Anatomy of a Platform for Prover Experimentation

Hampton Smith

Abstract—Most 3rd party provers stress performance over flexibility and modifiability as a non-functional attributes, making them ill-suited for prover experimentation. Taking one of the few more flexible systems, Coq, as a template, we explore how we may design a more modular prover that integrates directly with the current RESOLVE system.

1 Rationale

The majority of 3rd party provers available today hard code many algorithmic decisions. This is understandable since most provers are coded for efficiency. However, from a research perspective, this introduces two negative non-functional attributes: 1) it encourages a "black-box" architecture: hard-coded algorithmic decisions can be taken for granted and incompletely explained in documentation and 2) it discourages experimentation since the algorithm cannot be modified or, for open source provers, a thorough understanding of the code base is required to safely modify the proving algorithm. By contrast, some proof assistants like Coq combat these issues by providing sophisticated languages for defining custom proving tactics in a safe way. These flexible system are, however, few and far between and suffer from the usual problems of integrating a third-party solution with RESOLVE (binary compatibility, mathematical compatibility, etc.) In addition, they lack other features desirable in the context of a fully automated system. For example, since Coq assumes a human being is guiding the proof and only a "yes or no" answer is required, it is difficult to reconstruct the specific proof-path taken to arrive at an accepting state.

In this paper we briefly discuss our design for a flexible, RESOLVE-oriented proving solution intended to act as a platform for experimentation with both the VC generation and proving processes.

2 Design Goals

- Complete, robust data structures that encourage clear, safe manipulation of VCs.
- Plug-in architecture that permits new proof steps to be added with minimum modification to existing code.
- Composable proof strategies.
- End-user scriptable proof strategies.
- Reconstructable, step-by-step proofs that can be mechanically checked.
- Minimal assumptions on the part of the prover about structure and content of available theories.
- Minimal assumptions on the part of the user about the nature of the prover.

3 Architecture

This section will look at the architecture of four key aspects of our prover's design: the main proving loop, which endeavors to factor out common proof needs; the VC data structure, which is the core data structure on which the prover operates; the VCTransformer interface, which represents a basic atomic proof step on a VC; and the TransformationChooser interface, which acts as the brain of the prover.

3.1 Main Prover Loop

The main prover loop endeavors to be as simple as possible while still encapsulating a number of error-prone tasks such as alternative iteration, backtracking, and the assembling of proof reports. In pseudocode, the logic of this loop is as follows:

```
Prove(VC, TransformationChooser):

    if VC is trivially true:
        return TriviallyTrue
    Strategies ←
        TransformationChooser
            .suggestStrategies(VC)

    for Strategy in Strategies:
        NextStepVCs ← Strategy.apply(VC)

        for NextStepVC in NextStepVCs:
            proved ← Prove(
                NextStepVC,
                StrategyChooser)

            if proved is not Unprovable:
                return (
                    (Strategy, NextStepVC),
                     proved)

    return Unprovable
```

The recursive nature of the proving loop means that any step that fails gets backtracking for free.

3.2 The VC Data Structure

VCs in the prover are immutable collections of conjuncts divided into consequent and antecedent. Type safety is provided to ensure that antecedent conjuncts (which are assumed) and consequent conjuncts (which must be demonstrated) cannot be confused, with the only way to convert from a consequent to an antecedent via a call to the `assumed()` method (ideally, a proof object would be required, but the current design at least requires the programmer to explicitly acknowledge that she is making the conversion.) Basic methods for obtaining modified versions of the VC (such as by removing or adding a conjunct) are provided, but for flexibility more sophisticated transformations are not built-in but rather achieved by passing a VC to a VCTransformer object, discussed next.

3.3 The VCTransformer Interface

VCTransformers define a consistent, abstract way of providing alternative applications of a rule to a VC. Each VCTransformer represents a single proof step, such as an application of the Identity Property of Zero, replacing `x + 0` with `x`. A VCTransformer takes as its argument a VC and returns an iterator over alternative applications of that VC. For example, if the conjunct `(x + 0 = y + 0)` were to appear in a VC and the Identity Property VCTransformer were applied, with would iterate over two different VCs. One containing `(x = y + 0)` and the other, `(x + 0 = y)`. VCTransformers exist for developing theory by expanding the antecedent, performing substitutions in place, strengthening conjuncts, binding existentials, and others. Ideally there would be a one-to-one correspondence between VCTransformers and steps available in RESOLVE's proof language.

Because VCs are immutable and all proof steps occur via a VCTransformer, this interface provides the basic encapsulation of a sound mathematical operation. Assuming each VCTransformer subclass can be demonstrated sound, no outside code has the potential for introducing unsoundness.

