# Cooperative Integration of an Interactive Proof Assistant and an Automated Prover

Adam Chlipala[1,3,4] George C. Necula[2,4]

*Computer Science Division*
*University of California, Berkeley*
*Berkeley, California, USA*

**Abstract**

We propose a mechanism for semi-automated proving of theorems, using a tactic for the Coq proof assistant that consults a proof-generating Nelson-Oppen-style automated prover. Instead of simply proving or failing to prove a goal, our tactic decides on relevant case splits using theory-specific axioms, proves some of the resulting cases, and returns the remainder to the Coq user as subgoals. These subgoals can then be proved using inductions and lemma instantiations that are beyond the capabilities of the automated prover. We show that the Coq tactic language provides an excellent way to script this process to an extent not supported by current Nelson-Oppen provers. Like with any Coq proof, a separately checkable proof term in a core calculus is produced at the end of any successful proving session where our method is used, and we take advantage of the "proof by reflection" technique to translate the specialized first-order proofs of the automated prover into compact Coq representations.

*Keywords:* integration of interactive and automatic theorem proving, proof by reflection

## 1 Introduction

When proving properties of software or hardware systems, one is faced with a mix of proof obligations that typically includes many shallow facts drawn from a small number of decidable theories, such as the theories of equality, linear arithmetic, or uninterpreted functions; along with deep facts that require high-level proof techniques, such as induction, universal quantifier instantiation, or even higher-order reasoning. A recent study [11] suggests that to deal with such a mix of proof obligations one would need to combine the strengths of automated proving, such as Nelson-Oppen-based theorem provers; with those of interactive proof assistants, such as Coq.

The Nelson-Oppen architecture for cooperating decision procedures [18] supports the effective combination of separately-authored decision procedures for first-order

1 Email: adamc@cs.berkeley.edu
2 Email: necula@cs.berkeley.edu
3 Supported in part by a National Defense Science and Engineering Graduate Fellowship
4 Supported in part by NSF Grants CCF-0524784, CCR-0234689, CNS-0509544, and CCR-0225610

theories. Well-engineered Nelson-Oppen provers can discharge most of the shallow proof obligations very efficiently, with no user intervention. Examples of tools that take advantage of this technique include program verifiers in the style of Extended Static Checking [10,12] and counterexample-guided software model checkers [14].

A serious weakness of Nelson-Oppen provers is that the user must take care to model the system being verified in such a way that the resulting proof obligations are sufficiently simple. Furthermore, the automation of Nelson-Oppen provers is also their weakness. The only recourse one has when the Nelson-Oppen prover is unable to complete the task is to change the theorems that need to be proved, by changing the system being verified, its modeling, or its specification. This is often a very frustrating trial-and-error process.

The Extended Static Checking (ESC) family of program verification tools exemplifies one pattern of human input to automated theorem provers. The source code of a program to be verified is marked up with loop invariants and function pre- and postconditions, in the form of special comments. Some of these annotations make up the specification that the user wants to be sure that the program obeys, but most of them can be viewed as suggesting a proof strategy. The user chooses the strategy through a process of trial and error, incrementally modifying these special comments and re-running the verification tool. Choosing the inductive structure of proofs is one of the most challenging tasks an automated prover faces, so it is no surprise that this is where an ESC user is called on to provide advice.

An alternative interaction model is associated with *interactive proof assistants*, such as Coq [8], Isabelle/HOL [21], and PVS [22]. These systems provide a great deal of freedom in the choice of the logical ingredients used for modeling and specifying systems. They also provide elaborate interaction models in which the user can direct the proof search. Most of these systems also provide extensive mechanisms to automate proof tasks by means of tactics, but in most cases the extent of the resulting automation is smaller than what can be obtained with Nelson-Oppen provers. It's also true that effective use of an interactive proof assistant requires considerably more training than is needed for a Nelson-Oppen prover, so it is quite beneficial to minimize the portion of a proving task that must be performed in traditional interactive style.

### *1.1 Our Contribution*

In this paper, we will argue that automated theorem proving (ATP) and interactive theorem proving (ITP) can be combined synergistically. In particular, we will describe a prototype system for interfacing the proof assistant Coq with Kettle, a proof-generating Nelson-Oppen prover that we have developed and used previously for such applications as proof-carrying code. Our prototype allows for bidirectional interaction between ATP and ITP provers, in contrast to the first-order logic tactics that are available for many proof assistants.

Our approach improves the state of the art of first-order ATP for program verification. The main challenge in this area is choosing the right instantiations for quantified axioms; in particular, it is difficult to find the right balance between completeness and efficiency/termination. We suggest that instantiations that can-not possibly lead to non-termination should be made automatically, and ITPs can be used to let a human suggest other necessary instantiations and possibly continue the process by invoking the ATP recursively.

