

# TOWARD A SOUND INTEGRATION OF ISABELLE WITH A COMBINED DECISION PROCEDURE

TOM HARKE

ABSTRACT. I present work on a project to integrate Isabelle, an extremely versatile interactive proof assistant, with a combined decision procedure, the Cooperating Validity Checker (CVC). Isabelle is sound and flexible, however it is often tedious to use. CVC is fully automatic, but only handles decision problems expressible over a relatively weak set of theories including linear arithmetic, uninterpreted functions, data types, and first-order quantifier-free logic. My goal is to increase the amount of automation in Isabelle, by making it use CVC as an oracle for such problems, but without compromising Isabelle's soundness.

In this paper I report on the progress made toward this goal. The key to retaining soundness is in CVC's ability to produce proofs. I have implemented a basic infrastructure to translate these proofs into Isabelle tactics. This effort has revealed a number of issues that complicate the translation.

One complication is an unwanted conversion from a logic of partial functions to classical logic. Another is the unsoundness of the CVC proof system, which makes part of a CVC proof unusable. Fortunately, we can regenerate the unusable part by mimicking CVC's conversion.

The current state of the project includes a detailed plan to work around the complications, and a partial implementation that handles part of first-order quantifier-free logic.

## 1. INTRODUCTION

1.1. **Proof Tools.** Computer scientists need automated formal proof tools. Failure of hardware and software systems can be costly, in loss of both money and life. But such systems are becoming increasingly complex. An effective way of ruling out an entire class of bugs is to *prove* that a system satisfies a specific properties.

For generations mathematicians have managed without proof tools. Some reasons why computer scientists need them are: the details are more tedious; our artifacts are orders of magnitude larger; and the cost of failure can be great. Nelson & Oppen [NO79] suggest that: "Mechanical reasoning [be] used in program manipulation to verify routine facts or to catch slippery errors, not to prove mathematically interesting theorems." Recently some mathematicians have started to use proof tools. Tom Hales claims to have a proof of Kepler's sphere packing conjecture, however the team of mathematicians reviewing it have given up

due to its length and detail [Szp03]. Hales started the Flyspeck project [Hal] to machine check some of the details.

**1.2. A Range of Proof Tools.** A wide range of proof tools aid in the creation and verification of mathematical proofs. We now look at two extremes.

*1.2.1. Decision Procedures.* A **decision procedure** is an algorithm that solves a **decision problem** from within a **theory**. A theory is a set of functions (e.g.  $\{+, -, =\}$ ) together with axioms (e.g.  $\{\forall x, y \in \mathbb{Q}. x + y = y + x, \dots\}$ ). A decision problem is a class of related questions whose answers are “yes” or “no”.

For instance, consider the theory of linear equations over  $\mathbb{Q}$ , which has variables and allows equality ( $=$ ), addition ( $+$ ), and scalar multiplication. A decision problem from this theory is: Given a set of equations,  $\{a_{i1}x_1 + a_{i2}x_2 + \dots + a_{in}x_n = c_i \mid 1 \leq i \leq m\}$ , is there an assignment of the variables to values in  $\mathbb{Q}$  that satisfies all the equations? The decision procedure is essentially Gaussian elimination: perform elimination; if there is a solution, return “yes”; otherwise “no”.

Basic decision procedures are fully automatic, yet applicable to rather limited theories. Later, in §2.1, we review the more powerful cooperating decision procedures.

*1.2.2. Interactive Proof Assistants.* **General purpose interactive proof assistants** such as Isabelle [Pau89] are at the other extreme, having complementary strengths and weaknesses to decision procedures. These assistants are proof verifiers that require the user to provide a proof, but they are interactive because they allow part to be verified before the whole is complete. Their interactivity also allows them to be used to explore possible proofs. Their extreme versatility can model a range of mathematical and computational knowledge: e.g. hardware verification, correctness of algorithms, and properties of network protocols. They provide some automation, and more is possible if the user does some programming. They typically need much human guidance: Often a step of the proof must be broken down into smaller sub-steps in order for the tool to accept it.

