mathsf{fact}}\ a * \mathsf{fact}\ (\gamma_{\mathsf{fact}}\ a - 1)$
  $\wedge \neg\, \gamma_{\mathsf{fact}}\ a \leq 0 \wedge \exists(b : \alpha_{\mathsf{fact}}).\ \gamma_{\mathsf{fact}}\ b = \gamma_{\mathsf{fact}}\ a - 1).$

val $m : \mathsf{int}.$

goal $\mathsf{fact}\ m > 100 \wedge \exists(b : \alpha_{\mathsf{fact}}).\ \gamma_{\mathsf{fact}}\ b = m.$

The guards ensure that the result of recursive function calls is inspected (i.e., influences the truth value of the problem) only if the arguments are in the subset α_{fact}, for which the function is axiomatized.

4 Encoding Dependent Datatypes

We propose an extension to Nunchaku's type system with a simple flavor of dependent types. We assume a finite hierarchy of sorts. A Coq frontend would need to truncate the problem's hierarchy of universes. Our encoding is similar to the one proposed by Jacobs and Melham [24]. We, too, erase dependent parameters from types and use additional predicates to enforce constraints that would be lost otherwise—with the addition of dependent (co)datatypes. In (co)datatypes, we allow term parameters (such as the length of a list, of type nat) to occur as uniform parameters or as indices (i.e., each constructor can have a different value for this parameter), but type parameters should occur uniformly. We only forbid polymorphic recursion (type indices), because it is not compatible with the monomorphization step Nunchaku currently relies on.

In general, we consider dependent (co)datatype definitions of the form

$$
\begin{array}{l}
(\textbf{co})\textbf{data}\ \tau\ \overline{x}\ \overline{\alpha} := \\
\quad c_1 : \overline{\sigma^1} \to \tau\ \overline{t^1}\ \overline{\alpha} \\
\quad \vdots \\
\mid\ c_k : \overline{\sigma^k} \to \tau\ \overline{t^k}\ \overline{\alpha}
\end{array}
$$

where $\overline{x} := (x_i)_{i=1}^{m}$ is the tuple of term variables on which τ depends, $\overline{\alpha} := (\alpha_i)_{i=1}^{n}$ is the tuple of type variables, the types $(\sigma_i^k)_{i=1}^{\mathrm{arity}(c_k)}$ are the types of the arguments of the kth constructor, and the terms

$\overline{t^k} := (t_i^k)_{i=1}^m$ are the term arguments of the kth constructor's return type. More elaborate definitions, such as those interleaving type and term parameters in more intricate ways, are beyond the scope of this approach. We are aiming for a practical balance between expressiveness and ease of implementation.

Let $\tau'\ \overline{\alpha}$ be the encoding of τ where all term arguments have been removed. We introduce a predicate inv_τ, defined inductively (if τ is a datatype) or coinductively (if τ is a codatatype), that enforces the correspondence between $\overline{x}$ and $\tau'\ \overline{\alpha}$:

$$(\textbf{co})\textbf{pred}\ \mathsf{inv}_\tau : \Pi\overline{\alpha}.\ \overline{\alpha} \rightarrow \tau'\ \overline{\alpha} \rightarrow \mathsf{prop} :=$$
$$\bigwedge_{i=1}^{k} \left[\begin{array}{l} \forall\overline{x}\ (y_1 : a_1^i) \ldots (y_k : a^k_{\text{arity}(c_k)}). \\ \quad \left(\bigwedge_{j=1, y_j^k : \tau}^{\text{arity}(c_k)} \mathsf{inv}_\tau\ \overline{\alpha}\ y_j^k \right) \Rightarrow \mathsf{inv}_\tau\ \overline{\alpha}\ \ (c_k\ \overline{\alpha}\ \overline{y}) \end{array} \right].$$

The predicate inv_τ has one clause per constructor c_k of τ, which ensures that if the invariant holds for every argument $(y_j)_{j=1}^{\text{arity}(c_k)}$ of c_k that has type τ (a recursive instance of τ), it also holds for $c_k\ \overline{\alpha}\ \overline{y}$.

When encoding terms, we process binders on dependently-typed variables recursively as follows: $\forall v : \tau\ \bar{t}\ \bar{u}.\ \varphi$ becomes $\forall v : \tau'\ \bar{u}.\ \mathsf{inv}_\tau\ \bar{t}\ v \Rightarrow \varphi$, and a function $\lambda(x : \tau\ \bar{t}\ \bar{u}).\ v$ is translated to $\lambda(x : \tau'\ \bar{u}).\ (v\ \text{ASSERTING}\ \mathsf{inv}_\tau\ \bar{t}\ x)$.

Functions whose type depends on terms remain parameterized by these terms after the translation, but their definition specifies a precondition that links the term parameters to the encoded dependent type. The use of ASSERTING to encode the precondition ensures that the function is evaluated only if the condition is met, irrespective of the context (positive, negative, or unpolarized) of the function. Finally, some specific constructs such as equality (in Coq, equality is a dependent datatype) are translated directly into Nunchaku counterparts.

As an example, consider the type of vectors of length n. Here, n is an index, and α is a uniform type parameter:

$$\begin{array}{l} \textbf{data}\ \mathsf{vec} : \mathsf{nat} \rightarrow \mathsf{type} \rightarrow \mathsf{type} := \\ \quad \mathsf{nil}\alpha : \mathsf{vec}\ 0\ \alpha \\ \ \mid \forall(n : \mathsf{nat})\ (x : \alpha)\ (l : \mathsf{vec}\ n\ \alpha).\ \mathsf{cons}\ \alpha\ x\ l : \mathsf{vec}\ \ (\mathsf{S}\ n)\ \ \alpha. \end{array}$$

The encoded type vec' corresponds to the datatype of finite lists, and the invariant is

$$\begin{array}{l} \textbf{pred}\ \mathsf{inv}_{\mathsf{vec}} : \mathsf{nat} \rightarrow \mathsf{vec}'\ \alpha \rightarrow \mathsf{prop} := \\ \quad \mathsf{inv}_{\mathsf{vec}}\ 0\ (\mathsf{nil}\ \alpha) \\ \ \mid \forall(n : \mathsf{nat})\ (x : \alpha)\ (l : \mathsf{vec}'\ \alpha).\ \mathsf{inv}_{\mathsf{vec}}\ n\ l \Rightarrow \mathsf{inv}_{\mathsf{vec}}\ \ (\mathsf{S}\ n)\ \ (\mathsf{cons}\ \alpha\ x\ l). \end{array}$$

A formula $\forall(v : \mathsf{vec}\ n\ \tau).\ \varphi$ is translated to $\forall(v : \mathsf{vec}'\ \tau).\ \mathsf{inv}_{\mathsf{vec}}\ n\ v \Rightarrow \varphi$. A function $\lambda(v : \mathsf{vec}\ n\ \tau).\ t$ is translated to $\lambda(v : \mathsf{vec}'\ \tau).\ (t\ \text{ASSERTING}\ \mathsf{inv}_{\mathsf{vec}}\ n\ v)$.

Thus, the function returning the length of a vector, $\lambda n\ (l : \mathsf{vec}\ n\ \alpha).\ n$, becomes

$$\lambda n\ (l : \mathsf{vec}'\ \alpha).\ (n\ \text{ASSERTING}\ \mathsf{inv}_{\mathsf{vec}}\ n)$$

The append function $\lambda m\, n\ (l_1 : \mathsf{vec}\ m\ \alpha)\ (l_2 : \mathsf{vec}\ n\ \alpha).\ t$ (omitting the body) becomes

$$\lambda m\, n\ (l_1 : \mathsf{vec}'\ \alpha)\ (l_2 : \mathsf{vec}'\ \alpha).\ (t\ \text{ASSERTING}\ \mathsf{inv}_{\mathsf{vec}}\ m\ l_1 \wedge \mathsf{inv}_{\mathsf{vec}}\ n\ l_2)$$

And the mult function that multiplies two matrices, $\lambda m\, n\, k\ (A : \mathsf{matrix}\ m\ n)\ (B : \mathsf{matrix}\ n\ k).\ t$, returning a value of type $\mathsf{matrix}\ m\ k$, becomes

$$\lambda m\ n\ k\ (A : \mathsf{matrix}')\ (B : \mathsf{matrix}').\ (t\ \text{ASSERTING}\ \mathsf{inv}_{\mathsf{matrix}}\ m\ n\ A \wedge \mathsf{inv}_{\mathsf{matrix}}\ n\ k\ B)$$

5 Encoding Dependent Records and Type Classes

Type classes are a powerful tool for abstraction in Coq, Isabelle/HOL, and other proof assistants [41,45]. However, in dependently typed proofs assistants such as Coq, they are usually encoded as dependent records combining types, values, and proofs. We assume that type classes have been explicitly resolved by the frontend's type inference and focus on their representation as a record of values and propositions. Consider the following example from basic algebra:

$$\begin{array}{l}\textbf{class}\ \mathsf{monoid}\ a\ \textbf{where}\\ \quad \mathsf{e} : a\\ \quad \mathsf{op} : a \to a \to a\\ \quad \mathsf{left\_neutral} : \forall x.\ \mathsf{op}\ \mathsf{e}\ x = x\\ \quad \mathsf{assoc} : \forall x\,y\,z.\ \mathsf{op}\ (\mathsf{op}\ x\,y)\ z = \mathsf{op}\ x\ (\mathsf{op}\ y\,z).\end{array}$$

This definition of monoids can be encoded in a straightforward way as a dependent record—that is, a datatype with a single four-argument constructor. The encoding from Section 4 could then be applied. Here, we propose a more specific encoding that avoids introducing an inductive predicate $\mathsf{inv}_{\mathsf{monoid}}$. This transformation does not use dependent types, and its result still contains the required invariants of each type class, thereby requiring models to satisfy them.

Following our proposed scheme, a type class is translated into a nondependent datatype with one constructor whose arguments are the data fields (e.g., e and op for monoid). The proofs of the axioms can be erased, since they serve no purpose for model finding, and the additional properties $\mathsf{left\_neutral}$ and assoc are directly inserted at appropriate places in the problem.

The definition of monoid is translated to

$$\mathsf{inst}_{\mathsf{monoid}} : \Pi a.\ a \to (a \to a \to a) \to \mathsf{monoid}\ a.$$

$$\begin{array}{l}\textbf{pred}\ \mathsf{left\_neutral}_{\mathsf{monoid}} : \Pi a.\ \mathsf{monoid}\ a \to \mathsf{prop} :=\\ \quad \forall e\ op.\ (\forall x.\ op\ e\ x = x) \Rightarrow \mathsf{left\_neutral}_{\mathsf{monoid}}\ a\ (\mathsf{inst}_{\mathsf{monoid}}\ a\ e\ op).\end{array}$$

$$\begin{array}{l}\textbf{pred}\ \mathsf{assoc}_{\mathsf{monoid}} : \Pi a.\ \mathsf{monoid}\ a \to \mathsf{prop} :=\\ \quad \forall e\ op.\ (\forall x\,y\,z.\ op\ (op\ x\,y)\ z = op\ x\ (op\ y\,z)) \Rightarrow \mathsf{assoc}_{\mathsf{monoid}}\ a\ (\mathsf{inst}_{\mathsf{monoid}}\ a\ e\ op).\end{array}$$

A function definition

$$\begin{array}{l}\textbf{rec}\ \mathsf{f} : \Pi a.\ \mathsf{monoid}\ a \Rightarrow a \to \tau :=\\ \quad \forall (x : a).\ \mathsf{f}\ x = t.\end{array}$$

is translated to

$$\begin{array}{l}\textbf{rec}\ \mathsf{f} : \Pi a.\ \mathsf{monoid}\ a \to a \to \tau :=\\ \quad \forall (x : a).\ \mathsf{f}\ x = (t\ \textsc{asserting}\ \mathsf{left\_neutral}_{\mathsf{monoid}}\ a \wedge \mathsf{assoc}_{\mathsf{monoid}}\ a).\end{array}$$

In a proof assistant, users must explicitly register types as instances of type classes. For example, registering nat as a monoid instance might involve some syntax such as

$$\begin{array}{l}\textbf{instance}\ \mathsf{monoid}\ \mathsf{nat}\ \textbf{where}\\ \quad \mathsf{e} = 0\\ \quad \mathsf{op} = (+)\\ \quad \mathsf{left\_neutral} = \langle \text{proof of } \mathsf{left\_neutral} \rangle\\ \quad \mathsf{assoc} = \langle \text{proof of } \mathsf{assoc} \rangle.\end{array}$$

These would not have to be specified to Nunchaku; in a semantic setting, any type that satisfies the type class axioms would be considered a member of the type class. (For essentially the same reason, only definitions and axioms need to be specified in Nunchaku problems, and not derived lemmas.) Nonetheless, it might be more efficient to provide the instantiations to Nunchaku, so that it can eliminate true conditions such as $\mathsf{left\_neutral}_{\mathsf{monoid}}\ \mathsf{nat} \wedge \mathsf{assoc}_{\mathsf{monoid}}\ \mathsf{nat}$ that can arise as a result of its monomorphization phase.