We also suggest a new automation idiom of interest to the interactive and higher-order theorem proving communities. Proof assistants often contain "all or nothing" tactics for proving first-order goals. We propose that it can be advantageous to allow such tactics to return sets of simpler subgoals, representing cases that they weren't able to handle, where the ATP is responsible for choosing effective case splits that may not be syntactically apparent in the original goal. Traditional automated provers rely on very general types of reasoning heuristics, and the tactic languages of proof assistants like Coq provide an excellent complement, allowing for the rapid coding of specialized heuristics. Allowing the use of customized tactics *inside* of first-order proof searches allows simpler production of proof scripts than with monolithic calls to first-order tactics.

In the next section, we briefly overview related work and highlight the differences from our approach. Then, in Section 3, we review and contrast the important properties of Nelson-Oppen provers and the Coq proof assistant. In Section 4, we present an example that demonstrates our approach to semi-automatic program verification. Section 5 outlines the algorithmic aspects of our approach, and Section 6 describes how we encode Kettle-generated proofs as Coq proofs compactly using proof by reflection. Section 7 summarizes our implementation and case studies.

## 2 Related Work

The PVS proof assistant [22] comes with strategies that make use of various decision procedures to make simplifications, with an effect similar to what our Kettle tactic provides. However, we're not aware of any support for using matching rules to trigger case analyses, prove some cases, and ask the user to prove the others. PVS also doesn't generate proof terms in a small core calculus that admits simple checking, while the generation of such proofs is a main benefit of our technique.

Many proof-generating automated deduction tools have some support for automating proofs of purely first-order goals, including recent work on resolution proving for Coq [4] and Isabelle [16]. Resolution proving has also been integrated with a Martin-Löf-style logical framework [1]. As for automatic proofs of first-order goals relying on non-trivial ground theories, past projects have investigated the use of Nelson-Oppen proving with HOL [5] and HOL Light [15], generating higher-order logic proofs with the SVC decision procedure [23], reflection-based encoding of automatically-generated first-order proofs in Coq [7], an Isabelle/HOL interface to a Satisfiability Modulo Theories prover based on boolean satisfiability checking [13], and effective use of trusted external decision procedures by Coq [2]. None of these projects provides an interactive mode like ours or generates proofs that are as compact as our reflective proofs for goals involving arithmetic.

Nguyen et al. [19] have explored integrating a proof-generating rewriting tool with Coq. Their approach is complementary to ours, since rewriting and Nelson-Oppen-style proving are effective in different problem domains.

Boutin [6] provides a more general description of proof by reflection. Techniques very similar to those that we present in Section 6 are used in some tactics packaged with Coq, including a reflective version of the Omega decision procedure for Presburger arithmetic.

# 3 Preliminaries

## *3.1 Nelson-Oppen Provers*

Nelson-Oppen provers are based on an architecture for *cooperating satisfiability procedures*. There are three main aspects of the construction of procedures for different first-order theories: handling of atomic formulas, case splitting, and instantiation of quantified axioms.

At the most fundamental level, Nelson-Oppen provers provide an effective way to obtain a satisfiability procedure for conjunctions of literals belonging to several logical theories, by combining satisfiability procedures for the individual theories. Several important theories (e.g., linear arithmetic over rationals and uninterpreted functions) can be combined in this fashion.

Nelson-Oppen theorem provers can be extended easily with handling of disjunctive facts, by using case analysis and backtracking. This expands their applicability to include theories like the theory of arrays, whose usual formulation involves disjunctive reasoning. Nelson-Oppen provers have also been extended with a limited handling of quantifiers based on heuristics that decide when to instantiate quantifiers [9]. We will provide more detail in Section 5, where we present a standard Nelson-Oppen algorithm as one piece of our implementation.

Without lots of cleverness in the design of heuristics, instantiation along with case analysis can quickly blow up to the level where reasonably sized goals are intractable to prove. A major point of the new work we present here is that this kind of blow-up can be ameliorated in practice by getting the user of an interactive proof assistant involved in the selection of instantiations.

## *3.2 The Coq Proof Assistant*

The most important properties of the Coq proof assistant for our work are that it produces proof terms in a small core calculus and that it features an expressive tactic definition language.

Every successful Coq proving interaction produces a *proof term* in a small dependently-typed lambda calculus. Every tactic for Coq, including our tactic that interfaces with our Nelson-Oppen prover, must produce proofs in that formal language. Our prover Kettle produces proofs in a specialized language that includes many "high level" proof rules that involve running specialized algorithms. Any use of one of these rules has a straightforward translation into Coq proofs, but the obvious strategies can lead to asymptotic blow-ups in proof size. In Section 6, we discuss how we're able to implement these high-level rules in Coq in a way that gives us constant-overhead translation of applications of them.