**1.3. High-Assurance Architecture and Tools.**

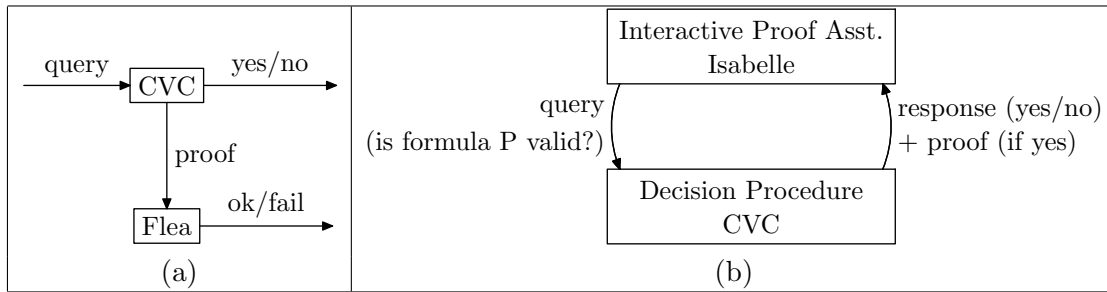


FIGURE 1. High-Assurance Uses of CVC

1.3.1. *The Idea of High-Assurance.* An obvious question about a proof tool is whether we trust it. A naïve answer is that when the tool is widely used then, over time, all significant bugs will turn up and be fixed. This answer is not acceptable if the debugging process involves lost Mars rovers and molten nuclear reactors. Moreover, the implementors of some tools recognize their instability. Stump and Dill [SD99], in regards to the Stanford Validity Checker (SVC), comment that “because decision procedures are a subject of active research, they are often experimental rather than mature and trusted tools.”

A better answer is to verify the proof tool. A **sound** proof tool will not infer an invalid conclusion from valid assumptions. I will use the term **high-assurance** to describe proof tools for which:

- there is a convincing (informal) argument for the tool’s soundness, and
- the details of the argument are simple enough for a human to understand.

The Cooperating Validity Checker (CVC) [SBD02] is a state-of-the-art combined decision procedure. CVC alone is a *non-example* of a high-assurance architecture. It has 150,000 lines of C++ code and it is possible that an error somewhere in that code causes unsoundness.

To mitigate this potential problem, the designers made CVC part of a set of tools: CVC can generate proofs of true results, for independent verification by the program Flea as shown in Figure 1 (a). The code for Flea is small, since checking proofs is much simpler than discovering them, and this code may be audited. The logic for Flea, though a bit large, may also be audited. This combination of Flea with CVC has a high-assurance architecture.

Though the *architecture* of this combination is high-assurance, the *implementation* is not. I have found number of unsound rules in the logic it uses (see appendix 8.3). While it is doubtful that the unsoundnesses are exploited, verification now requires examining the source to CVC, defeating the aim of the architecture.

Another example is Isabelle, whose architecture is high-assurance (we’ll see it in more detail in §2.2). Briefly, the reason for its high-assurance is that only a small kernel of code directly manipulates theorems.

**1.4. Research Goal.** My goal is to integrate Isabelle, a high-assurance proof assistant, with a state-of-the-art combined decision procedure, while maintaining Isabelle’s level of assurance. My motivation is to reduce the amount of uninteresting user interaction required in Isabelle. Current combined decision procedures are not powerful enough to solve many interesting problems, but are fully automated on their domain. Isabelle is versatile, but not automated enough. I hope to have the best of both tools.

Isabelle proofs can be tedious. For instance there is no tool in Isabelle that in one step proves

$$\forall x \in \mathbb{Z}. (x + 1 = 0) \rightarrow f(x + 2) = f(1) \tag{1}$$

instead one must first prove a lemma, or tweak one of the existing tools. In a long proof too many diversions like this clutter the exposition, and the experienced user would rather justify this fact by appealing to “basic mathematics”.

To maintain high-assurance the decision procedure must return a proof along with each positive answer. CVC is currently the only decision procedure with this capability. The proofs that it produces for Flea may also be used to drive an Isabelle proof, as shown in Figure 1 (b).