6 Related Work

There are many competing approaches to refuting logical formulas. The main ones are *finite model finding* and *code execution*. Alternatives include infinite model generation [11], counterexample-producing decision procedures [13], model checking [17], and saturations [1].

Finite model finding consists of enumerating all potential finite models, starting with a cardinality of one for the domains. Some model finders explore the search space directly; FINDER [40], SEM [46], Alloy's precursor [22], and Mace versions 3 and 4 [30] are of this type. Other tools reduce the problem to propositional satisfiability and invoke a SAT solver; these include early versions of Mace (or MACE) [31], Paradox [16], Kodkod [43] and its frontend Alloy [23], and FM-Darwin [3]. Finally, some theorem provers implement finite model finding on top of their proof calculus; this is the case for KIV [35], iProver [25], and CVC4 [38]. To make finite model finding more useful, techniques have been developed to search for partial fragments of infinite models [6, 19, 26, 37, 42].

The idea with code execution is to generate test inputs and evaluate the goal, seen as a functional program. For Haskell, QuickCheck [15] generates random inputs, SmallCheck [39] systematically enumerates inputs starting with small ones, and Lazy SmallCheck [39] relies on narrowing to avoid evaluating irrelevant subterms. A promising combination of bounded model checking and narrowing is implemented in HBMC, the Haskell Bounded Model Checker [14].

In proof assistants, Refute [44] and Nitpick [8] for Isabelle/HOL are based on finite model finding. QuickCheck-like systems have been developed for Agda [20], Isabelle/HOL [10], PVS [33], FoCaLiZe [12], and now Coq with QuickChick [34]. Agsy for Agda [27] employs narrowing. Isabelle's Quickcheck combines random testing, bounded exhaustive testing, and narrowing in one tool [10]. Finally, ACL2 [29] combines random testing and theorem proving.

Our experience with Isabelle is that Nitpick and Quickcheck have complementary strengths and weaknesses [5, Section 3.6] and that it would be a mistake to rely on a single strategy. For example, debugging the axiomatic specification of the C++ memory model [9] was a heavy combinatorial task where Nitpick's SAT solving excelled, whereas for the formalization of a Java-like language [28] it made more sense to develop an executable specification and invoke Quickcheck. Nunchaku currently stands firmly in the finite model finding world, but we plan to develop an alternative translation pipeline to generate Haskell code and invoke QuickCheck, SmallCheck, Lazy SmallCheck, and HBMC.

7 Conclusion

Nunchaku supports polymorphic higher-order logic by a series of transformations that yield a first-order problem suitable for finite model finding. This paper introduced further transformations that extend the translation pipeline to support dependent types and type classes as found in Coq and similar systems. More work is necessary to fully specify these transformations, prove them correct, and implement them. We plan an integration in Coq but will happily collaborate with the developers of other systems to build further frontends; in particular, we are already in contact with the developers of Lean, a promising new proof assistant based on type theory.

We generally contend that too much work has gone into engineering the individual proof assistants, and too little into developing compositional methods and tools with a broad applicability across systems. Nunchaku is our attempt at changing this state of affairs for counterexample generation.

Acknowledgment We are grateful to the anonymous reviewers for making many useful comments and suggestions and for pointing to related work. We also thank Mark Summerfield, who suggested many textual improvements. Cruanes is supported by the Inria technological development action "Contre-exemples utilisables par Isabelle et Coq" (CUIC). Nunchaku would not exist today had it not been for the foresight and support of Stephan Merz, Andrew Reynolds, and Cesare Tinelli.

References

[1] Leo Bachmair & Harald Ganzinger (2001): *Resolution theorem proving*. In Alan Robinson & Andrei Voronkov, editors: *Handbook of Automated Reasoning*, I, Elsevier, pp. 19–99.

[2] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds & Cesare Tinelli (2011): *CVC4*. In Ganesh Gopalakrishnan & Shaz Qadeer, editors: *CAV 2011*, *LNCS* 6806, Springer, pp. 171–177, doi:10.1007/978-3-642-22110-1_14.

[3] Peter Baumgartner, Alexander Fuchs, Hans de Nivelle & Cesare Tinelli (2009): *Computing finite models by reduction to function-free clause logic*. *J. Applied Logic* 7(1), pp. 58–74, doi:10.1016/j.jal.2007.07.005.

[4] Yves Bertot & Pierre Castéran (2004): *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer.

[5] Jasmin Christian Blanchette (2012): *Automatic Proofs and Refutations for Higher-Order Logic*. Ph.D. thesis, Technische Universität München.

[6] Jasmin Christian Blanchette (2013): *Relational analysis of (co)inductive predicates, (co)inductive datatypes, and (co)recursive functions*. *Softw. Qual. J.* 21(1), pp. 101–126, doi:10.1007/s11219-011-9148-5.

[7] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson & Josef Urban (2016): *Hammering towards QED*. *J. Formal. Reasoning* 9(1), pp. 101–148, doi:10.6092/issn.1972-5787/4593.

[8] Jasmin Christian Blanchette & Tobias Nipkow (2010): *Nitpick: A counterexample generator for higher-order logic based on a relational model finder*. In Matt Kaufmann & Lawrence C. Paulson, editors: *ITP 2010*, *LNCS* 6172, Springer, pp. 131–146, doi:10.1007/978-3-642-14052-5_11.

[9] Jasmin Christian Blanchette, Tjark Weber, Mark Batty, Scott Owens & Susmit Sarkar (2011): *Nitpicking C++ concurrency*. In: *PPDP 2011*, ACM, pp. 113–124, doi:10.1145/2003476.2003493.

[10] Lukas Bulwahn (2012): *The new Quickcheck for Isabelle: Random, exhaustive and symbolic testing under one roof*. In Chris Hawblitzel & Dale Miller, editors: *CPP 2012*, *LNCS* 7679, Springer, pp. 92–108, doi:10.1007/978-3-642-35308-6_10.

[11] Ricardo Caferra, Alexander Leitsch & Nicolas Peltier (2004): *Automated Model Building*. *Applied Logic* 31, Springer.

[12] Matthieu Carlier, Catherine Dubois & Arnaud Gotlieb (2012): *A first step in the design of a formally verified constraint-based testing tool: FocalTest*. In Achim D. Brucker & Jacques Julliand, editors: *TAP 2012*, *LNCS* 7305, Springer, pp. 35–50, doi:10.1007/978-3-642-30473-6_5.

[13] Amine Chaieb & Tobias Nipkow (2008): *Proof synthesis and reflection for linear arithmetic*. *J. Autom. Reasoning* 41(1), pp. 33–59, doi:10.1007/s10817-008-9101-x.

[14] Koen Claessen (2016): Private communication.

[15] Koen Claessen & John Hughes (2000): *QuickCheck: A lightweight tool for random testing of Haskell programs*. In: *ICFP '00*, ACM, pp. 268–279, doi:10.1145/357766.351266.

[16] Koen Claessen & Niklas Sörensson (2003): *New techniques that improve MACE-style model finding*. In: *MODEL*.

[17] Edmund M. Clarke, Jr., Orna Grumberg & Doron A. Peled (1999): *Model Checking*. MIT Press.

[18] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts & Hernán Vanzetto (2012): *TLA$^+$ proofs*. In Dimitra Giannakopoulou & Dominique Méry, editors: *FM 2012*, *LNCS* 7436, Springer, pp. 147–154, doi:10.1007/978-3-642-32759-9_14.

[19] Andriy Dunets, Gerhard Schellhorn & Wolfgang Reif (2010): *Automated flaw detection in algebraic specifications*. *J. Autom. Reasoning* 45(4), pp. 359–395, doi:10.1007/s10817-010-9166-1.

[20] Peter Dybjer, Qiao Haiyan & Makoto Takeyama (2003): *Combining testing and proving in dependent type theory*. In David A. Basin & Burkhart Wolff, editors: *TPHOLs 2003*, *LNCS* 2758, Springer, pp. 188–203, doi:10.1007/10930755_12.

[21] Walter Guttmann, Georg Struth & Tjark Weber (2011): *Automating algebraic methods in Isabelle*. In Shengchao Qin & Zongyan Qiu, editors: *ICFEM 2011*, *LNCS* 6991, Springer, pp. 617–632, doi:10.1007/978-3-642-24559-6_41.

[22] Daniel Jackson (1996): *Nitpick: A checkable specification language*. In: *FMSP '96*, pp. 60–69.

[23] Daniel Jackson (2006): *Software Abstractions: Logic, Language, and Analysis*. MIT Press.

[24] Bart Jacobs & Tom Melham (1993): *Translating dependent type theory into higher order logic*. In M. Bezem & J.F. Groote, editors: *TLCA 1993*, *LNCS* 664, Springer, pp. 209–229, doi:10.1007/BFb0037108.

[25] Konstantin Korovin (2013): *Non-cyclic sorts for first-order satisfiability*. In Pascal Fontaine, Christophe Ringeissen & Renate A. Schmidt, editors: *FroCoS 2013*, *LNCS* 8152, Springer, pp. 214–228, doi:10.1007/978-3-642-40885-4_15.

[26] Viktor Kuncak & Daniel Jackson (2005): *Relational analysis of algebraic datatypes*. In Michel Wermelinger & Harald Gall, editors: *ESEC/FSE 2005*, ACM, pp. 207–216, doi:10.1145/1081706.1081740.

[27] Fredrik Lindblad (2007): *Property directed generation of first-order test data*. In Marco Morazán, editor: *TFP 2007*, Intellect, pp. 105–123.

[28] Andreas Lochbihler & Lukas Bulwahn (2011): *Animating the formalised semantics of a Java-like language*. In Marko van Eekelen, Herman Geuvers, Julien Schmalz & Freek Wiedijk, editors: *ITP 2011*, *LNCS* 6898, Springer, pp. 216–232, doi:10.1007/978-3-642-22863-6_17.

[29] Panagiotis Manolios (2013): *Counterexample generation meets interactive theorem proving: Current results and future opportunities*. In Sandrine Blazy, Christine Paulin-Mohring & David Pichardie, editors: *ITP 2013*, *LNCS* 7998, Springer, p. 18, doi:10.1007/978-3-642-39634-2_4.

[30] William McCune: *Prover9 and Mace4*. http://www.cs.unm.edu/~mccune/prover9/.

[31] William McCune (1994): *A Davis–Putnam program and its application to finite first-order model search: quasigroup existence problems*. Technical Report, Argonne National Laboratory.

[32] Tobias Nipkow, Lawrence C. Paulson & Markus Wenzel (2002): *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. *LNCS* 2283, Springer.

[33] Sam Owre (2006): *Random testing in PVS*. In: *AFM '06*.

[34] Zoe Paraskevopoulou, Catalin Hritcu, Maxime Dénès, Leonidas Lampropoulos & Benjamin C. Pierce (2015): *Foundational property-based testing*. In Christian Urban & Xingyuan Zhang, editors: *ITP 2015*, *LNCS* 9236, Springer, pp. 325–343, doi:10.1007/978-3-319-22102-1_22.

[35] Wolfgang Reif, Gerhard Schellhorn & Andreas Thums (2001): *Flaw detection in formal specifications*. In Rajeev Goré, Alexander Leitsch & Tobias Nipkow, editors: *IJCAR 2001*, *LNCS* 2083, Springer, pp. 642–657, doi:10.1007/3-540-45744-5_52.

[36] Andrew Reynolds & Jasmin Christian Blanchette (2015): *A decision procedure for (co)datatypes in SMT solvers*. In Amy Felty & Aart Middeldorp, editors: *CADE-25*, *LNCS* 9195, Springer, pp. 197–213, doi:10.1007/978-3-319-21401-6_13.

[37] Andrew Reynolds, Jasmin Christian Blanchette, Simon Cruanes & Cesare Tinelli (2016): *Model finding for recursive functions in SMT*. In N. Olivetti & A. Tiwari, editors: *IJCAR 2016*, *LNCS* 9706, Springer.

[38] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstić, Morgan Deters & Clark Barrett (2013): *Quantifier instantiation techniques for finite model finding in SMT*. In Maria Paola Bonacina, editor: *CADE-24*, *LNCS* 7898, Springer, pp. 377–391, doi:10.1007/978-3-642-38574-2_26.