Coq also features an untyped *tactic language* for building proof strategies com-

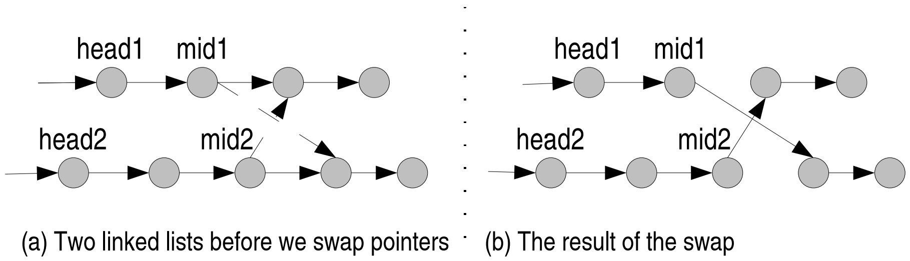


Fig. 1. Swapping pointers between two linked lists

positionally. We observe that, modulo use of reasoning in the theory of equality, this language is a strict superset of the rules used for instantiation triggers in Nelson-Oppen provers. It allows the implementation of many much more complicated and clever instantiation strategies, not to mention proof strategies based on higher-order logic programming, rewriting, and more. We will take advantage of this in our support for two-way interaction between Coq and Kettle.

## 4 A Motivating Example

We begin by illustrating the use of our approach for program verification with a simple example. Consider the simple operation on linked lists demonstrated by Figure 1. We begin with two acyclic and disjoint singly-linked lists, along with a designated element of each list. Then we swap the "next" pointers of these designated elements, producing two new lists.

One property of this operation that we might like to verify is that the two lists remain acyclic after the swap. Indeed, we expect that the above diagram has convinced the reader of this fact instantly. Nonetheless, proving this fact formally can be a painstaking task. It is precisely for "obvious" properties like this that we can hope to gain the most by applying the kind of semi-automated techniques that we propose. This example is particularly interesting here because its most natural solution involves the undecidable theory of transitive closure, for which we can't hope to build a complete ATP tool.

We will now step through the process of proving a representative lemma used in the proof of this theorem. We need to know that any intermediate node of an acyclic list can also be viewed as the head of an acyclic list:

```
Lemma acyclic_reach : forall next head mid,
  acyclic_list next head
  -> (head -- next --> mid)
  -> acyclic_list next mid.
```

Here, `next` is a functional memory value, represented as a map from a list node pointer to its "next" field. The notation `p -- next --> q` indicates that pointer `q` is reachable from pointer `p` by following "next" pointers in memory `next`. `->` is the Coq notation for implication.

We use a simple definition of list acyclicity in terms of the existence of a path to the null pointer, ignoring the case of infinite lists with no repeating elements:

```
Definition acyclic_list next head := head -- next --> null.
```

The informal proof of the lemma is simple. The acyclicity of the list rooted at `head` is witnessed by a path to `null`. We know that `head` also has a path to `mid`, and, by basic properties of these paths, the path to `mid` must be a prefix of the path to `null`. The portion of the longer path that comes after this prefix shows precisely that the list rooted at `mid` is acyclic.

Constructing a formal proof in Coq requires a bit more work. Reachability is defined inductively as the propositional function `trans_closure`, standing for transitive-reflexive closure, for which the arrow syntax shown earlier is a shorthand:

```
Inductive trans_closure (next : memory pointer)
      : pointer -> pointer -> Prop :=
  | TC_Eq : forall p, p -- next --> p
  | TC_Step : forall p q, ((p # next) -- next --> q)
                        -> (p -- next --> q)
where "p1 -- m --> p2" := (trans_closure m p1 p2).
```

The infix operator "#" is used to access fields of objects via pointers.

This allows us to induct easily over reachability derivations, so we'll begin our proof by deciding to proceed by induction on the derivation of the first hypothesis. We indicate this by requesting that a particular *tactic* be used. (We'll indicate tactic inputs by boxing them, as below.)

```
induction 1.
```

This produces two subgoals, for the base and inductive cases of the proof. The base case is:

```
p = null -> p -- next --> mid -> acyclic_list next mid
```

We hope that Kettle, our Nelson-Oppen prover, is able to prove this simple goal:

```
kettle.
```

but, in this case, it isn't. We undo this last tactic use and, after a bit of pondering, we notice that the essential fact here is that only null pointers are reachable from `null`. We separately prove this as a lemma with its own inductive proof:

```
Lemma reach_null : forall next p,
  null -- next --> p
  -> p = null.
```