**1.5. Paper Outline.** The remainder of this paper is organized as follows: §2 discusses background necessary to understand this work; §3 outlines my research plan and progress; Results and challenges are in §4; Then §5 and §6 discuss related and future work, respectively; §7 has the conclusion.

For the sake of readability, I use mathematical notation in place of the various tools' notations. For questions about notation, refer to appendix 8.1. I also simplify examples from CVC when doing so makes no essential changes.

This paper later discusses four specific translations. When we use a CVC proof to construct an Isabelle tactic we will say that we **transpose** the proof. We will reserve this term, as well as **desugar**, **convert** and **add retract** for specific translations. The general term **translation** is still used for *other* translations.

## 2. BACKGROUND

**2.1. Combined Decision Procedures.** Basic decision procedures are only applicable to rather limited theories. Some example theories with decision procedures are: the theory of linear equations; the theory of uninterpreted functions (where  $x = y$  implies  $f(x) = f(y)$  for any function  $f$ ). But consider the system of equations:

$$f(1) = 0 \quad \wedge \quad f(x) = 1 \quad \wedge \quad x + 1 = 2 \quad (2)$$

This system is unsatisfiable: it is easy to see that  $x = 1$  and hence the contradiction  $1 = f(x) = f(1) = 0$ .

Suppose we had two decision procedures, one for each theory. Neither procedure alone could determine the unsatisfiability of (2): The first would not know that  $x = 1$  implies  $f(x) = f(1)$ , while the second would not recognize that  $x + 1 = 2$  is equivalent to  $x = 1$ . The detection of this unsatisfiability requires a more powerful procedure.

It is possible to combine some decision procedures into more powerful tools, while maintaining automation. Nelson and Oppen proposed a framework which allows separate decision procedures for linear arithmetic, list functions (**head**, **tail**, **cons**, **nil**), uninterpreted functions, and others to cooperate [NO79]. The combined decision procedure is more powerful than the sum of its parts. Thus their decision procedure can determine the unsatisfiability of (2) as well as that of:

$$x \leq y \quad \wedge \quad y \leq x + \mathbf{head}(\mathbf{cons}(0, z)) \quad \wedge \quad P(h(x) - h(y)) \quad \wedge \quad \neg P(0) \quad (3)$$

Recently, the most advanced combined decision procedures have been developed by two groups. First, a group at Stanford led by David Dill, which has produced SVC, its successor CVC, and are currently working on CVC-Lite. Second, a group at Stanford Research Institute (SRI), whose most recent combined decision procedure is the Integrated Canonizer and Solver (ICS)[For03].

## 2.2. Fully Expansive Theorem Provers.

2.2.1. *A High-Assurance Architecture.* The Stanford LCF (Logic for Computable Functions) and Edinburgh LCF systems introduced fully-expansive theorem proving [Gor00]. We are not concerned with the specific logic of these systems, but instead with their sound architecture, which the proof assistant Isabelle adopts.

In the fully-expansive approach there are few opportunities for soundness errors. There is a small kernel of basic proof rules. No theorem can be introduced, except by calling a function in the kernel: One example kernel call in Isabelle is `trivial`, which takes a term,  $P$ , and constructs the theorem that assuming  $P$  we may conclude  $P$  (i.e.  $P \vdash P$ ); Another is to apply (higher-order) resolution to two existing theorems; Few other rules are allowed. The type of a theorem is abstract, and the implementation language has a sound typing system, so this interface can not be bypassed. Thus, the only place soundness errors can occur is in the kernel.

Restricting the possible unsoundnesses reduces the work needed to verify Isabelle’s implementation. The set of kernel calls is a small: A logician can verify their soundness with pencil and paper; A programmer can verify that the implementation matches the paper specification. No other component of Isabelle needs checking as there is a mechanism to guarantee that all accepted proofs are composed only of these tactics<sup>1</sup>.

Of course, an error can occur anywhere else in Isabelle’s implementation. With this architecture such an error affects completeness: a valid proof will be rejected due to the wrong kernel calls being made. Such an error will never silently allow an unsound proof to be accepted. At worst, it will frustrate the user.

---

<sup>1</sup>One still has to trust the compiler and the type system of the implementation language. But they are widely used for other applications, and hopefully extensive use has thoroughly debugged them. This argument for soundness is not airtight, but it leaks in significantly fewer places than the arguments most systems offer.