[39] Colin Runciman, Matthew Naylor & Fredrik Lindblad (2008): *SmallCheck and Lazy SmallCheck: Automatic exhaustive testing for small values*. In Andy Gill, editor: *Haskell 2008*, ACM, pp. 37–48, doi:10.1145/1411286.1411292.

[40] John K. Slaney (1994): *FINDER: Finite domain enumerator—System description*. In Alan Bundy, editor: *CADE-12*, *LNCS* 814, Springer, pp. 798–801, doi:10.1007/3-540-58156-1_63.

[41] Matthieu Sozeau & Nicolas Oury (2008): *First-class type classes*. In Otmane Ait Mohamed, César Muñoz & Sofiène Tahar, editors: *TPHOLs 2008*, *LNCS* 5170, Springer, pp. 278–293, doi:10.1007/978-3-540-71067-7_23.

[42] Philippe Suter, Ali Sinan Köksal & Viktor Kuncak (2011): *Satisfiability modulo recursive programs*. In Eran Yahav, editor: *SAS 2011*, *LNCS* 6887, Springer, pp. 298–315, doi:10.1007/978-3-642-23702-7_23.

[43] Emina Torlak & Daniel Jackson (2007): *Kodkod: A relational model finder*. In Orna Grumberg & Michael Huth, editors: *TACAS 2007*, *LNCS* 4424, Springer, pp. 632–647, doi:10.1007/978-3-540-71209-1_49.

[44] Tjark Weber (2008): *SAT-Based Finite Model Generation for Higher-Order Logic*. Ph.D. thesis, Technische Universität München.

[45] Markus Wenzel (1997): *Type classes and overloading in higher-order logic*. In Elsa L. Gunter & Amy Felty, editors: *TPHOLs 1997*, *LNCS* 1275, Springer, pp. 307–322, doi:10.1007/BFb0028402.

[46] Jian Zhang & Hantao Zhang (1995): *SEM: A system for enumerating models*. In Chris S. Mellish, editor: *IJCAI-95*, 1, Morgan Kaufmann, pp. 298–303.

Goal Translation for a Hammer for Coq (Extended Abstract)

Łukasz Czajka
lukasz.czajka@uibk.ac.at
University of Innsbruck, Austria

Cezary Kaliszyk
cezary.kaliszyk@uibk.ac.at
University of Innsbruck, Austria

Hammers are tools that provide general purpose automation for formal proof assistants. Despite the gaining popularity of the more advanced versions of type theory, there are no hammers for such systems. We present an extension of the various hammer components to type theory: (i) a translation of a significant part of the Coq logic into the format of automated proof systems; (ii) a proof reconstruction mechanism based on a Ben-Yelles-type algorithm combined with limited rewriting, congruence closure and a first-order generalization of the left rules of Dyckhoff's system LJT.

1 Introduction

Justifying small proof steps is usually a significant part of the process of formalizing proofs in an *interactive theorem proving* (ITP), or *proof assistant*, system. Many of such goals would be considered trivial by mathematicians. Still, state-of-the-art ITPs require the user to spend an important part of the formalization effort on them. The main points that constitute this effort are usually library search, minor transformations on the already proved theorems (such as reordering assumptions or reasoning modulo associativity-commutativity), as well as combining a small number of simple known lemmas. To reduce this effort various automation techniques have been conceived, including techniques from automated reasoning and domain specific decision procedures. The strongest general propose automation technique, available for various interactive theorem provers today is provided by "hammers" [10].

Hammers are proof assistant tools that employ external automated theorem provers (ATPs) in order to automatically find proofs of user given conjectures. There are three main components of a hammer:

- Lemma selection (also called relevance filtering or premise selection) that heuristically chooses a subset of the accessible lemmas that are likely useful for the given conjecture.
- Translation (encoding) of the user given conjecture together with the selected lemmas to the logics and input formats of automated theorem provers (ATPs). The focus is usually on first-order logic as the majority of the most efficient ATPs today support this foundation. The automated systems are in turn used to either find an ATP proof or just further narrow down the subset of lemmas to precisely those that are necessary in the proof.
- Proof reconstruction, which uses the obtained information from the successful ATP run, to reprove the lemma in the logic of the proof assistant.

Robust hammers exist for proof assistants based on higher-order logic (Sledgehammer [27] for Isabelle/HOL [33], HOLyHammer [20] for HOL Light [18] and HOL4 [31]) or dependently typed set theory (MizAR [21] for Mizar [7, 34, 6]). The general-purpose automation provided by the most advanced hammers is able to solve 40–50% of the top-level goals in various developments [10], as well as more than 70% of the user-visible subgoals [11], and as such has been found very useful in various proof developments [17].

Despite the gaining popularity of the more advanced versions of type theory, implemented by systems such as Agda [12], Coq [8], Lean [25], and Matita [4], there are no hammers for such systems. The

J.C. Blanchette and C. Kaliszyk:
Hammers for Type Theories (HaTT'16)
EPTCS 210, 2016, pp. 13–20, doi:10.4204/EPTCS.210.4

construction of such a tool has so far been hindered by the lack of a usable encoding component, as well as by comparatively weak proof reconstuction.

For the proof assistants whose logics are based on the Calculus of Constructions and its extensions, the existing encodings in first-order logic so far cover only limited fragments of the source logic [1, 32, 9]. Why3 [16] provides a translation from its own logic [15] (which is a subset of the Coq logic, including features like rank-1 polymorphism, algebraic data types, recursive functions and inductive predicates) to the format of various first-order provers (in fact Why3 has been initially used as a translation backend for HOLyHammer). Recently, an encoding of the dependently typed higher-order logic of F* into first-order logic has also been developed [2].

The built-in HOL automation is able to reconstruct the majority of the automatically found proofs using either internal proof search [19] or source-level reconstruction. The internal proof search mechanisms provided in Coq, such as the `firstorder` tactic [13], have been insufficient for this purpose so far. Matita's ordered paramodulation [5] is able to reconstruct many goals with up to two or three premises, and the congruence-closure based internal automation techniques in Lean [24] are also promising.

The SMTCoq [3] project has developed an approach to use external SAT and SMT solvers and verify their proof witnesses. Small checkers are implemented using reflection for parts of the SAT and SMT proof reconstruction, such as one for CNF computation and one for congruence closure. The procedure is able to handle Coq goals in the subset of the logic that corresponds to the logics of the input systems.

Contributions. We present our recently developed proof advice components for type theory and systems based on it. We first introduce an encoding of the Calculus of Inductive Constructions, including the additional logical constructions introduced by the Coq system, in untyped first-order logic with equality. We implement the translation and evaluate it experimentally on the standard library of the Coq proof assistant. We advocate that the encoding is sufficient for a hammer system for Coq: the success rates are comparable to those demonstrated by early hammer systems for Isabelle/HOL and Mizar, while the dependencies used in the ATP proofs are most often sufficient to prove the original theorems. Strictly speaking, our translation is neither sound nor complete. However, our experiments suggest that the encoding is "sound enough" to be usable. Moreover, we believe that a "core" version of the translation is sound and we are currently working on a proof of this fact.

Secondly, we present a proof reconstruction mechanism based on a Ben-Yelles-type procedure combined with a first-order generalization of the left rules of Dyckhoff's LJT, congruence closure and heuristic rewriting. With this still preliminary proof search procedure we are able to reprove almost 90% of the problems solved by the ATPs, using the dependencies extracted from the ATP output.

2 Translation

In this section we introduce an encoding of (a close approximation of) the Calculus of Inductive Constructions into untyped first-order logic with equality. The encoding should be a practical one, which implies that its general theoretical soundness is not the main focus, i.e., of course the translation needs to be "sound enough" to be usable, but it is more important that the encoding is efficient enough to provide practically useful information about the necessary proof dependencies. In particular, the encoding needs to be shallow, meaning that Coq terms of type Prop are translated directly to corresponding first-order formulas. Our translation is in fact unsound, e.g., it assumes proof irrelevance and ignores certain universe constraints. However, we believe that under the assumption of proof irrelevance a "core" version of the translation is sound, and we are currently working on a proof.

Below we present a variant of the translation for a fragment of the logic of Coq. The intention here is to provide a general idea, but not to describe the encoding in detail. In the first-order language we assume a unary predicate P, a binary predicate T and a binary function symbol @. Usually, we write ts instead of $@(t,s)$.

For the sake of efficiency, terms of type Prop are encoded directly as FOL formulas using a function $\mathscr{F}$. Terms that have type Type but not Prop are encoded using a function $\mathscr{G}$ as guards which essentially specify what it means for an object to have the given type. For instance, $\forall f : \tau.\varphi$ where $\tau = \Pi x : \alpha.\beta$ is translated to $\forall f.\mathscr{G}(\tau, f) \to \mathscr{F}(\varphi)$ where $\mathscr{G}(\tau, f) = \forall x.\mathscr{G}(\alpha, x) \to \mathscr{G}(\beta, fx)$. So $\mathscr{G}(\tau, f)$ says that an object f has type $\tau = \Pi x : \alpha.\beta$ if for any object x of type α, the application fx has type β. Function $\mathscr{F}$ encoding propositions as FOL formulas is defined by:

- If $\Gamma \vdash t : \mathrm{Prop}$ then $\mathscr{F}_\Gamma(\Pi x : t.s) = \mathscr{F}_\Gamma(t) \to \mathscr{F}_{\Gamma,x:t}(s)$.
- If $\Gamma \nvdash t : \mathrm{Prop}$ then $\mathscr{F}_\Gamma(\Pi x : t.s) = \forall x.\mathscr{G}_\Gamma(t,x) \to \mathscr{F}_{\Gamma,x:t}(s)$.
- Otherwise, if none of the above apply, $\mathscr{F}_\Gamma(t) = P(\mathscr{C}_\Gamma(t))$.

Function $\mathscr{G}$ encoding types as guards is defined by:

- If $t = \Pi x : t_1.t_2$ and $\Gamma \vdash t_1 : \mathrm{Prop}$ then $\mathscr{G}_\Gamma(\Pi x : t_1.t_2, s) = \mathscr{F}_\Gamma(t_1) \to \mathscr{G}_{\Gamma,x:t_1}(t_2, s)$.
- If $t = \Pi x : t_1.t_2$ and $\Gamma \nvdash t_1 : \mathrm{Prop}$ then $\mathscr{G}_\Gamma(\Pi x : t_1.t_2, s) = \forall x.\mathscr{G}_\Gamma(t_1, x) \to \mathscr{G}_{\Gamma,x:t_1}(t_2, sx)$.
- Otherwise, when t is not a product $\mathscr{G}_\Gamma(t,s) = T(u, \mathscr{C}_\Gamma(t))$.

Function $\mathscr{C}$ encoding terms as FOL terms is defined by:

- $\mathscr{C}_\Gamma(b) = b$ for b being a variable or a constant,
- $\mathscr{C}_\Gamma(ts)$ is equal to:
  - $\mathscr{C}_\Gamma(t)$ if $\Gamma \vdash s : A : \mathrm{Prop}$ for some A,
  - $\mathscr{C}_\Gamma(t)\mathscr{C}_\Gamma(s)$ otherwise.
- $\mathscr{C}_\Gamma(\Pi x : t.s) = P\vec{y}$ for a fresh constant P where $\vec{y} = \mathrm{FV}(\Pi x : t.s)$ and
  - if $\Gamma \vdash (\Pi x : t.s) : \mathrm{Prop}$ then $\forall \vec{y}.P\vec{y} \leftrightarrow \mathscr{F}_\Gamma(\Pi x : t.s)$ is a new axiom,
  - if $\Gamma \nvdash (\Pi x : t.s) : \mathrm{Prop}$ then $\forall \vec{y}z.P\vec{y}z \leftrightarrow \mathscr{G}_\Gamma(\Pi x : t.s, z)$ is a new axiom.
- $\mathscr{C}_\Gamma(\lambda \vec{x} : \vec{t}.s) = F\vec{y}$ where s does not start with a lambda-abstraction any more, F is a fresh constant, $\vec{y} = \mathrm{FV}(\lambda \vec{x} : \vec{t}.s)$ and $\forall \vec{y}.\mathscr{F}_\Gamma(\forall \vec{x} : \vec{t}.F\vec{y}\vec{x} = s)$ is a new axiom.
- $\mathscr{C}_\Gamma(\texttt{case}(t, c, n, \lambda \vec{a} : \vec{\alpha}.\lambda x : c\vec{p}\vec{a}.\tau, \lambda \vec{x_1} : \vec{\tau_1}.s_1, \ldots, \lambda \vec{x_k} : \vec{\tau_k}.s_k)) = F\vec{y_1}\vec{y_2}$ for a fresh constant F where
  - $I(c : \gamma : \kappa := c_1 : \gamma_1 : \kappa_1, \ldots, c_k : \gamma_k : \kappa_k) \in E$,
  - $\Gamma_2 = \vec{y_2} : \vec{\rho_2} = \mathrm{FC}(\Gamma; t)$,
  - $\Gamma_1 = \vec{y_1} : \vec{\rho_1} = \mathrm{FC}(\Gamma; \lambda \vec{y_2} : \vec{\rho_2}.t(\lambda \vec{x_1} : \vec{\tau_1}.s_1) \ldots (\lambda \vec{x_k} : \vec{\tau_k}.s_k))$,
  - $\gamma_i = \Pi \vec{z_i} : \vec{\beta_i}.\Pi \vec{x_i} : \vec{\tau_i}.\sigma_i$ for $i = 1, \ldots, k$,
  - the following is a new axiom:

$$\begin{array}{rcl}\forall \vec{y_1}.\mathscr{F}_{\Gamma_1}(\forall \vec{y_2} : \vec{\rho_2} & . & (\exists \vec{z_1} : \vec{\beta_1}.\exists \vec{x_1} : \vec{\tau_1}.t = c_1\vec{z_1}\vec{x_1} \wedge F\vec{y_1}\vec{y_2} = s_1) \\ & \vee & \ldots \\ & \vee & (\exists \vec{z_k} : \vec{\beta_k}.\exists \vec{x_k} : \vec{\tau_k}.t = c_k\vec{z_k}\vec{x_k} \wedge F\vec{y_1}\vec{y_2} = s_k))\end{array}$$