Now, in our original proof, we can suggest that a particular instantiation of the lemma will be useful[5], and then call our Kettle tactic:

```
use (reach_null next mid); kettle.
```

This time, the proof is completed automatically. That leaves us with a sequent

[5] For clarity, we use the name `use` for Coq's built-in `generalize` tactic.

for the inductive case, which we abbreviate here:

```
 H : p # next -- next --> null
IH : p # next -- next --> mid
     -> acyclic_list next mid
=====================================
 p -- next --> mid
 -> acyclic_list next mid
```

We ask Kettle to prove this goal:

```
kettle.
```

In this case, it can't prove the goal entirely, but it *is* able to deduce certain special conditions under which it *can* finish the proof. It returns to us a set of subgoals that together cover all of the cases it couldn't handle. Here is an abbreviated version of one of these sequents:

```
     H : p -- next --> mid
  GOAL : ~ mid -- next --> null
CASE_1 : p -- next --> null
CASE_2 : ~ p # next -- next --> mid
=================================================
 False
```

The original goal appears negated as a hypothesis, and our new goal is `False`, in the standard Nelson-Oppen style. We've also picked up two additional hypotheses expressing a particular path through a *case analysis tree* that Kettle determined was relevant. The set of `CASE` hypotheses tells us that the case tree considers whether or not `p` reaches `null` and whether or not `p # next` reaches `mid`. One of the cases for which Kettle wasn't able to prove the lemma is the case where the first fact is true and the second is false.

Examining the set of hypotheses, we determine that we need to proceed by cases on the derivation of hypothesis `H`. We prove another lemma that will be helpful here:

```
Lemma trans_cases : forall next p q,
  p -- next --> q
  -> p = q \/ p # next -- next --> q.
```

Returning to the original proof, an instantiation of this lemma does the trick:

```
use (trans_cases next p mid); kettle.
```

Why didn't Kettle deduce the importance of this case analysis on its own? The answer has to do with the black art of choosing narrow enough quantifier instantiation heuristics to avoid infinite rule application sequences that keep triggering themselves. If we performed a case analysis like this for every fact `p -- next --> q` among our hypotheses, then each fact `p # next -- next --> q` added would trigger another application of the same rule, and we would go on forever. There are heuristics for choosing when to apply rules, but it is inevitable that no *terminating* heuristic will be complete, since the theory of transitive closure is undecidable. The primary advantage of our approach is in making it convenient for a human to inject insight into matching choices at points in the proof where automated techniques get stuck.

At this point, another similar subgoal remains. We're able to handle it in a similar manner, finishing the proof.

### *4.1 Scripting Instantiations*

We finish with a fairly short proof script, but it still has some undesirable properties. We had to choose a case analysis to perform manually. Would it be possible to *script* our strategy in choosing case analyses so that those parts of the proof could also be automatic? In this case, our answer is yes. We can use Coq's tactic language as a *language for coding matching strategies*. This language is quite expressive, which is important, since we will be using it to express custom solutions to a problem that is undecidable in general.

In this case, we observe that our proofs, and many others like them that we can imagine, fit a particular heuristic pattern. We start by calling `kettle` to discharge as many cases as possible using only Kettle's automatic heuristics for choosing case analyses. In each remaining case, we use at most one case analysis directly on any single reachability hypothesis. Kettle doesn't attempt such instantiations in general, because they can lead to infinite chains of cascading lemma instantiations. However, we know that Kettle will be able to finish the proof in every remaining case after a single instantiation of this kind, though it may be a different instantiation for each case. In general, by allowing specialized case analysis choices in different cases, we can improve Kettle's proof search efficiency exponentially.

We can script a tactic `bounded_kettle` that follows this strategy with the following code:

```
Ltac trans_then_kettle :=
  match goal with
    | [ H : ?P -- ?NEXT --> ?Q |- _ ] =>
      use (trans_cases NEXT P Q); kettle; fail
  end.
Ltac bounded_kettle := kettle; try trans_then_kettle.
```

`bounded_kettle` first calls normal `kettle` to handle the "easy" cases. After that, it attempts to use another tactic `trans_then_kettle` to solve the remaining cases. The `try` keyword indicates that when `trans_then_kettle` fails to solve one of these goals, that subgoal should be left for the user instead of signaling an overall failure.

The `trans_then_kettle` tactic begins by pattern matching on the current sequent, looking for a hypothesis that makes a reachability statement. If it finds one, it triggers a case analysis on the derivation of that hypothesis (using `trans_cases`) and calls Kettle. If Kettle succeeds in proving the goal completely, then `trans_then_kettle` has succeeded. If any subgoals remain, then we reach the `fail` tactic, which declares the proving effort a failure. The semantics of pattern matching is based on *back-tracking search*, so a failure with one choice of `H` will lead us to move on to the next hypothesis matching the given pattern. If we are unable to prove the goal with any of the hypotheses, then the overall `trans_then_kettle` invocation fails, which causes control to reach the `try` and leave this subgoal for the user.