<pre style="margin: 0;"> isabelle code 1 Goal "A&amp;B--&gt;B&amp;A"; (* uses tactic *) 2 by (resolve_tac [impI] 1); 3 by (resolve_tac [conjI] 1); 4 by (resolve_tac [conjunct2] 1); 5 by (assume_tac 1); 6 by (resolve_tac [conjunct1] 1); 7 by (assume_tac 1); 8 qed "conjunction_is_commutative"; </pre>	<pre style="margin: 0;"> isabelle code Goal "A&amp;B--&gt;B&amp;A"; by (REPEAT (resolve_tac              [impI,conjI] 1              ORELSE (resolve_tac                      [conjunct1,conjunct2] 1                      THEN assume_tac 1                      ))); qed "conjunction_is_commutative"; </pre>
(a)	(b)

FIGURE 2. Two Isabelle Proofs

Modern examples of such fully-expansive theorem provers are Isabelle [Pau03b] and HOL (a proof assistant for higher order logic) [Gor00]. In the rest of this paper we will shift from calling Isabelle “sound” to “high-assurance” to acknowledge that there is still an opportunity for unsoundness if the human auditor is careless.

*2.2.2. Creating Proofs.* Mathematicians present proofs in a forward manner, starting with assumptions, working toward a conclusion. Proof discovery usually happens the other way around: starting with a conclusion or goal, attempting to step backward and seeing what subgoals must be attained. The Isabelle **tactic** implements this backward proof step. An example is conjunction introduction, which, when applied to the goal  $A \wedge B$  generates the two subgoals  $A$  and  $B$ <sup>2</sup>.

An example proof in Isabelle that  $A \wedge B \rightarrow B \wedge A$  is shown in Figure 2 (a). Figure 3 shows some steps of the “growth” of the corresponding proof tree. The numbers in the two figures correspond.

A proof consists of a sequence of tactics and references to previously proven theorems. Every proof can be expanded to a proof-tree whose nodes (horizontal lines) are kernel calls, and whose edges are subgoals. Hence the name **fully expansive**. Such a tree is large, due to the inference rules being primitive.

One optimization is to only expand the proof of a lemma once, when it is introduced. For example, `conjunct1` is actually a previously proved lemma, but the steps of its derivation

<sup>2</sup>the name reflects its use in a forward proof

$\frac{\vdots}{A \wedge B \rightarrow B \wedge A}$ <p>(1)</p>	$\frac{\frac{\vdots}{B \wedge A}}{A \wedge B \rightarrow B \wedge A}$ <p>[by impI]</p> <p>(2)</p>	$\frac{\frac{\frac{\vdots}{B} \quad \frac{\vdots}{A}}{B \wedge A} \text{ [by conjI]}}{A \wedge B \rightarrow B \wedge A}$ <p>[by impI]</p> <p>(3)</p>
$\frac{\frac{\frac{\vdots}{A \wedge B}}{B} \text{ [by conjunct2]} \quad \frac{\vdots}{A}}{B \wedge A} \text{ [by conjI]}$ <p>[by impI]</p> <p>(4)</p>	$\frac{\frac{[A \wedge B]}{A \wedge B} \text{ [by assumption]} \quad \frac{[A \wedge B]}{A \wedge B} \text{ [by assumption]}}{\frac{B \wedge A}{A} \text{ [by conjunct1]}} \text{ [by conjI]}$ <p>[by impI]</p> <p>(7)</p>	

FIGURE 3. Growth of a Proof Tree

are not redone when it is used<sup>3</sup>. Another optimization retains only the fringe of the growing tree: In Figure 3 the fringe comprises the parts immediately below the ellipses. With these optimizations the whole tree is forgotten once the proof is done. In both optimizations Isabelle remembers only the fact that verification succeeded, and not the details. This is justified since the details never change.