Here t is the term matched on, the type of t has the form $c\vec{p}\vec{u}$, the integer n denotes the number of parameters (which is the length of $\vec{p}$), the type $\tau[\vec{u}/\vec{a}, t/x]$ is the return type, i.e., the type of the whole case expression, $\vec{a} \cap \mathrm{FV}(\vec{p}) = \emptyset$, and $s_i[\vec{v}/\vec{x_i}]$ is the value of the case expression if the value of t is $c_i\vec{p}\vec{v}$. The free variable context $\mathrm{FC}(\Gamma;t)$ of t in Γ is defined inductively: $\mathrm{FC}(\emptyset;t) = \emptyset$; $\mathrm{FC}(\Gamma, x : \tau; t) = \mathrm{FC}(\Gamma; \lambda x : \tau.t), x : \tau$ if $x \in \mathrm{FV}(t)$; and $\mathrm{FC}(\Gamma, x : \tau; t) = \mathrm{FC}(\Gamma; t)$ if $x \notin \mathrm{FV}(t)$.

In the data exported from Coq there are three types of declarations: definitions, typing declarations and inductive declarations. We briefly describe how all of them are translated.

A definition $c = t : \tau : \kappa$ is translated as follows.

- If $\kappa = \mathrm{Prop}$ then add $\mathscr{F}(\tau)$ as a new axiom with label c.
- If $\kappa \neq \mathrm{Prop}$ then
  - add $\mathscr{G}(\tau, c)$ as a new axiom,
  - if $\tau = \mathrm{Prop}$ then add $c \leftrightarrow \mathscr{F}(t)$ as a new axiom with label c,
  - if $\tau = \mathrm{Set}$ or $\tau = \mathrm{Type}$ then add $\forall f.cf \leftrightarrow \mathscr{G}(t, f)$ as a new axiom with label c,
  - if $\tau \notin \{\mathrm{Prop}, \mathrm{Set}, \mathrm{Type}\}$ then add the equation $c = \mathscr{C}(t)$ as a new axiom with label c.

A typing declaration $c : \tau : \kappa$ is translated as follows.

- If $\kappa = \mathrm{Prop}$ then add $\mathscr{F}(\tau)$ as a new axiom with label c.
- If $\kappa \neq \mathrm{Prop}$ then add $\mathscr{G}(\tau, c)$ as a new axiom with label c.

An inductive declaration $I(c : \tau : \kappa := c_1 : \tau_1 : \kappa_1, \ldots, c_n : \tau_n : \kappa_n)$ is translated as follows.

- Translate the typing declaration $c : \tau : \kappa$.
- Translate each typing declaration $c_i : \tau_i : \kappa$ for $i = 1, \ldots, n$.
- Add axioms stating injectivity of constructors, axioms stating non-equality of different constructors, and the "inversion" axioms for elements of the inductive type.

For inductive types also induction principles and recursor definitions are translated.

The above only gives a general outline of the translation. In practice, we make a number of optimisations, e.g., the arity optimisation by Meng and Paulson [23], or translating fully applied functions with target type Prop directly to first-order predicates.

3 Reconstruction

We report on our work on proof reconstruction. We evaluate the Coq internal reconstruction mechanisms including `tauto` and `firstorder` [13] on the original proof dependencies and on the ATP found proofs, which are in certain cases more precise. In particular `firstorder` seems insufficient for finding proofs for problems created using the advice obtained from the ATP runs. This is partly caused by the fact that it does not fully axiomatize equality, but even on problems which require only purely logical first-order reasoning its running time is sometimes unacceptable.

The formulas that we attempt to reprove usually belong to fragments of intuitionistic logic low in the Mints hierarchy [29]. Most of proved theorems follow by combining a few known lemmas. This raises a possibility of devising an automated proof procedure optimized for these fragments of intuitionistic logic, and for the usage of the advice obtained from the ATP runs. We implemented a preliminary version of a Ben-Yelles-type procedure (essentially `eauto`-type proof search with a looping check) augmented with

Prover	Solved%	Solved	Sum%	Sum	Unique
Vampire	32.9	6839	32.9	6839	855
Z3	27.6	5734	34.9	7265	390
E Prover	25.8	5376	35.3	7337	72
any	35.3	7337	35.3	7337	

Table 1: Results of the experimental evaluation on the 20803 FOL problems generated from the propositions in the Coq standard library.

a first-order generalization of the left rules of Dyckhoff's system LJT [14], the use of the `congruence` tactic, and heuristic rewriting using equational hypotheses.

It is important to note that while the external ATPs we employ are classical and the translation assumes proof irrelevance, the proof reconstruction phase does not assume any additional axioms. We reprove the theorems in the intuitionistic logic of Coq, effectively using the output of the ATPs merely as hints for our hand-crafted proof search procedure. Therefore, if the ATP proof is inherently classical then proof reconstruction will fail. Currently, the only information from ATP runs we use is a list of lemmas needed by the ATP to prove the theorem (these are added to the context) and a list of constant definitions used in the ATP proof (we try unfolding these constants and no others).

Another thing to note is that we do not use the information contained in the Coq standard library during reconstruction. This would not make sense for our evaluation of the reconstruction mechanism, since we try to reprove the theorems from the Coq standard library. In particular, we do not use any preexisting hint databases available in Coq, not even the core database (we use the `auto` and `eauto` tactics with the `nocore` option). Also, we do not use any domain-specific decision procedures available as Coq tactics, e.g., `field`, `ring` or `omega`.

4 Evaluation

We evaluated our translation on the problems generated from all declarations of terms of type Prop in the Coq standard library of Coq version 8.5. We used the following classical ATPs: E Prover version 1.9 [30], Vampire version 4.0 [22] and Z3 version 4.0 [26]. The methodology was to measure the number of theorems that the ATP could reprove from their extended dependencies within a time limit of 30 s for each problem. The extended dependencies of a theorem are obtained by taking all constants occuring in the proof term of the theorem in Coq standard library, and recursively taking all constants occuring in the types and non-proof definitions of any dependencies extracted so far. Because of the use of extended dependencies, the average number of generated FOL axioms for a problem is 193. We limited the recursive extraction of extended dependencies to depth 2.

The evaluation was performed on a 48-core server with 2.2 GHz AMD Opteron CPUs and 320 GB RAM. Each problem was always assigned one CPU core. Table 1 shows the results of our evaluation. The column “Solved%” denotes the percentage (rounded to the first decimal place) of the problems solved by a given prover, and “Solved” the number of problems solved out of the total number of 20803 problems. The column “Sum%” denotes the percentage, and “Sum” the total number, of problems solved by the prover or any of the provers listed above it. The column “Unique” denotes the number of problems the given prover solved but no other prover could solve.

We also evaluated various proof reconstruction mechanisms on the problems originating from ATP

Tactic	Time	Solved%	Solved
`yreconstr0`	10s	26.8	1965
`yreconstr`	1s	83.1	6097
`yreconstr`	2s	85.8	6296
`yreconstr`	5s	87.5	6421
`yreconstr`	10s	88.1	6466
`yreconstr`	15s	88.2	6473
`simple`	1s	50.1	3674
`firstorder'`	10s	69.6	5103
`jprover`	10s	56.1	4114
any		90.1	6609

Table 2: Results of the evaluation of proof reconstruction on the 7337 problems solved by the ATPs.

proofs of lemmas in the Coq standard library. In our setting, the Ben-Yelles-type algorithm mentioned in the previous section tends to perform significantly better than the available Coq's tactics. The results of the evaluation are presented in Table 2. Our tactic (`yreconstr`) manages to reconstruct about 88% of the reproved theorems. However, it needs to be remarked that if we use the advice obtained from ATP runs then about 50% of the the reproved theorems follow by a combination of hypothesis simplification, the tactics `intuition`, `auto`, `easy`, `congruence` and a few heuristics (tactic `simple`). Moreover, the `yreconstr` tactic without any hints (`yreconstr0`), i.e., without using any of the information obtained from ATP runs, achieves a success rate of about 26%. The reconstruction success rate of the `firstorder` tactic combined with various heuristics is about 70% if generic axioms for equality are added to the context (tactic `firstorder'`). The `jp` tactic (which integrates the intuitionistic first-order automated theorem prover JProver [28] into Coq) combined with various heuristics and equality axioms (tactic `jprover`) achieves a reconstruction success rate of about 56%. This low success rate is explained by the fact that in contrast to the `firstorder` tactic the `jp` tactic cannot be parameterised by a tactic used at the leaves of the search tree when no logical rule applies.

Acknowledgments. We thank the organizers of the First Coq Coding Sprint, especially Yves Bertot, for the help with implementing Coq export plugins. We wish to thank Thibault Gauthier for the first version of the Coq exported data, as as well as Claudio Sacerdoti-Coen for improvements to the exported data and fruitful discussions on Coq proof reconstruction. This work has been supported by the Austrian Science Fund (FWF) grant P26201.

References

[1] Andreas Abel, Thierry Coquand & Ulf Norell (2005): *Connecting a Logical Framework to a First-Order Logic Prover*. In Bernhard Gramlich, editor: *Frontiers of Combining Systems (FroCoS 2005), LNCS* 3717, Springer, pp. 285–301, doi:10.1007/11559306_17.

[2] Alejandro Aguirre, Cătălin Hrițcu, Chantal Keller & Nikhil Swamy (2016): *From* F* *to SMT (Extended Abstract)*. In: *Hammers for Type Theories, HaTT 2016.* To appear.

[3] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry & Benjamin Werner (2011): *A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses*. In Jean-Pierre Jouan-

naud & Zhong Shao, editors: *Certified Programs and Proofs (CPP 2011)*, *LNCS* 7086, Springer, pp. 135–150, doi:10.1007/978-3-642-25379-9_12.

[4] Andrea Asperti, Wilmer Ricciotti & Claudio Sacerdoti Coen (2014): *Matita Tutorial*. *J. Formalized Reasoning* 7(2), pp. 91–199, doi:10.6092/issn.1972-5787/4651.

[5] Andrea Asperti & Enrico Tassi (2007): *Higher order Proof Reconstruction from Paramodulation-Based Refutations: The Unit Equality Case*. In Manuel Kauers, Manfred Kerber, Robert Miner & Wolfgang Windsteiger, editors: *Mathematical Knowledge Management (MKM 2007)*, *LNCS* 4573, Springer, pp. 146–160, doi:10.1007/978-3-540-73086-6_14.

[6] Grzegorz Bancerek (2003): *On the structure of Mizar types*. *Electr. Notes Theor. Comput. Sci.* 85(7), pp. 69–85, doi:10.1016/S1571-0661(04)80758-8.

[7] Grzegorz Bancerek, Czeslaw Bylinski, Adam Grabowski, Artur Kornilowicz, Roman Matuszewski, Adam Naumowicz, Karol Pak & Josef Urban (2015): *Mizar: State-of-the-art and Beyond*. In: *Intelligent Computer Mathematics - International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings*, pp. 261–279, doi:10.1007/978-3-319-20615-8_17.