Using `bounded_kettle`, we are able to prove `acyclic_reach` with a succinct one-line proof script:

```
intros until mid; use (reach_null next mid);
      induction 2; bounded_kettle.
```

## 5 Algorithm Outline

We now give a high level outline of our algorithm, structured as two cooperating subroutines representing Kettle and Coq, shown as Algorithms 1 and 2. This presentation doesn't follow the structure of the real implementation; rather, we chose it to express the main aspects of interaction and automated proving. One notable omission is explicit translations between Kettle and Coq formula and proof formats. We discuss proof encoding issues in the next section.

Here's the main idea: The Kettle subroutine attempts to prove *falsehood* from a particular set of hypothesis formulas. Besides the context $\Gamma$ of these hypotheses, it tracks an *E-graph* $G$. E-graphs [10] are compact structures for representing equality relationships in the theory of equality and uninterpreted functions. Like in Nelson-Oppen provers in general, we use them to avoid redundant instantiations of quantifiers with syntactically different terms that we know to be equal. The Kettle procedure tries to solve a goal through an iterating process of making new quantifier instantiations, trying ground decision procedures that use only those context formulas that are conjunctions of literals, and performing case splits from disjunctive hypotheses. The instantiation procedure is parameterized on a set of rules $R$, which are quantified formulas together with syntactic triggers specifying patterns that must be matched in the E-graph before they are used. When none of these methods works, Kettle returns a *proof hole* to be dealt with by the Coq user instead of failing. Note that Kettle invokes itself recursively, so these holes can appear sprinkled throughout the final proof tree.

The Coq subroutine is the normal Coq interactive proving loop, with the pseudocode for our Kettle tactic included inline. The main thing to note here is that the cooperative interaction between Kettle and Coq comes from the ability for the user to call the kettle tactic interactively while several levels deep in recursive calls to Coq, each of which may have called Kettle before moving to the next level of recursion.

We give simplified algorithms that omit many optimizations from our real implementation, including extensive support in Kettle for an approach based on in-place updates and explicit snapshot and undo. Kettle also uses some smart heuristics for minimizing the number of case splits it performs, based on ideas pioneered in SAT solving [20,3].

We have not explored more subtle interaction models between Kettle and Coq

**Algorithm 1** $\mathsf{Kettle}(\mathsf{context}\ \Gamma, \mathsf{egraph}\ G)$

**if** $(proof \leftarrow \mathsf{groundTheories}(\Gamma)) \neq \mathsf{failure}$ **then**
  **return** $proof$
**else if** there is $(\phi_1 \vee \phi_2) \in \Gamma$ such that $\phi_1 \notin \Gamma$ and $\phi_2 \notin \Gamma$ **then**
  **return** $\mathsf{orElimination}(\phi_1, \phi_2, \mathsf{Kettle}(\Gamma \cup \{\phi_1\}, G), \mathsf{Kettle}(\Gamma \cup \{\phi_2\}, G)$
**else if** $\exists$ matching rule $R$ whose pattern matches $G$, where the tuple $t$ of pattern variable instantiations is not equal to any tuple matched previously for $R$ in the current call stack, modulo the congruence induced by $G$ **then**
  $\Gamma' \leftarrow \Gamma \cup \{$instantiation of $R$ with $t\}$
  $G' \leftarrow$ expansion of $G$ to include new terms present in the instantiation
  **return** $\mathsf{Kettle}(\Gamma', G')$
**else**
  **return** $\mathsf{proofHole}$
**end if**

**Algorithm 2** $\mathsf{Coq}(\mathsf{context}\ \Gamma, \mathsf{goal\ formula}\ \phi)$

$tactic \leftarrow$ result of querying the user
**if** $tactic = \mathsf{kettle}$ **then**
  $\Gamma' \leftarrow \Gamma \cup \{\neg\phi\}$
  $proof \leftarrow \mathsf{Kettle}(\Gamma', \mathsf{buildEgraph}(\Gamma'))$
  **for all** proof holes in $proof$ **do**
    Call Coq recursively on the goal of that proof hole, with $\Gamma$ expanded appropriately to consider hypothetical judgments.
    Substitute the resulting proof for this hole.
  **end for**
  **return** $\mathsf{notElimination}(\phi, proof)$
**else**
  ...standard handling of other Coq tactics...
**end if**

where one sees the other as something besides a black box. This means that, at present, Kettle isn't able to work incrementally and take advantage of similarities across queries from Coq. However, Kettle was designed to be used in just this way, with its snapshot and undo features, so we expect that a little engineering work can remove this inefficiency.