Working only with such primitive steps is tedious. One remedy is to use lemmas. Another is to use **tacticals**. Tacticals combine existing tactics into new, more powerful tactics. Examples are: **THEN**, which sequences two tactics; **ORELSE**, which tries one tactic first, and then if it fails, tries the second; and **REPEAT**, which repeats a tactic until it fails. The two parts of Figure 2 prove the same theorem. The style in (a) applies exactly the right rules to establish the goal but it becomes tedious for more complex problems. The style in (b) is small program that searches for a proof, which is not as efficient, but is more concise and may be reusable on other similar subgoals.

2.2.3. *Current Automation in Isabelle.* Proofs in Isabelle are often tedious, despite the availability of tactics and tacticals. We need more automation than these simple tacticals provide. Isabelle has three quite powerful tactics.

One is the full Presburger arithmetic solver [CN03]. Unlike the linear arithmetic presented in §1.2.1, full Presburger arithmetic allows arbitrary first-order predicate logical formulas,

<sup>3</sup>So in this case the node of the tree actually corresponds to a fat horizontal line hiding the details of a proof tree.

including quantifiers, built upon linear inequalities. This solver is effectively a decision procedure combining two theories over  $\mathbb{Z}$ : first-order logic and linear arithmetic.

The other two are the **simplifier** and the **classical reasoner** [Pau03b]. The simplifier performs rewriting, both unconditional such as  $x + 0 \mapsto x$ , and conditional such as  $|x| \mapsto -x$  when  $x < 0$ . The simplifier is customizable, allowing users to add and delete rules, and to replace three of its subtactics with user defined ones. The classical reasoner automates proof search by working with a sequent calculus. The classical reasoner is also customizable. For instance, the user may specify: the heuristics for the search; whether to call the simplifier; additional sequents.

None of Isabelle’s powerful tactics can directly handle the goal:

$$(x + 1 = 0) \quad \rightarrow \quad f(x + 2) = f(1) \tag{4}$$

The Presburger arithmetic tactic does not handle uninterpreted functions. The simplifier will solve (4) if we first prove  $x + 1 = 0 \rightarrow x + 2 = 1$  and use it as a conditional rewrite rule. But this is too much work to do for every such goal. The classical reasoner does not handle arithmetic.

**2.3. The Edinburgh Logical Framework (LF).** Now I discuss the main background necessary to understand the proofs generated by CVC.

The proofs that CVC generates for Flea are expressed in a variant of the Edinburgh Logical Framework (LF) [HHP93]. LF uses a parallel correspondence between theorems and types, and between proofs of a theorem to terms inhabiting a type. This correspondence converts the problem of proof-checking to one of type-checking: A proof establishes a theorem exactly when the corresponding term has the type corresponding to the theorem.

In order to be powerful enough to model a variety of logics, LF requires **dependent types**, where the *type* of one term may depend upon the *value* of another term. The reader is likely to know of types such as integers, as well as type families like list which are parameterized by other types, e.g. lists of integer and lists of real. With dependent typing, a type may be parameterized by value, so, for instance, the type of integers mod 7, and the type of lists of length 5 are expressible.

introduction	elimination		example usage
$\frac{P \quad Q}{P \wedge Q} \text{[CI]}$	$\frac{P \wedge Q}{P} \text{[CE1]}$	$\frac{P \wedge Q}{Q} \text{[CE2]}$	$\frac{\frac{A \wedge B}{B} \text{[CE2]} \quad \frac{A \wedge B}{A} \text{[CE1]}}{B \wedge A} \text{[CI]}$

FIGURE 4. Conjunction in Natural Deduction

```

----- Flea code -----
1 conj : prop -> prop -> prop.
2 CI  : {P:prop} {Q:prop} (proof P) -> (proof Q) -> (proof (conj P Q)).
3 CE1 : {P:prop} {Q:prop} (proof (conj P Q)) -> (proof P).
4 CE2 : {P:prop} {Q:prop} (proof (conj P Q)) -> (proof Q).
5
6 example =
7   [A:prop] [B:prop] [p:proof (conj A B)] CI B A (CE2 A B p) (CE1 A B p)
8   : {A:prop} {B:prop} (proof (conj A B)) -> (proof (conj B A)).