[8] Yves Bertot (2008): *A Short Presentation of Coq*. In Otmane Aït Mohamed, César A. Muñoz & Sofiène Tahar, editors: *Theorem Proving in Higher Order Logics (TPHOLs 2008)*, *LNCS* 5170, Springer, pp. 12–16, doi:10.1007/978-3-540-71067-7_3.

[9] Marc Bezem, Dimitri Hendriks & Hans de Nivelle (2002): *Automated Proof Construction in Type Theory Using Resolution*. *J. Autom. Reasoning* 29(3-4), pp. 253–275, doi:10.1023/A:1021939521172.

[10] Jasmin C. Blanchette, Cezary Kaliszyk, Lawrence C. Paulson & Josef Urban (2016): *Hammering towards QED*. *J. Formalized Reasoning* 9(1), pp. 101–148, doi:10.6092/issn.1972-5787/4593. Available at http://jfr.unibo.it/article/view/4593.

[11] Jasmin Christian Blanchette, David Greenaway, Cezary Kaliszyk, Daniel Kühlwein & Josef Urban (2016): *A Learning-Based Relevance Filter for Isabelle/HOL*. J. Autom. Reasoning, to appear. http://cl-informatik.uibk.ac.at/cek/mash2.pdf.

[12] Ana Bove, Peter Dybjer & Ulf Norell (2009): *A Brief Overview of Agda - A Functional Language with Dependent Types*. In Stefan Berghofer, Tobias Nipkow, Christian Urban & Makarius Wenzel, editors: *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, *LNCS* 5674, Springer, pp. 73–78, doi:10.1007/978-3-642-03359-9_6.

[13] Pierre Corbineau (2003): *First-Order Reasoning in the Calculus of Inductive Constructions*. In Stefano Berardi, Mario Coppo & Ferruccio Damiani, editors: *Types for Proofs and Programs (TYPES 2003)*, *LNCS* 3085, Springer, pp. 162–177, doi:10.1007/978-3-540-24849-1_11.

[14] Roy Dyckhoff (1992): *Contraction-Free Sequent Calculi for Intuitionistic Logic*. *J. Symb. Log.* 57(3), pp. 795–807, doi:10.2307/2275431.

[15] Jean-Christophe Filliâtre (2013): *One Logic to Use Them All*. In Maria Paola Bonacina, editor: *International Conference on Automated Deduction (CADE 2013)*, *LNCS* 7898, Springer, pp. 1–20, doi:10.1007/978-3-642-38574-2_1.

[16] Jean-Christophe Filliâtre & Andrei Paskevich (2013): *Why3 - Where Programs Meet Provers*. In: *European Symposium on Programming (ESOP 2013)*, pp. 125–128, doi:10.1007/978-3-642-37036-6_8.

[17] Thomas Hales (2013–2014): *Developments in Formal Proofs*. *Séminaire Bourbaki* 1086. abs/1408.6474.

[18] John Harrison (2009): *HOL Light: An Overview*. In Stefan Berghofer, Tobias Nipkow, Christian Urban & Makarius Wenzel, editors: *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, *LNCS* 5674, Springer, pp. 60–66, doi:10.1007/978-3-642-03359-9_4.

[19] Joe Hurd (2003): *First-Order Proof Tactics in Higher-Order Logic Theorem Provers*. In Myla Archer, Ben Di Vito & César Muñoz, editors: *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)*, *NASA Technical Reports* NASA/CP-2003-212448, pp. 56–68. Available at http://techreports.larc.nasa.gov/ltrs/PDF/2003/cp/NASA-2003-cp212448.pdf.

[20] Cezary Kaliszyk & Josef Urban (2014): *Learning-Assisted Automated Reasoning with Flyspeck*. *J. Autom. Reasoning* 53(2), pp. 173–213, doi:`10.1007/s10817-014-9303-3`.

[21] Cezary Kaliszyk & Josef Urban (2015): *MizAR 40 for Mizar 40*. *J. Autom. Reasoning* 55(3), pp. 245–256, doi:`10.1007/s10817-015-9330-8`.

[22] Laura Kovács & Andrei Voronkov (2013): *First-Order Theorem Proving and Vampire*. In: *Computer-Aided Verification (CAV 2013)*, pp. 1–35, doi:`10.1007/978-3-642-39799-8_1`.

[23] Jia Meng & Lawrence C. Paulson (2008): *Translating Higher-Order Clauses to First-Order Clauses*. *Journal of Automated Reasoning* 40(1), pp. 35–60, doi:`10.1007/s10817-007-9085-y`.

[24] Leonardo de Moura & Daniel Selsam (2016): *Congruence Closure in Intensional Type Theory*. In: *International Joint Conference on Automated Reasoning, IJCAR 2016*. To appear.

[25] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn & Jakob von Raumer (2005): *The Lean Theorem Prover*. In Amy P. Felty & Aart Middeldorp, editors: *International Conference on Automated Deduction (CADE 2005)*, *LNCS* 9195, Springer, pp. 378–388, doi:`10.1007/978-3-319-21401-6_26`.

[26] Leonardo Mendonça de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In C. R. Ramakrishnan & Jakob Rehof, editors: *TACAS 2008*, *LNCS* 4963, Springer, pp. 337–340, doi:`10.1007/978-3-540-78800-3_24`.

[27] Lawrence C. Paulson & Jasmin Blanchette (2010): *Three Years of Experience with Sledgehammer, a Practical Link between Automated and Interactive Theorem Provers*. In: *8th IWIL*. Available at `http://www4.in.tum.de/~schulz/PAPERS/STS-IWIL-2010.pdf`.

[28] Stephan Schmitt, Lori Lorigo, Christoph Kreitz & Aleksey Nogin (2001): *JProver : Integrating Connection-Based Theorem Proving into Interactive Proof Assistants*. In Rajeev Goré, Alexander Leitsch & Tobias Nipkow, editors: *Automated Reasoning, First International Joint Conference, IJCAR 2001, Siena, Italy, June 18-23, 2001, Proceedings*, *Lecture Notes in Computer Science* 2083, Springer, pp. 421–426, doi:`10.1007/3-540-45744-5_34`.

[29] Aleksy Schubert, Paweł Urzyczyn & Konrad Zdanowski (2015): *On the Mints Hierarchy in First-Order Intuitionistic Logic*. In Andrew M. Pitts, editor: *Foundations of Software Science and Computation Structures (FoSSaCS 2015)*, *Lecture Notes in Computer Science* 9034, Springer, pp. 451–465, doi:`10.1007/978-3-662-46678-0_29`.

[30] Stephan Schulz (2013): *System Description: E 1.8*. In: *Logic for Programming, Artificial Intelligence (LPAR 2013)*, pp. 735–743, doi:`10.1007/978-3-642-45221-5_49`.

[31] Konrad Slind & Michael Norrish (2008): *A Brief Overview of HOL4*. In Otmane Ait Mohamed, César Muñoz & Sofiène Tahar, editors: *TPHOLs 2008*, *LNCS* 5170, Springer, pp. 28–32, doi:`10.1007/978-3-540-71067-7_6`.

[32] Tanel Tammet & Jan M. Smith (1998): *Optimized Encodings of Fragments of Type Theory in First-Order Logic*. *J. Log. Comput.* 8(6), pp. 713–744, doi:`10.1093/logcom/8.6.713`.

[33] Makarius Wenzel, Lawrence C. Paulson & Tobias Nipkow (2008): *The Isabelle Framework*. In Otmane Aït Mohamed, César A. Muñoz & Sofiène Tahar, editors: *Theorem Proving in Higher Order Logics (TPHOLs 2008)*, *LNCS* 5170, Springer, pp. 33–38, doi:`10.1007/978-3-540-71067-7_7`.

[34] Freek Wiedijk (2007): *Mizar's Soft Type System*. In: *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, pp. 383–399, doi:`10.1007/978-3-540-74591-4_28`.

Extending SMTCoq, a Certified Checker for SMT (Extended Abstract)

Burak Ekici
The University of Iowa
Iowa City, USA
burak-ekici@uiowa.edu

Guy Katz
New York University
New York, USA
guy.katz@nyu.edu

Chantal Keller
LRI, Université Paris-Sud
Orsay, France
Chantal.Keller@lri.fr

Alain Mebsout
The University of Iowa
Iowa City, USA
alain-mebsout@uiowa.edu

Andrew J. Reynolds
The University of Iowa
Iowa City, USA
andrew-reynolds@uiowa.edu

Cesare Tinelli
The University of Iowa
Iowa City, USA
cesare-tinelli@uiowa.edu

This extended abstract reports on current progress of SMTCoq, a communication tool between the Coq proof assistant and external SAT and SMT solvers. Based on a checker for generic first-order certificates implemented and proved correct in Coq, SMTCoq offers facilities both to check external SAT and SMT answers and to improve Coq's automation using such solvers, in a safe way. Currently supporting the SAT solver ZChaff, and the SMT solver veriT for the combination of the theories of congruence closure and linear integer arithmetic, SMTCoq is meant to be extendable with a reasonable amount of effort: we present work in progress to support the SMT solver CVC4 and the theory of bit vectors.

1 Introduction

SMTCoq[1] [1] is a tool that allows the Coq [2] proof assistant to communicate with external automatic solvers for Boolean satisfiability (SAT) and Satisfiability Modulo Theories (SMT). Its twofold goal is to:

- increase the confidence in SAT and SMT solvers: SMTCoq provides an independent and certified checker for SAT and SMT proof witnesses;
- safely increase the level of automation of Coq: SMTCoq provides starting safe tactics to solve a class of Coq goals automatically by calling external solvers and checking their answers (following a *skeptical* approach).

SMTCoq currently supports the SAT solver ZChaff [19] and the SMT solver veriT [10] for the quantifier-free fragment of the combined theory of linear integer arithmetic and equality with uninterpreted functions. For this combined theory, SMTCoq's certificate checker has proved to be as efficient as state-of-the-art certified checkers [1, 9].

There is a large variety of SAT and SMT solvers, with each solver typically excelling at solving problems in some specific class of propositional or first-order problems. While the SAT and SMT communities have adopted standard languages for expressing *input* problems (namely the DIMACS standard for SAT and the SMT-LIB [4] standard for SMT), agreeing on a common *output* language for proof witnesses has proven to be more challenging. Several formats [11, 21, 6] have been proposed but none has emerged as a standard yet. Each proof-producing solver currently implements its own variant of these formats.

[1] SMTCoq is distributed as free software at https://github.com/smtcoq/smtcoq.

J.C. Blanchette and C. Kaliszyk:
Hammers for Type Theories (HaTT'16)
EPTCS 210, 2016, pp. 21–29, doi:10.4204/EPTCS.210.5

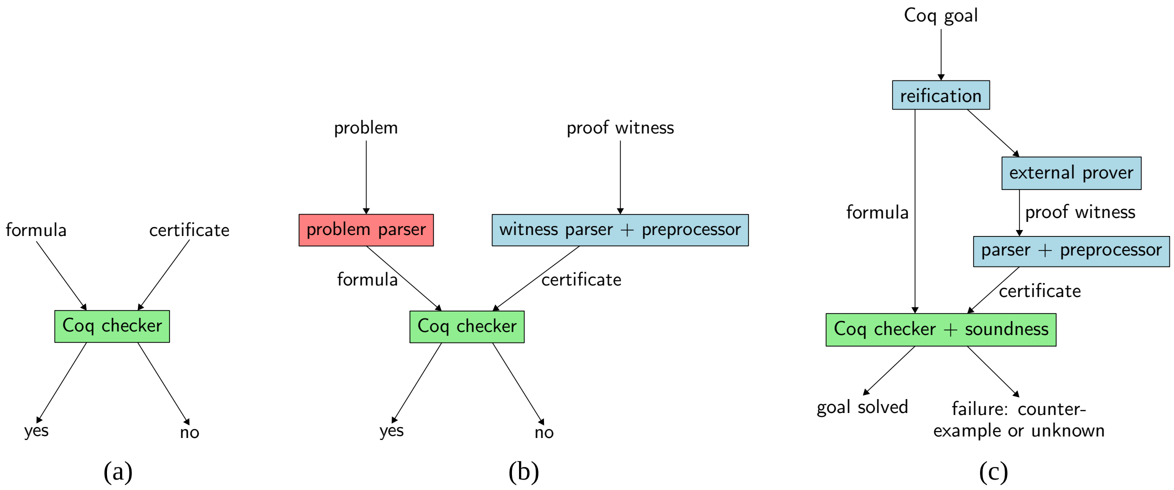

Figure 1: SMTCoq's main checker and its uses

To be able to combine the advantages of multiple SAT and SMT solvers despite the lack of common standards for representing proof certicates, SMTCoq has been designed to be modular along two dimensions:

- supporting new theories: SMTCoq's main checker is an extendable combination of independent *small checkers*;
- supporting new solvers: SMTCoq's kernel relies on a generic certificate format that can encode most SAT and SMT reasonings for supported theories; the encoding can be done during a *preprocessing* phase, which does not need to be certified.