## 6 Producing Proofs

The structure of Nelson-Oppen provers is well-suited to the generation of proof terms. Kettle uses a proof language that witnesses sequences of choices made by the cooperating decision procedures, with the ultimate goal to ensure that proof checking does not involve search and is therefore both faster and simpler than proving. However, Kettle's proofs are not built from basic axioms alone. For example, all linear arithmetic proofs in Kettle are ultimately based on a proof rule that involves linear arithmetic simplification. Because they use such "domain-specific" proof rules, Kettle's proofs are small and easy to generate [17].

Here is a sample of the definition of the OCaml datatype for Kettle proofs:

```
type kettle_proof =
  Hyp of hypName
| Andi of kettle_proof * kettle_proof
| Ore of kettle_proof * (hypName * kettle_proof)
                      * (hypName * kettle_proof)
| Alle of kettle_proof * kettle_exp
| FromDNF of kettle_pred * kettle_proof
| Hole of kettle_pred
| ...
```

The first four cases are standard natural deduction proof rules. For instance, `Ore` is a standard $\vee$-elimination rule that introduces named hypothetical judgments in two of its subproofs. Then we have `FromDNF`, a high-level rule that we will discuss shortly.

Finally, there is the `Hole` proof constructor. In normal operation, Kettle does its best to determine effective case analyses and prove the goal under every choice of cases. In our use of it as a Coq tactic, we want to allow more than just complete success or complete failure; the user must be able to take over in cases that Kettle wasn't able to discharge. The simple extension of Kettle's proof language with the `Hole` constructor allows for this; when our modified Kettle isn't able to prove a goal predicate $p$ in a given case, it generates a proof `Hole`($p$). The Kettle tactic detects these holes and requests that the user prove each hole's predicate interactively.

Here is a sample of the structure of our main function to translate Kettle proofs to Coq proof terms. For clarity, we show it in terms of a simpler ML interface to Coq than is really provided, with more automatic type inference.

```
let rec translate_pf = function
  Hyp s -> Variable s
| Andi (pf1, pf2) -> Apply (andi, [translate_pf pf1;
                                   translate_pf pf2])
| Ore (pf, (h1, pf1), (h2, pf2)) ->
  Apply (ore, [translate_pf pf; Lambda (h1, translate_pf pf1);
                                Lambda (h2, translate_pf pf2)])
| Alle (pf, e) -> Apply (translate_pf pf, [translate_exp e])
| Hole p -> ProveWithSubgoal (translate_pred p)
| FromDNF (p, pf) ->
  Apply (prove_from_dnf, [symbolic_prop p; translate_pf pf])
| ...
```

The reader can see that the standard natural deduction proof rules have simple translations. The most interesting differences have to do with the translation of implication, universal quantification, and functions. These can be handled uniformly with dependent products on the Coq side, while they involve distinct constructs in Kettle. The `ProveWithSubgoal` function that is used for the translation of the `Hole` uses standard Coq mechanisms for recording a subgoal to be proved later. We discuss next the translation of the `FromDNF` Kettle proof constructor.

### *6.1 Normalizing Propositions*

The `FromDNF` case is an interesting one. This is one of Kettle's high level proof rules that triggers the use of an algorithm that doesn't require search. `FromDNF` is used at a point in the proof where Kettle determines that it is helpful to convert the goal to disjunctive normal form before proceeding. The proof rule deduces the original form of the goal from a proof of its disjunctive normal form:

$$\frac{p \xrightarrow{\mathsf{DNF}} p' \quad p'}{p} \; \mathit{fromDNF}$$

As suggested in the above proof rule, checking such a proof in Kettle requires a function for conversion of formulas to their disjunctive normal forms. It is possible to use a straightforward translation of `FromDNF` uses into Coq proof terms: there is always a tedious, unenlightening translation into a tree of applications of basic Coq lemmas about propositional logic. However, these proof trees will contain intermediate terms repeated at multiple levels, leading to significant increases in proof size. We can do better by using a proof idiom called *proof by reflection*.

We start by defining in Coq a new type of abstract syntax trees for propositions, whose partial definition we show below.

```
Inductive symbolic_prop : Type :=
  | P_Prop : Prop -> symbolic_prop
  | P_And : list symbolic_prop -> symbolic_prop
  | P_All : forall (T : Set), (T -> symbolic_prop)
                              -> symbolic_prop
  | ...
```

The first thing to notice about the inductive type `symbolic_prop` is that it is complete for the set of possible Coq propositions, which are represented by the built-in sort `Prop`. Any proposition $P$ is representable trivially as `P_Prop`($P$). The additional structure is necessary to expose possibilities for syntactic rewriting.