3.4 The TransformationChooser Interface

The TransformationChooser interface represents the basic building block of the prover's proof strategy plugin-in architecture. TransformationChoosers are responsible for prioritizing and suggesting what steps should be tried next given a proof state. This is achieved functionally—TransformationChoosers are agnostic of the general proving process. They are, however, provided with a whiteboard so that multi-step strategies are possible. This whiteboard is affected by backtracking that is taken care of by the main proof loop (thought of another way: the whiteboard is a component of the VC and likewise immutable).

In order to aid composition, a library of high-level TransformationChooser combinators are available, themselves TransformationChoosers. For example, combinators exist to chain together the suggestions of two TransformationChoosers at each step, or to continuously apply suggestions from one TransformationChooser until it cannot suggest any more, then fail over to another. Functionality that might ordinarily be thought of as a part of the proof loop such as limiting proof length or simplifying at each step is also implemented using TransformationChoosers, leaving these behaviors entirely configurable. The interface ensures that a TransformationChooser is entirely agnostic of any parent TransformationChoosers.

4 Status and Future Work

The entire architecture has been implemented and is working smoothly. At this time, proof reports are produced in a human-readable format rather than a machine-checkable one. While each VCTransformer should represent an atomic step, this yields very lengthy proofs when, e.g., extensive blind antecedent development is performed as a pre-processing step, and thus VCTransformers exist to collapse these steps into a single one in the proof output. Ideally a mechanism should be developed to post-process proofs and collapse irrelevant conjuncts and subpaths.

No mechanism currently exists to allow a TransformationChooser to be assembled on-the-fly at run-time. However, the proof process is entirely customizable by modifying only a single method to produce the desired TransformationChooser and pass it along with the VC to the prover. In light of this, we may choose to qualify scripted proving as a low-priority goal since we expect that experimentation is being performed by members inside the group for the moment.

With the increased flexibility of this design, an exploding proof space has become a serious concern. The prover manages to make serious efficiency gains using immutable data structures, as reference copying may be employed freely. However, holes exist in the interface and mutable data structures are still an integral part of its functioning (the conjuncts themselves, for example, are still represented using the general prover's mutable abstract syntax tree and thus care must be observed to copy them if they are to escape a VC). Plugging these holes has become a high priority in order to achieve high efficiency. In addition, sanity checks for analyzing TransformationChoosers should be implemented to assist experimenters in flagging unnecessary state explosion.

5 Preliminary Results

The flexibility of this design has enabled some preliminary experimentation with a number of aspects of the proving process. Experience with a few of these experimentations is provided here.

As the complexities of components grow, it is easy to end up with VCs that require many steps to prove with straightforward theorems. For example, this VC from code to flip a Cursor_List specified as a String with an Integer pointer to the cursor position:

```
|S.List| - 0 > 0
=======================>
((|(Left_Substring(S.List, 0) o
Right_Substring(S.List, (0 + 1)))| - 0) <
(|S.List| - 0))
```

This challenges our hypothesis that VCs arising from modular code in a clean language like RESOLVE should be straightforward to prove: the proof space including proofs of length seven (such as would be required here) is very large. However, a human mathematician would consider many of the required steps obvious. Labeling certain theorems (such as the Identity of Zero, here) as "obvious" would violate our design goal of a prover-agnostic user—his job is to state theorems, not qualify them. However, by creating a fitness function to qualify theorems that strictly simplify a VC (i.e., result in strictly fewer function applications) and feeding those theorems to a TransformationChooser that ignores length restrictions and applies steps until it can apply no more, we can configure the prover with the heuristic to simplify as much as possible before beginning the proof-search-proper. We have had preliminary success with this approach in bringing many VCs from different programs into the range of simple provability.

Another heuristic we have experimented with arises from theory development. Since theory development may be cumulative (having a list of many true things does not burden us), it can be done in large batches. However, a human mathematician would recognize that given an assumption like:

```
|S| = 0
```

Expanding this assumption to include:

```
|S| = 0 and
S = empty_string
```

Would be more helpful than to expand it with:

```
|S| = 0 and
|S| + 0 = 0 and
|S| + 0 = 0 + 0 + 0
```

In an attempt to mechanize this intuition, we have implemented a VCTransformation that eliminates equivalent antecedent conjuncts that are stated in the same terms. That is, while it will still develop `|S| = 0` into `|S| + 0 = 0`, it will not then admit the further fact that `|S| + 0 = 0 + 0` since this fact introduces no new functions or literals. The usefulness of this heuristic in curbing antecedent explosion is still under active investigation.

Acknowledgements

This research is funded in part by NSF grants CCF-0811748 and DMS-0701187.