In this abstract, we emphasize the key ideas behind the modularity of SMTCoq, and validate this by reporting on work in progress on the integration of the SMT solver CVC4 [3] and the theory of bit vectors. We simultaneously aim at:

- offering to CVC4 users the possibility to formally check its answers in a trusted environment like Coq;
- bringing the power of a versatile and widely used SMT solver like CVC4 to Coq;
- providing in Coq a decision procedure for bit vectors, a theory widely used, for instance, for verifying circuits or programs using machine integers.

2 The SMTCoq Tool

2.1 General Idea

The heart of SMTCoq is a checker for a generic format of certificates (close to the format proposed by Besson *et al.* [6]), implemented and proved correct inside Coq (see Figure 1a). Taking advantage of Coq's computational capabilities the SMTCoq checker is fully executable, either inside Coq or after extraction to a general-purpose language [18].

The Coq signature of this checker is the following:

```
checker : formula → certificate → bool
```

where the type `formula` represents the deep embedding in Coq of SMT formulas, and the type `certificate` represents SMTCoq's format of certificates.

The checker's soundness is stated with respect to a translation function from the deep embedding of SMT formulas into Coq terms:

```
⟦•⟧ : formula → bool
```

that interprets every SMT formula into its Coq Boolean counterpart. The correctness of the checker:

```
checker_sound : ∀ f c, checker f c = true → ⟦f⟧
```

thus means that, given a formula and a certificate for which the checker answers positively, then the interpretation in Coq of the formula is valid.

The choice of the type of Booleans `bool` as the codomain of the translation function ⟦•⟧, instead of the type of (intuitionistic) propositions `Prop`, allows us to handle the checking of the classical reasoning made by SMT solvers without adding any axioms. The SSReflect [12] plugin for Coq can be used to bridge the gap between propositions and Booleans for the theories considered by SMTCoq. The major shortcoming of this approach is that it does not allow quantifiers inside goals sent to SMT solvers, although it does not prevent one from feeding these solvers universally quantified lemmas. To increase the expressivity of SMTCoq with respect to quantifiers, one will need to switch to propositions, and handle classical logic either by axioms or by restricting attention to decidable atoms of the considered combined theory.

The first use case of this correct-by-construction checker is to check the validity of a proof witness, or proof *certificate* coming from an external solver against some input problem (Figure 1b). In this use case, the trusted base is both Coq and the parser of the input problem. The parse is part of the trusted based because we need to make sure we are effectively verifying a proof of the problem we sent to the external solver. However, this parser is fairly straightforward.

The second use case is within a Coq tactic (Figure 1c). We can give a Coq goal to an external solver and get a proof certificate for it. If the checker can validate the certificate, the soundness of the checker allow us to establish a proof of the initial goal. This process is known as *computational reflection* as it uses a computation (here, the execution of the checker) inside a proof. In this use case, the trusted base consists only of Coq: if something else goes wrong (e.g., the checker cannot validate the certificate), the tactic will fail, but nothing unsound will be added to the system.

In both cases, a crucial aspect for modularity purposes is the possibility to *preprocess* proof certificates before sending them to the SMTCoq checker, without having to prove anything about this preprocessing stage. Again, if the preprocessor is buggy, the checker will fail to validate the proof certificate (by returning `false`), which means that while nothing is learned, nothing unsafe is added to Coq's context. This allows us to easily extend SMTCoq with new solvers: as long as the certificate coming from the new solver can be logically encoded into SMTCoq's certificate format, we can implement this encoding at the preprocessing stage. As a result, SMTCoq's current support for both ZChaff and veriT is provided through the implementation of a preprocessor for each solver. Both preprocessors convert to the same proof format, thus sharing the same checker.

Using a preprocessor is also beneficial for efficiency: proof certificates may be encoded more compactly before being sent to the SMTCoq checker, which may improve performance.

2.2 The Checker

We now provide more details on the checker of SMTCoq. As presented in Figure 2, it consists of a *main checker* obtained as the combination of several *small checkers*, each specialized in one aspect of proof checking in SMT (e.g., CNF conversion, propositional reasoning, reasoning in the theory of equality, linear arithmetic reasoning, and so on).

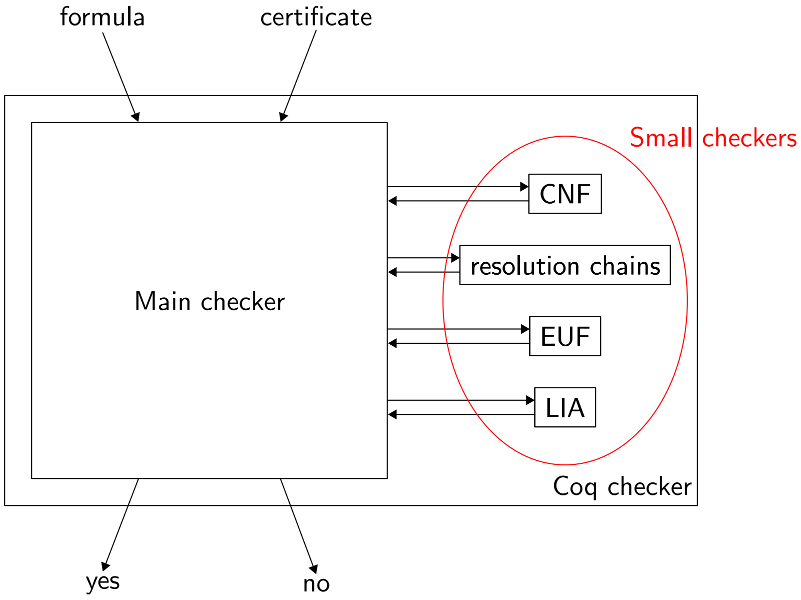

Figure 2: Internals of the Coq checker

The type `certificate` is actually the aggregation of specialized types, one for each small checker. The role of the main checker is thus to dispatch each piece of the certificate to its dedicated small checker, until the initial formula is proved.

A small checker is a Coq program that, given a (possibly empty) list of formulas and a certificate associated with it (which may be just a piece of the input certificate), computes a new formula:

```
small_checker : list formula → certificate_sc → formula
```

The soundness of the `checker` comes from the soundness of each small checker, stated as follows:

```
small_checker_sound : ∀ f₁ ... fₙ c,
    ⟦f₁⟧ ∧ ... ∧ ⟦fₙ⟧ → ⟦small_checker [f₁;...;fₙ] c⟧
```

meaning that the small checker returns a formula which is implied (after translation into Coq's logic) by the conjunction of its premises. Note that the list of premises may be empty: in such a case, the small checker returns a tautology in Coq.

Here are some examples of small checkers.

- For propositional resolution chains, the checker takes as input a list of premises and returns a resolvent if it exists, or a trivially true clause otherwise. In this case, a certificate is not required as part of the small checker's input.
- For the theory of equality with uninterpreted functions (EUF), the checker takes as input a formula in this theory formulated as a certificate (corresponding to a theory lemma produced by the SMT solver), and returns the formula if it is able to check it, or a trivially true clause otherwise. In this case, no premises are given.

- For linear integer arithmetic (LIA), the checker works similarly to the EUF checker, but checks the formula using Micromega [5], an efficient decision procedure for this theory implemented in Coq.

The only thing that small checkers need to share is the type `formula`, and its interpretation into Coq Booleans. Each small checker may then reason independently, using separate pieces of the certificate. Again, this is crucial for modularity: to extend SMTCoq with a new theory, one only has to extend the type `formula` with the signature of this theory and, independently of the already existing checkers, implement a small checker for this theory and prove its soundness.

Notice that "small checker" can be understood in a very general sense: any function that, given a list of first-order formulas, returns an implied first-order formula, can be plugged into SMTCoq as a small checker. In principle, such a checker could even be as complex as an SMT solver, as long as it can be proved correct in Coq.

3 Work in Progress: Extensions to CVC4 and Bit Vector Arithmetic

3.1 Support for CVC4

CVC4 is a proof-producing SMT solver, whose proof format is based on the Logical Framework with Side Conditions (LFSC) [21]. LFSC extends the Edinburgh Logical Framework (LF) [14] by allowing types with computational *side conditions*, explicit computational checks defined as programs in a small but expressive functional first-order programming language. The language has built-in types for arbitrary precision integers and rationals, ML-style pattern matching over LFSC type constructors, recursion, a minimal support for exceptions, and a very restricted set of imperative features. One can define proof rules in LFSC as typing rules that may optionally include a side condition written in this language. When checking the application of such proof rules, an LFSC checker computes actual parameters for the side condition and executes its code; if the side condition fails, the LFSC checker rejects the rule application. The validity of an LFSC proof witness thus relies on the correctness of the side condition functions used in the proof. LFSC comes with a set of pre-defined side conditions for various theories, used by the CVC4 proof production mechanism.

The key differences between LFSC and the SMTCoq format are presented in Table 1.

	LFSC	SMTCoq
Rules	deduction + computation	deduction + certificate
Nested proofs	supported	not supported

Table 1: Main differences between the LFSC and SMTCoq certificate formats

The major difference lies in the presentation of the deduction rules. In SMTCoq, the small checkers deduce a new formula from already known formulas, possibly with the help of a piece of certificate that depends on the theory. The LFSC format is more uniform, thanks to the side conditions described above.

To support LFSC, and so CVC4, we are in the process of implementing (in OCaml) an untrusted preprocessor that transforms LFSC proofs into SMTCoq proofs. To this end, for some theories, we need to replay parts of the side conditions, in order to produce the corresponding SMTCoq premises, conclusion and piece of certificate that will be passed to the small checkers. This encoding, however, is relatively straightforward:

- for propositional reasoning, LFSC side conditions use the same logical content as SMTCoq rules;

- CNF conversion and EUF proofs are nested in LFSC, so they require some processing for the moment;
- for linear integer arithmetic, since SMTCoq relies on an existing decision procedure in Coq, it only needs to know what theory lemma is being proved, and can ignore the actual proof steps in the LFSC certificate.

One difficulty in translating LFSC proofs to the SMTCoq format comes from to the possibility in LFSC of using natural-deduction-style proofs, where one can nest one proof inside another. For instance, it is possible to have lemmas inside an LFSC proof whose witnesses are themselves LFSC proofs. The architecture of the main and small checkers of SMTCoq does not currently allow this sort of nesting: every clause produced by the small checkers needs to be a direct consequence of input clauses or clauses that were previously produced. To encode an LFSC proof into SMTCoq, our preprocessor thus linearizes nested proofs. The LFSC proofs generated by CVC4 are constructed in such a way that this does not cause a blow-up in practice; however, to support LFSC in general, we plan to extend SMTCoq certificates with nested proofs. Again, this extension should be made easier by the modularity inside the checker. It should impact only the main checker, and not the various small checkers already in SMTCoq.

3.2 Support for Bit Vector Arithmetic

CVC4 has been recently extended to produce LFSC proofs for the quantifier-free fragment of the SMT theory of bit vectors [13]. To check proof certificates in this theory, SMTCoq needs be extended with it. As explained in Section 2.2, to do that one needs to:

1. extend the Coq representation of formulas with the signature of the bit vector theory and the interpretation function into Coq terms;
2. implement (new) small checkers and their corresponding certificates for this theory, and prove their correctness.

Step 1 is a simple extension on the SMTCoq side. The major difficulty is that Coq itself has limited support for bit vectors. Its bit vector library provides only the implementation of bitwise operations (and not arithmetic operations), and no proofs. We are thus currently implementing a more complete library for this theory. Step 2 involves implementing and adding new certified Coq programs (the small checkers). As mentioned, however, because of SMTCoq's design, none of the previous small checkers and their proofs of correctness need to be changed as a result of this addition.

LFSC proofs for bit vectors produce by CVC4 mainly involve the following two kinds of deduction steps:

- *bit-blasting* steps that reduce the input bit vector formula to an equisatisfiable propositional formula;
- standard propositional reasoning steps (based on resolution).

The propositional steps can be handled directly by previous small checkers. For the bit-blasting steps, we implemented new small checkers that relate terms of the bit vector theory with lists of Boolean formulas representing their bits; we are currently working on producing proofs of correctness in Coq for these small checkers.