Next we define interpretation functions for propositions, along with a translation from arbitrary propositions into DNF and its soundness theorem.

```
Definition interp_prop : symbolic_prop -> Prop := ....
Definition to_dnf : symbolic_prop -> symbolic_prop := ....

Theorem prove_from_dnf : forall (p : symbolic_prop),
  interp_prop (to_dnf p) -> interp_prop p.
```

Since Coq includes a basic functional programming language as a subset of its logic, it is straightforward to implement these functions and prove their soundness. The Coq function `to_dnf` mirrors the functionality of the corresponding function in the OCaml implementation of Kettle. While it would be an onerous task with little benefit to make this OCaml function proof-generating, Coq's dependent type system and interactive proving support make implementing this algorithm with a soundness proof easy.

Now suppose that a given Kettle execution at some point translates $\neg(P \wedge Q)$ into its disjunctive normal form $\neg P \vee \neg Q$ before proceeding. Let $pf$ be a proof of the normalized proposition, translated into a Coq proof term. A proof of the original goal is:

$$\texttt{prove\_from\_dnf}\ (\texttt{P\_Not}\ (\texttt{P\_And}\ (\texttt{P\_Prop}\ P)\ (\texttt{P\_Prop}\ Q)))\ pf$$

Why does Coq accept this proof? The answer is that the Coq proof checker identifies terms up to equivalence via standard lambda calculus rewriting rules, such as beta reduction. A series of beta reductions and other rewritings transforms

$$\texttt{interp\_prop}\ (\texttt{to\_dnf}\ (\texttt{P\_Not}\ (\texttt{P\_And}\ (\texttt{P\_Prop}\ P)\ (\texttt{P\_Prop}\ Q))))$$

into $\neg P \vee \neg Q$, which is exactly what $pf$ proves.

Even though translations of arbitrary formulas to DNF must sometimes increase formula size significantly, we have managed to keep the sizes of our DNF translation proofs relatively small in all cases. That is, we generate small *proofs*, while the DNF translation of *formulas* has the usual size characteristics. Besides the size of the proof $pf$, which we can't avoid, the only other non-constant component of proof size comes from one linear-size restatement of the goal in a different form. In a more traditional proof based on, e.g., a balanced proof tree successively applying simplification lemmas, each atomic formula would be mentioned a logarithmic number of times, which requires an asymptotically longer representation than we provide.

### *6.2 Proofs for Linear Arithmetic*

Another class of interesting high-level rules deals with arithmetic simplifications used by decision procedures for linear arithmetic. For instance, we have the following rule to encode compactly the proof of equality of two arithmetic expressions:

$$\frac{n \neq 0, \quad \bigwedge_i (e_i = e'_i), \quad n(e - e') - \sum_i f_i(e_i - e'_i) \leadsto 0}{e = e'}\ \mathit{aritheq}$$

The arrow $\leadsto$ denotes a syntactic simplification algorithm to be run as part of proof checking. The rule is used to conclude the equality of two integer expressions $e$ and $e'$. It takes as additional inputs a constant factor $n$ and $k$ triples $\langle e_i, e'_i, f_i \rangle$. We show $e = e'$ by showing $ne + \sum_i f_i e_i = ne' + \sum_i f_i e'_i$, given proofs that $e_i = e'_i$ for each $i$.

As in the case of proposition normalization, we can't perform syntactic analysis directly on arbitrary integer-valued expressions in Coq. Instead, we introduce another type of abstract syntax trees:

```
Inductive linear_Z : Set :=
  | LZ_Const : Z -> linear_Z
  | LZ_Var : var -> linear_Z
  | LZ_Plus : linear_Z -> linear_Z -> linear_Z
  | LZ_Minus : linear_Z -> linear_Z -> linear_Z
  | LZ_Mult : Z -> linear_Z -> linear_Z.
```

We can then implement the simplification algorithm in Coq, prove its soundness, and use it in the Coq versions of rules like *aritheq*. Again, the strategy is to implement in Coq both the machinery that Kettle uses to check uses of its high-level proof constructors and the proof of its soundness.

## 7 Implementation and Preliminary Case Studies

We have implemented our Kettle tactic in OCaml. Along with a library version of the Kettle code originally developed for a standalone prover, it is linked into a custom Coq binary. We have used our tactic in a few case studies to validate the utility of our approach. None of these studies is large enough to provide hard data on how much more effective a user of our tool can be than he would be without it, but they served to test the robustness of our implementation and allow us to present some preliminary figures.