```

FIGURE 5. Conjunction in LF

Despite the use of dependent types, type-checking in LF is still decidable. Thus, proof-checking is automatic. Flea adds features such as primitive arrays which make its system potentially undecidable, but Stump makes an informal argument that decidability still holds [Stu02, chapter 4].

Figures 4 and 5 show a comparison of conjunction described in natural deduction [HR00, chapter 1] and in LF, respectively. For the sake of readability I will express LF proofs in natural deduction notation whenever possible.

Figure 5 requires some explanation. LF uses colon (`:`) to assert that the term (on the left-hand side) has the type (on the right). The right-associative arrow infix (`->`) constructs function types. Thus line 1 declares the existence of a primitive (curried) function `conj` taking a pair of propositions to a new proposition. Braces (`{, }`) are used for defining dependent types. Line 2 declares the existence of the function `CI` taking four arguments: the first and second of type `prop`, the third of type `proof P`, the fourth of type `proof Q`. Function `CI` returns a value of type `proof (conj P Q)`. Note that the types of the last two arguments (and the return value) are dependent upon the values of the first two, and thus we need names for those values. In a similar way `CE1` and `CE2` each take two propositions and a proof and return a proof.

To express a theorem we provide the corresponding type *and* a term expressing the proof. This term is composed of terms previously declared or constructed. For instance, line 7 is the proof of the theorem in line 8. Brackets (`[, ]`) introduce parameter names and types. The binding of the name `example`, using `=`, allows later reference to this term.

LF is a general purpose tool for encoding logics. Flea is also general purpose, based on LF, but with nonstandard extensions such as primitive support for arrays and integers. We need only deal with one specific logic, designed by the authors of CVC and Flea. I will refer to this logic as `pfsys`<sup>4</sup>. This first-order logic comprises the signatures and rules of: equality, conjunction, disjunction, if-then-else, uninterpreted functions, algebraic data types (such as enumerations, and binary trees), arrays, tuples and records. `pfsys` attempts to capture two different logics: a logic of partial functions which allows undefined expressions, and the familiar predicate classical logic. It also models the conversion of formulas in the logic of partial functions to formulas in classical logic, which is the next topic of discussion.

**2.4. Type Correctness Conditions.** **Type correctness conditions** (TCCs) are common in the proofs generated by CVC, so knowledge of them is necessary to understand these proofs. The framework that CVC uses to combine decision procedures is based on a classical logic where all functions are total. Yet some theories such as arithmetic and algebraic data types require partial functions. For instance, division is useful, despite being undefined when the denominator is zero; likewise `tail` in the theory of lists is useful though not defined on `nil`. CVC uses TCCs to model formulas containing partial functions with formulas containing only total functions. A TCC for the function  $f$  is a logical expression which is true exactly on the domain of  $f$ . For instance, the TCC for  $f(x, y) = \frac{x}{y}$  is  $y \neq 0$ .

Stump [Stu02, chapter 4] shows how to model formulas from this logic of partial functions in the more familiar classical logic, giving a conversion process  $(\_)^*$  and a proof that

$$\Gamma \models_p \phi \text{ if and only if } \Gamma^* \models_c \phi^* \quad (\text{TCC-equiv})$$

where the subscript on  $\models$  indicates the logic used (partial function, or classical),  $\Gamma$  is a set of axioms, and  $\Gamma^*$  is the set of results of applying this conversion to each element of  $\Gamma$ .

---

<sup>4</sup>so named because it appears in the file `pfsys.flea`

Stump defines the conversion recursively on the structure of the formula. First we need a TCC for every function: For function  $f$  we require a related function  $TCC_f$  taking the same inputs, and having the property that  $TCC_f(x_1 \dots x_n) = T$  if and only iff  $f(x_1 \dots x_n)$  is defined. CVC additionally requires that a  $TCC_f$  be a quantifier free formula, involving only total functions<sup>5</sup>. Next we need to convert arbitrary formulas. The conversion is:

$$(\phi)^* = \begin{cases} P(t_1, \dots, t_n) \wedge (t_1)^* \wedge \dots \wedge (t_n)^* & \phi = P(t_1, \dots, t_n) \text{ for atomic predicate } P \\ TCC_f(t_1, \dots, t_n) \wedge (t_1)^* \wedge \dots \wedge (t_n)^* & \phi = f(t_1, \dots, t_n) \text{ for function } f \\ T & \phi = x \text{ for variable } x \\ (\psi)^* \wedge (\chi)^* & \phi = \psi \wedge \chi \\ \neg(\psi)^* & \phi = \neg\psi \end{cases} \quad (\text{TCC-gen})$$