LFSC proofs generated by CVC4 involve a third kind of step: formula simplifications based on the equivalence of two bit-vector terms or atomic formulas (for instance, by normalizing inequalities). Currently, these simplification steps are not provided a detailed LFSC subproof by CVC4, although there are plans to do so in the near future. In the current SMTCoq implementation then, we assume those steps, as in the LFSC proof coming from CVC4, or let the user prove them, in the case of tactics. Since those steps correspond to applications of CVC4-defined rewriting and simplification rules, we plan for now to prove the correctness of these rules once and for all at the Coq level, and to pre-process simplification steps into applications of these rules.

4 Related Work

In addition to related work already discussed throughout the paper, we now briefly mention a few more notable projects. Heule *et al.* implemented an efficient checker for state-of-the-art SAT techniques, verified in ACL2 [15, 24]. It is mainly based on a generalization of extended resolution [22, 17] and on reverse unit propagation [11]. SMTCoq currently handles only standard extended resolution for its propositional part.

Efficient proof reconstruction for SAT and SMT solvers has been implemented in proof assistants based on higher-order logic [23, 9]. Some of these reconstructions also handle the theory of bit vectors [8]. This approach is based on translating SAT/SMT certificates to applications of the inference rules of the kernels of these proof assistants. In contrast, our approach in Coq is based on computational reflection: the certificate is directly processed by the reduction mechanism of Coq's kernel.

Based on an efficient encoding of a large subset of HOL goals into first-order logic, the Sledgehammer tactic [20] allows HOL-based proof assistants to efficiently and reliably help manual proving. Proofs are replayed using either the proof reconstruction mechanism described above or a built-in first-order prover. We hope that SMTCoq can help in adding such techniques into Coq and other Type Theory-based proof assistants, by providing a proof replay mechanism based on certificates.

5 Conclusion and Future Work

SMTCoq has been designed to be modular in such a way that facilities its extension with new solvers and new theories. In particular, such extensions should not require any changes in existing checkers or in their proofs of soundness. Thus, while it may require some effort to certify new small checkers or to translate new proof formats into the SMTCoq format, such extensions require only local changes. Our current extensions to CVC4 and bit vectors arithmetic validate this goal: indeed, the work so far consisted mostly in implementing an untrusted preprocessor for certificates and adding new, independent checkers. One limiting aspect of SMTCoq is the lack of support for nested proofs, which we plan to add. Thanks to the modularity of the checker, we believe this feature too can be added locally.

In the future we plan to continue extending the expressivity of SMTCoq, and in particular to offer support for the SMT theory of arrays (for which CVC4 is also proof-producing). We believe we can match, and perhaps even improve upon existing work in terms of efficiency.

The current major limitation of SMTCoq resides in its set of tactics: presently, it can only handle goals that are directly provable by SMT solvers, without much encoding of Coq logic into first-order logic. Our longer term plan is to combine ongoing work on *hammering* [7] for proof assistants based on Type Theory (such as Coq) with the certificate checking capabilities offered by SMTCoq.

Acknowledgments. We wish to thank the anonymous reviewers for their helpful and constructive feedback. This work was supported in part by the Air Force Research Laboratory (AFRL) and the Defense

Advanced Research Projects Agency (DARPA) under contracts FA8750-13-2-0241 and FA8750-15-C-0113. Any opinions, findings, and conclusions or recommendations expressed above are those of the authors and do not necessarily reflect the views of AFRL or DARPA.

References

[1] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry & Benjamin Werner (2011): *A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses*. In Jouannaud & Shao [16], pp. 135–150, doi:10.1007/978-3-642-25379-9_12.

[2] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliâtre, E. Gimenez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy et al. (2000): *The Coq proof assistant: reference manual*. Technical Report, INRIA.

[3] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds & Cesare Tinelli (2011): *CVC4*. In Ganesh Gopalakrishnan & Shaz Qadeer, editors: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, *Lecture Notes in Computer Science* 6806, Springer, pp. 171–177, doi:10.1007/978-3-642-22110-1_14.

[4] Clark Barrett, Aaron Stump & Cesare Tinelli (2010): *The SMT-LIB Standard: Version 2.0*. In A. Gupta & D. Kroening, editors: *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*.

[5] F. Besson (2006): *Fast Reflexive Arithmetic Tactics the Linear Case and Beyond*. In Thorsten Altenkirch & Conor McBride, editors: *TYPES*, *Lecture Notes in Computer Science* 4502, Springer, pp. 48–62, doi:10.1007/978-3-540-74464-1_4.

[6] F. Besson, P. Fontaine & L. Théry (2011): *A Flexible Proof Format for SMT: a Proposal*. In: *PxTP 2011: First International Workshop on Proof eXchange for Theorem Proving August 1, 2011 Affiliated with CADE 2011, 31 July-5 August 2011 Wrocław, Poland*, pp. 15–26.

[7] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson & Josef Urban (2016): *Hammering towards QED*. *J. Formalized Reasoning* 9(1), pp. 101–148, doi:10.6092/issn.1972-5787/4593.

[8] Sascha Böhme, Anthony C. J. Fox, Thomas Sewell & Tjark Weber (2011): *Reconstruction of Z3's Bit-Vector Proofs in HOL4 and Isabelle/HOL*. In Jouannaud & Shao [16], pp. 183–198, doi:10.1007/978-3-642-25379-9_15.

[9] Sascha Böhme & Tjark Weber (2010): *Fast LCF-Style Proof Reconstruction for Z3*. In Matt Kaufmann & Lawrence C. Paulson, editors: *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, *Lecture Notes in Computer Science* 6172, Springer, pp. 179–194, doi:10.1007/978-3-642-14052-5_14.

[10] T. Bouton, D.C.B. de Oliveira, D. Déharbe & P. Fontaine (2009): *veriT: An Open, Trustable and Efficient SMT-Solver*. In R. A. Schmidt, editor: *CADE*, *Lecture Notes in Computer Science* 5663, Springer, pp. 151–156, doi:10.1007/978-3-642-02959-2_12.

[11] Allen Van Gelder (2012): *Producing and verifying extremely large propositional refutations - Have your cake and eat it too*. *Ann. Math. Artif. Intell.* 65(4), pp. 329–372, doi:10.1007/s10472-012-9322-x.

[12] Georges Gonthier & Assia Mahboubi (2010): *An introduction to small scale reflection in Coq*. *J. Formalized Reasoning* 3(2), pp. 95–152, doi:10.6092/issn.1972-5787/1979.

[13] Liana Hadarean, Clark W. Barrett, Andrew Reynolds, Cesare Tinelli & Morgan Deters (2015): *Fine Grained SMT Proofs for the Theory of Fixed-Width Bit-Vectors*. In Martin Davis, Ansgar Fehnker, Annabelle McIver & Andrei Voronkov, editors: *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, *Lecture Notes in Computer Science* 9450, Springer, pp. 340–355, doi:10.1007/978-3-662-48899-7_24.

[14] Robert Harper, Furio Honsell & Gordon D. Plotkin (1993): *A Framework for Defining Logics*. *J. ACM* 40(1), pp. 143–184, doi:10.1145/138027.138060.

[15] Marijn Heule, Warren A. Hunt Jr. & Nathan Wetzler (2013): *Verifying Refutations with Extended Resolution*. In Maria Paola Bonacina, editor: *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, *Lecture Notes in Computer Science* 7898, Springer, pp. 345–359, doi:10.1007/978-3-642-38574-2_24.

[16] Jean-Pierre Jouannaud & Zhong Shao, editors (2011): *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*. *Lecture Notes in Computer Science* 7086, Springer, doi:10.1007/978-3-642-25379-9.

[17] Oliver Kullmann (1999): *On a Generalization of Extended Resolution*. *Discrete Applied Mathematics* 96-97, pp. 149–176, doi:10.1016/S0166-218X(99)00037-2.

[18] Pierre Letouzey (2002): *A New Extraction for Coq*. In Herman Geuvers & Freek Wiedijk, editors: *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers*, *Lecture Notes in Computer Science* 2646, Springer, pp. 200–219, doi:10.1007/3-540-39185-1_12.

[19] Yogesh S. Mahajan, Zhaohui Fu & Sharad Malik (2004): *Zchaff2004: An Efficient SAT Solver*. In Holger H. Hoos & David G. Mitchell, editors: *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*, *Lecture Notes in Computer Science* 3542, Springer, pp. 360–375, doi:10.1007/11527695_27.

[20] Lawrence C. Paulson & Jasmin Christian Blanchette (2010): *Three years of experience with Sledgehammer, a Practical Link Between Automatic and Interactive Theorem Provers*. In Geoff Sutcliffe, Stephan Schulz & Eugenia Ternovska, editors: *The 8th International Workshop on the Implementation of Logics, IWIL 2010, Yogyakarta, Indonesia, October 9, 2011*, *EPiC Series* 2, EasyChair, pp. 1–11. Available at `http://www.easychair.org/publications/?page=820355915`.

[21] Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean & Cesare Tinelli (2013): *SMT proof checking using a logical framework*. *Formal Methods in System Design* 42(1), pp. 91–118, doi:10.1007/s10703-012-0163-3.

[22] G. Tseitin (1970): *On the Complexity of Proofs in Propositional Logics*. In: *Seminars in Mathematics*, 8, pp. 466–483.

[23] T. Weber (2008): *SAT-based Finite Model Generation for Higher-Order Logic*. Ph.D. thesis, Institut für Informatik, Technische Universität München, Germany. Available at `http://www.cl.cam.ac.uk/~tw333/publications/weber08satbased.html`.

[24] Nathan Wetzler, Marijn Heule & Warren A. Hunt Jr. (2013): *Mechanical Verification of SAT Refutations with Extended Resolution*. In Sandrine Blazy, Christine Paulin-Mohring & David Pichardie, editors: *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, *Lecture Notes in Computer Science* 7998, Springer, pp. 229–244, doi:10.1007/978-3-642-39634-2_18.

Towards the Integration of an Intuitionistic First-Order Prover into Coq

Fabian Kunze
Saarland University
s9fakunz@stud.uni-saarland.de

An efficient intuitionistic first-order prover integrated into Coq is useful to replay proofs found by external automated theorem provers. We propose a two-phase approach: An intuitionistic prover generates a certificate based on the matrix characterization of intuitionistic first-order logic; the certificate is then translated into a sequent-style proof.

1 Introduction

Sledgehammer [11] and HOLyHammer [5] drastically improved the productivity for users of proof assistants. They make the capabilities of automated theorem provers (ATPs) available from within interactive proof assistants.

The large, monolithic design of state-of-the-art theorem provers can not be easily trusted to be free of bugs. Thus invoking theorem provers as an oracle is unacceptable for most users. Proof assistants are more trustworthy because all reasoning is checked by a kernel intentionally kept small.

To integrate external provers, small yet efficient, *certified* provers *integrated* into the proof assistant are used: Although it is often possible to mechanically translate the proof to a format accepted by the proof assistant, the integrated prover allows for the reconstruction without the full knowledge of all axioms and rules used by the external prover. Thus an integrated prover simplifies the integration of not only one but different external provers.

There has been effort to integrate classical provers into Coq, e.g. SMTCoq [1], Satallax [3] and why3 [2], but they produce proofs that assume classical axioms. As a fair amount of proof developments avoids assuming additional axioms, the acceptance of a future ‘Coq Hammer’ benefits from the integration of an efficient, *intuitionistic* prover.

2 Existing Intuitionistic Provers in Coq

The existing intuitionistic first-order provers integrated into Coq are not very strong. We evaluated `firstorder` [4], a built-in tactic based on a sequent calculus, and JProver [14], a plugin available for Coq. Using Coq version `8.6pl1`, we considered first-order problems that are likely to emerge in a future ‘Coq Hammer’.

For example, we tested formulas where the instantiate of quantifiers is not immediately determined using a goal-driven approach:

$$\begin{aligned}&(\forall x, x = x) \land (\forall x, Px \lor Qx)\\ \land\ &(\forall xy, x = y \land Px \Rightarrow Ry) \land (\forall xy, x = y \land Qx \Rightarrow Ry) \Rightarrow (\forall x, Rx).\end{aligned}$$

J.C. Blanchette and C. Kaliszyk:
Hammers for Type Theories (HaTT’16)
EPTCS 210, 2016, pp. 30–35, doi:10.4204/EPTCS.210.6

On this formula, `firstorder` was unable to find a proof during the several minutes we run it. JProver succeeded in less than one second.

We also invoked both provers on several set-theoretical problems from the ILTP (Intuitionistic Logic Theorem Proving) library [12]. Similar to the intended use case, we only supplied the axioms needed for the proofs, resulting in problems like

$$\begin{aligned}&(\forall ABX, X \in A \cup B \Leftrightarrow X \in A \vee X \in B)\\ \wedge\ &(\forall AB, A = B \Leftrightarrow A \subseteq B \wedge B \subseteq A)\\ \wedge\ &(\forall AB, A \subseteq B \Leftrightarrow \forall X, X \in A \Rightarrow X \in B) \Rightarrow (\forall A, A \cup A = A).\end{aligned}$$

On this and similar problems, both `firstorder` and JProver failed to find proofs before we aborted them after running several minutes.