The largest study so far involved constructing a complete proof of the main theorem presented for our motivating example in Section 4. Compared to an earlier manual attempt, we used our new tactic to reduce the number of proof script lines for problem-specific theorems from 37 to 16. The resulting scripts are also less brittle; they involve fewer specific references to the structure of a goal, so that it's easier to re-use them after small changes to the problem statement.

We also tried using our tactic with a number of the tutorial examples for the Caduceus [11] verification condition generator for C programs. The results were promising; we were able to prove most of the obligations with single Kettle invocations, and prove almost all through at most an explicit induction and instantiation of the inductive hypothesis. We hope to explore this source of examples further so as to be able to present more concrete results.

## 8 Conclusion

We have described an implementation of a tactic for the Coq proof assistant that uses a Nelson-Oppen theorem prover. Our tactic produces compact proofs in Coq's higher-order logic. Instead of serving as a straight decision procedure, it will automatically detect opportunities for case analysis, prove some cases, and return the rest to the Coq user as subgoals. The user can then use Coq's rich tactic language to program customized strategies for proceeding, where these strategies could be too expensive for the automated prover to use. The synergy we enable between these two types of proving tools allows many theorems to be proven more efficiently and succinctly than with either approach alone.

**Acknowledgements**

Thanks to Bor-Yuh Evan Chang and the anonymous referees for helpful feedback on earlier versions of this paper.

## References

[1] Andreas Abel, Thierry Coquand, and Ulf Norell. Connecting a logical framework to a first-order logic prover. In *FroCos'05: Proceedings of the 5th International Workshop on Frontiers of Combining Systems*, pages 285–301, 2005.

[2] Nicolas Ayache and Jean-Christophe Fillitre. Combining the Coq proof assistant with first-order decision procedures, March 2006. Unpublished manuscript.

[3] Clark W. Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker category B. In *CAV'04: Proceedings of the 16th International Conference on Computer Aided Verification*, pages 515–518, 2004.

[4] Marc Bezem, Dimitri Hendriks, and Hans de Nivelle. Automated proof construction in type theory using resolution. *J. Automated Reasoning*, 29(3):253–275, 2002.

[5] Richard J. Boulton. Combining decision procedures in the HOL system. In *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, pages 75–89, London, UK, 1995. Springer-Verlag.

[6] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In *STACS'97: Proceedings of the 14th Annual Symposium on Theoretical Aspects of Computer Science*, pages 515–529, 1997.

[7] Evelyne Contejean and Pierre Corbineau. Reflecting proofs in first-order logic with equality. In *CADE-20: Proceedings of the 20th International Conference on Automated Deduction*, pages 7–22, 2005.

[8] Coq Development Team. The Coq proof assistant reference manual, version 8.0. 2006.

[9] D. Detlefs, G. Nelson, and J. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, Palo Alto, CA, 2003.

[10] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, 130 Lytton Ave., Palo Alto, December 1998.

[11] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In *ICFEM'04: Proceedings of the 6th International Conference on Formal Engineering Methods*, pages 15–29, 2004.

[12] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI'02: ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245, New York, NY, USA, 2002. ACM Press.

[13] P. Fontaine, J.-Y. Marion, S. Merz, L. Prensa Nieto, and A. Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In *TACAS'06: Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2006.

[14] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 58–70, New York, NY, USA, 2002. ACM Press.

[15] S. McLaughlin, C. Barrett, and Y. Ge. Cooperating theorem provers: A case study combining CVC Lite and HOL Light. In *PDPAR'05: Proceedings of the Third Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, 2005.

[16] Jia Meng and Lawrence C. Paulson. Experiments on supporting interactive proof using resolution. In *IJCAR'04: Proceedings of the 2nd International Joint Conference on Automated Reasoning*, pages 372–384, 2004.

[17] G. C. Necula and P. Lee. Efficient representation and validation of proofs. In *LICS '98: Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science*, page 93, Washington, DC, USA, 1998. IEEE Computer Society.

[18] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[19] Quang Huy Nguyen, Claude Kirchner, and Hélène Kirchner. External rewriting for skeptical proof assistants. *J. Automated Reasoning*, 29(3-4):309–336, 2002.

[20] Robert Nieuwenhuis and Albert Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. In *CAV'05: Proceedings of the 17th International Conference on Computer Aided Verification*, pages 321–334, 2005.

[21] Tobias Nipkow, Lawrence C. Paulson, and M Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, London, UK, 2002.

[22] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *CADE-11: Proceedings of the 11th International Conference on Automated Deduction*, pages 748–752, London, UK, 1992. Springer-Verlag.

[23] Aaron Stump and David L. Dill. Generating proofs from a decision procedure. In A. Pnueli and P. Traverso, editors, *RTRV'99: Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy, 1999.