Note: the connectives  $\vee, \rightarrow, \leftrightarrow$  may be defined in terms of  $\wedge, \neg$ ; constants are nullary functions. An example conversion is:

$$\begin{aligned} \left(\frac{1}{x} \neq 0\right)^* &= \left(\neg\left(\frac{1}{x} = 0\right)\right)^* \\ &= \neg\left(\frac{1}{x} = 0\right)^* \\ &= \neg\left(\left(\frac{1}{x} = 0\right) \wedge \left(\frac{1}{x}\right)^* \wedge (0)^*\right) \\ &= \neg\left(\left(\frac{1}{x} = 0\right) \wedge TCC_{(f)}(1, x) \wedge (1)^* \wedge (0)^* \wedge (0)^*\right) \\ &= \neg\left(\left(\frac{1}{x} = 0\right) \wedge (x \neq 0) \wedge T \wedge T \wedge T\right) \\ &= \neg\left(\frac{1}{x} = 0\right) \vee (x = 0) \\ &= \left(\frac{1}{x} \neq 0\right) \vee (x = 0) \end{aligned}$$

The converted function still involves division, however it is now treated as total with an arbitrary fixed value when  $x = 0$ . This value is irrelevant since the right disjunct is true when  $x = 0$ . An instance of (TCC-equiv) summarizes these observations as:

$$\Gamma \models_p \left(\frac{1}{x} \neq 0\right) \text{ if and only if } \Gamma^* \models_c \left(\frac{1}{x} \neq 0\right) \vee (x = 0) \quad (5)$$

where  $\Gamma$  contains the axioms for arithmetic and propositional logic.

2.4.1. *Problems With The Conversion.* We now encounter two problems, one is a lack documentation, the other a flaw in `pfsys`'s implementation. The first problem is that (TCC-gen) neglects to define the TCC for the if-then-else function. At the time of writing, it is not

<sup>5</sup>otherwise it is not modelable in CVC, or, respectively, we need further recursion to convert  $TCC_f$ , and it is not clear that this will terminate

$$\begin{array}{c}
\frac{\vdots}{\vdash_c \phi^*} \quad \frac{\vdots}{\vdash_p \phi \text{ iff } \vdash_c \phi^*}}{\vdash_p \phi} \quad \text{(a)}
\end{array}
\qquad
\begin{array}{c}
\frac{\vdots}{\phi^*} \quad \frac{\vdots}{\phi = \phi^*}}{\phi} \quad \text{(b)}
\end{array}$$

FIGURE 6. Correct and Incorrect Conversions

clear what the exact rule is for transforming if-then-else's, but it is certainly not treated as a regular function.

The second problem is that `pfsys` fails to use two syntactic judgments  $\vdash_c$  and  $\vdash_p$  to model the distinct semantic notions of entailment,  $\models_c$  and  $\models_p$ . It models the equivalence in (TCC-equiv) as in Figure 6 (b), where it should be using Figure 6 (a). Moreover, it confuses three distinct notions of equality: equality in classical logic, equality in the logic of partial functions, and meta-equality expressed by “if and only if”. The same LF operator is used for all three, and this is the basis of the unsoundness in appendix 8.3.1.

### 3. RESEARCH PLAN

In this section I outline the basic project. I will attempt to: convince the reader that the project is worthwhile and plausible; give a breakdown of tasks; and summarize its status.

**3.1. Evaluation of the Project.** The project will be a success when I have a tactic that: is high-assurance; contributes capabilities not possessed by other Isabelle tactics; whose domain is easy for new users to understand; and is fast enough for interactive use.