Therefore, faster intuitionistic provers integrated into Coq are necessary for a ‘Coq Hammer’ used in practice.

3 Proposed Architecture

We propose to employ the recent improvements on automated, intuitionistic first-order theorem proving by Otten: ileanCoP [7, 8] and the forthcoming intuitionistic version of nanoCoP [10, 9]. The existing implementations of both provers verified that the formulas in Section 2 are valid in under a second. Both provers are based on the existence of proof certificates for the matrix characterization of (intuitionistic) validity [15], which can be translated to sequent-style proofs [13].

This architecture is similar to that of JProver (which uses the same characterization of validity), but uses a more efficient proof search procedure, leading to a higher success rate.

3.1 Finding Proof Certificates

The performance of ileanCoP is well in identifying valid formulas, compared to other intuitionistic provers [8]. But it does not keep track of the proof found. Furthermore, it is based on a *clausal* variant of the matrix characterization for intuitionistic logic. The necessary translation into a non-clausal matrix proof has been sketched in the correctness proof of ileanCoP [7], but to our knowledge has not yet been implemented.

The classical prover nanoCoP [10] solves both problems: It outputs the proof certificate found and uses the non-clausal matrix characterization of classical validity. Otten is currently extending nanoCoP to an intuitionistic variant by integrating prefix unification [15], a method already employed to derive ileanCoP from the classical prover leanCoP.

In our proposed architecture, the proof certificate for a first-order formula F consists of a *multiplicity* μ, a pair of substitutions $\sigma = (\sigma_Q, \sigma_J)$ and a set of pairs of σ-*complementary* atoms in the formula (connections) that *spans* F^μ.

We will now give an very informal intuition about this certificate.

A Part of the certificate is already needed for the matrix characterization of classical logic: The multiplicity μ takes care of the multiple instances an all-quantified subformula of F may be needed in the proof. One part of ‘σ-complementary’ ensures that that two atoms in a connection are identical under the (non-circular) term substitution σ_Q, but have different polarity.

The set of connections *spans* the formula if every *path* through the formula contains at least one connections. In the quantifier-free case, each path correspond to a disjunction in the conjunctive normal form. In the case of formulas with quantifiers, each path correspond a branch of a (classical) analytic tableaux, where quantifiers are instantiated according to σ_Q.

The main difference in the intuitionistic characterization is the use of σ_J to ensure that the positions of the pair of complementary atoms in the formula are 'compatible'. The position of an atom is defined by structural recursion on the formula and represented by a string, consisting of fresh constants and fresh variables.

An example of this for an intuitionistic valid formula is $P \Rightarrow P$, where the two atoms can be made complementary: The position of the first P is described by the string z with a fresh variable z, while the position of the second P is the string a consisting of a fresh constant a. Defining $\sigma_J(z) = a$ unifies those strings.

For the formula $\neg P \vee P$, a theorem of classical, but not intuitionistic logic, the two atoms can not be made complementary: The position of the first P is described by xa, while the one for the second P is b. As the second position contains no variable and no a, we can never unify those strings.

This concept generalizes to quantified formulas, but for the main idea, it suffices to study the cases for non-quantified formulas.

For a more formal definition and a few more examples, we recommend the first two Sections of [7], and Chapter 8, §4 of [15].

It should be noted that one of the main improvements of nanoCoP compared to JProver is the handling of the multiplicities: nanoCoP adds instances of subformulas during the proof search as needed, while JProver fixes the multiplicity before searching for an proof; on failure, an additional instance of the whole formula gets added and the proof is retried. Although both are complete, the first approach is more goal-driven and thus expected to be more efficient.

3.2 Generating Sequence Proofs

The high-level idea is that the proof certificate guarantees that on each branch of the sequent-style proof, eventually complementary atoms are found. The difficulty is to traverse the formulas in the right order, which depends on σ_J.

The translation of a matrix characterisation proof certificate into a sequent-style proof has already been investigated and implemented for JProver[13]. We intend to adopt this translation, as we expect it to be reasonable fast: In the examples we tried and where JProver succeeded, the sequence-style proof produced was rather short. In the cases where JProver did not succeed in an acceptable time, it did not even reach the sequence-proof generation. Thus we conclude that the bottleneck of JProver, at least in the examples we tried, is the proof certificate search.

4 Discussion

Modular vs Monolithic

We explicitly want to use a modular implementation for the two phases, possibly written in multiple languages. The Prolog version of the intuitionistic variant of nanoCoP is expected to materialize soon and there is already an implementation of the sequence proof generating algorithm integrated into Coq. Thus we expect no challenge in creating a prototype of the suggested architecture using the Prolog program. This would allow us to test whether the proposed setup is suitable for the intended use case.

In the longer term, it would be desirable to have a native OCaml implementation of the proof search procedure, allowing for a deployment within Coq, without additional binaries. The classical leanCoP has been ported to OCaml for the HOL light proof assistant, with performence comparable to the Prolog version[6]. This port can serve as a starting point for a native OCaml version of the forthcoming intuitionistic nanoCoP. Then, the modular approach allows to optionally use external proof procedures. This allows to evaluate improvements to the Prolog proof procedure before porting them.

Also, a modular design allows to more easily use parts an implementation this for other, intuitionistic proof assistants. This additional usage should be kept in mind while developing this, and other, tools towards Hammers in Type Theory.

Explicit Proofs vs Reflection

One approach in proof automation in Coq is 'proof by reflection': Some or all parts of the the proof search procedure are written in Coq, including a correctness proof. The proof of a statement then *is* the call to this Coq procedure.

One argument for 'proof by reflection' in Coq is the efficiency. But this is just a benefit compared to an implementation using Ltac, the tactic language in Coq: The evaluation of native Coq terms is heavily optimized to the extend of native machine code compilation and execution. In contrast, Ltac is just interpreted on top of several layers of abstraction. As we propose to use OCaml, not Ltac, for the computationally intense parts, this argument does not apply here.

We assume that the search for the proof certificate could be more easily written, modified, or enriched with heuristics, when using a language allowing side effect. This discourages the use of reflection in the first part of our proposed architecture.

Reflection seems to be more reasonable for the second part, the translation to a sequence proof: There is no need to explicitly generate the sequence proof when a certified procedure guarantees that the sequence proof *does* exist when the certificate satisfies the appropriate conditions.

The challenge here would be that the proof certificate must annotated with type information rich enough to reduce to proofs for all formulas we intend to proof: This means that when the terms in the formula are not single sorted, but have of more complex types, e.g. dependent types, this must be incorporated in the proof certificate, the translation procedure itself and its correctness proof. At first, it seems that a benefit would be that the translation is proven to be sound by design. But to check the conditions that a proof certificate is indeed valid is more or less computationally equivalent hard as to generating a sequence-style proof.

Another aspect to consider is that some usage, a formula that is not first-order can be transformed into an first-order formula such that a proof of the later formula can be translated back to a proof of the former formula. In a reflective proof reconstruction, this intermediate steps may can not type-check.

Intuitionistic vs Classical

Automated theorem proving in intuitionistic logic is computationally harder than in classical logic. For developments assuming classical axioms, the intuitionistic part of both phases can be made optional, resembling the classical proof search of nanoCoP without significant overhead.

Note that the proof search in this proposed architecture does neither need skolemization nor clausal normal forms. Thus more structure of the different lemmas and parts of the formulas is preserved and in some sense, this approach is closer to humans reasoning. Further investigation of this architecture could lead to insights useful for automated reasoning in proof assistants of classical logic.

Acknowledgements

We thank Jens Otten for his helpful discussions and suggestions, and Jasmin Blanchette and the anonymous reviewers for their comments on this extended abstract.

References

[1] Michael Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry & Benjamin Werner (2011): *Certified Programs and Proofs: First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, chapter A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses, pp. 135–150. Springer Berlin Heidelberg, Berlin, Heidelberg. Available at `http://dx.doi.org/10.1007/978-3-642-25379-9_12`.

[2] François Bobot, Jean-Christophe Filliâtre, Claude Marché & Andrei Paskevich (2011): *Why3: Shepherd Your Herd of Provers*. In K. Rustan M. Leino & Michał Moskal, editors: *Boogie 2011*, pp. 53–64.

[3] Chad E. Brown (2012): *Satallax: An Automatic Higher-Order Prover*. In Bernhard Gramlich, Dale Miller & Uli Sattler, editors: *Automated Reasoning—6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, *Lecture Notes in Computer Science* 7364, Springer, pp. 111–117. Available at `http://dx.doi.org/10.1007/978-3-642-31365-3_11`.

[4] Pierre Corbineau (2004): *First-Order Reasoning in the Calculus of Inductive Constructions*, pp. 162–177. Springer Berlin Heidelberg, Berlin, Heidelberg. Available at `http://dx.doi.org/10.1007/978-3-540-24849-1_11`.

[5] Cezary Kaliszyk & Josef Urban (2015): *HOL(y)Hammer: Online ATP Service for HOL Light*. *Mathematics in Computer Science* 9(1), pp. 5–22. Available at `http://dx.doi.org/10.1007/s11786-014-0182-0`.

[6] Cezary Kaliszyk, Josef Urban & Jiři Vyskočil (2015): *Certified Connection Tableaux Proofs for HOL Light and TPTP*. In: *Proceedings of the 2015 Conference on Certified Programs and Proofs*, CPP '15, ACM, New York, NY, USA, pp. 59–66. Available at `http://doi.acm.org/10.1145/2676724.2693176`.

[7] Jens Otten (2005): *Clausal Connection-Based Theorem Proving in Intuitionistic First-Order Logic*. In Bernhard Beckert, editor: *Automated Reasoning with Analytic Tableaux and Related Methods: 14th International Conference, TABLEAUX 2005, Koblenz, Germany, September 14-17, 2005. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 245–261. Available at `http://dx.doi.org/10.1007/11554554_19`.

[8] Jens Otten (2008): *leanCoP 2.0 and ileanCoP 1.2: High Performance Lean Theorem Proving in Classical and Intuitionistic Logic (System Descriptions)*. In Alessandro Armando, Peter Baumgartner & Gilles Dowek, editors: *Automated Reasoning: 4th International Joint Conference, IJCAR 2008 Sydney, Australia, August 12-15, 2008 Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 283–291. Available at `http://dx.doi.org/10.1007/978-3-540-71070-7_23`.

[9] Jens Otten (2016): personal communication.

[10] Jens Otten (2016): *nanoCoP: A Non-clausal Connection Prover*. In: *Automated Reasoning: 8th International Joint Conference, IJCAR 2016 Coimbra, Portugal, June 27 -July 2, 2016 Proceedings.* To appear.

[11] Lawrence C. Paulson & Jasmin Christian Blanchette (2010): *Three years of experience with Sledgehammer, a Practical Link Between Automatic and Interactive Theorem Provers*. In Geoff Sutcliffe, Stephan Schulz & Eugenia Ternovska, editors: *The 8th International Workshop on the Implementation of Logics, IWIL 2010, Yogyakarta, Indonesia, October 9, 2011*, *EPiC Series* 2, EasyChair, pp. 1–11. Available at `http://www.easychair.org/publications/?page=820355915`.

[12] Thomas Raths, Jens Otten & Christoph Kreitz (2007): *The ILTP Problem Library for Intuitionistic Logic: Release v1.1*. *Journal of Automated Reasoning* 38(1-3), pp. 261–271. Available at `http://dx.doi.org/10.1007/s10817-006-9060-z`.

[13] Stephan Schmitt & Christoph Kreitz (1996): *Converting non-classical matrix proofs into sequent-style systems*. In: *Automated Deduction — Cade-13: 13th International Conference on Automated Deduction New Brunswick, NJ, USA, July 30 – August 3, 1996 Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 418–432. Available at `http://dx.doi.org/10.1007/3-540-61511-3_104`.

[14] Stephan Schmitt, Lori Lorigo, Christoph Kreitz & Alexey Nogin (2001): JProver *: Integrating Connection-based Theorem Proving into Interactive Proof Assistants*. In R. Gore, A. Leitsch & T. Nipkow, editors: *International Joint Conference on Automated Reasoning, Lecture Notes in Artificial Intelligence* 2083, Springer Verlag, pp. 421–426, doi:10.1007/3-540-45744-5_34.

[15] Lincoln Wallen (1990): *Automated Deduction in Nonclassical Logics*. MIT Press, Cambridge, MA, USA.