The big challenge in achieving the first three of these goals is minimizing incompleteness. Due to Isabelle's architecture, the resulting tactic will be sound no matter how carelessly written: but every error decreases completeness. Due to the nature of decision procedures, the tactic's domain is easy to explain. It is capable of solving quantifier-free problems built from:

- equality (=)
- predicate logic ( $\vee$ ,  $\wedge$ ,  $\neg$ ,  $\rightarrow$ ,  $T$ ,  $F$ , variables)
- linear arithmetic ( $+$ ,  $-$ ,  $<$ ,  $\leq$ , and scalar multiplication) over  $\mathbb{Q}$   
(and over  $\mathbb{N}$  and  $\mathbb{Z}$  though this is incomplete)

- algebraic data types (e.g. enumerations, lists, trees)
- uninterpreted functions

These are clearly new capabilities. Every incompleteness is an exception to the regular description, and decreases its understandability. One such incompleteness in handling theorems in  $\mathbb{N}$  and  $\mathbb{Z}$  by translating them to theorems in  $\mathbb{Q}$ . Another incompleteness is due to an oddity in CVC’s proof system (§8.3.3).

3.1.1. *Modes.* Proof generation adds overhead to CVC’s execution, and transposing proofs to tactics takes yet more time. Now we discuss a possible trade of usability for speed.

My proposed tactic has two modes of use: fast and sound. Sound mode transposes a CVC proof to an Isabelle tactic. Fast mode generates no proof, it simply trusts CVC’s when it says “yes”. The earlier description of Isabelle is not quite accurate: Isabelle also allows this trusted use, but flags any theorem dependent upon the external oracle with a warning of possible unsoundness.

If sound mode is too slow for interactive use then the envisaged use of the tactic is that fast mode is used for exploration and then, once a proof is found, the user restarts the tactic in sound mode in the background. It’s possible that sound mode fails where fast mode succeeded, but this should be rare as it would arise from a bug in CVC.

The existence of two modes complicates usage. It is preferable to have sound mode fast enough, but speed is currently a low priority. In the remainder of the paper we discuss sound mode exclusively, unless otherwise noted.

### 3.2. Plan of Attack.

**Basic fast mode:** The scope of this step is fast mode for the theories for which the correspondence between Isabelle and CVC is clear: logic, arithmetic (on  $\mathbb{Q}$ ), uninterpreted functions. No proof transpositions are required.

**Logic in sound mode:** This is a prerequisite for other work, since logic is the glue that binds together propositions in other theories. This step includes making a framework for transposing proofs to tactics, and coding specific rules. About 60 of Flea’s 190 rules relate to propositional logic and equality.

**Uninterpreted functions in sound mode:** These are needed to implement division (§4.2.3). Only 1 new rule is needed.

**Arithmetic in sound mode:** This is next due to both its general usefulness (e.g. [Hei99, chapter 8]) and to complement basic fast mode. About 35 new rules are needed.

**Algebraic data types (in both modes):** Isabelle has a facility for extracting declarations of all data types visible from the current context. Both modes require extracting declarations, and translating them to CVC’s input language. Adding data types to sound mode requires 15 new rules.

**Extending arithmetic to  $\mathbb{N}$  and  $\mathbb{Z}$ :** This step is useful, but introduces incompleteness.

**3.3. Current Status.** Basic fast mode is done, though it does not handle division by zero. Logic in sound mode is partly done: 30 of the 60 logic rules are implemented; those that remain are straightforward, with the most serious issue being rule schemata (§4.2.5). I have also examined proofs from the other steps. The main obstacle presented by these is the treatment of partial functions, which has resulted in a major redesign of the interface between Isabelle and CVC, and will be discussed in detail in §4.2.

## 4. RESULTS AND CHALLENGES

In this section we discuss the results so far as well as some of the challenges encountered. My results are a partial implementation of the proof transposer, and more importantly, a design that overcomes the problems encountered in the initial work.

**4.1. Initial Work.** My initial, naïve view of the interface for sound mode is shown in Figure 7. In it, a valid query from Isabelle is fed to CVC, which produces a proof. I need to transpose this proof into an Isabelle tactic, which will then establish the query as an actual theorem within Isabelle.

**4.1.1. A Partial Implementation.** Isabelle is written in the programming language ML [Wik87], and a user may write additional tactics in ML as well. My first step was to

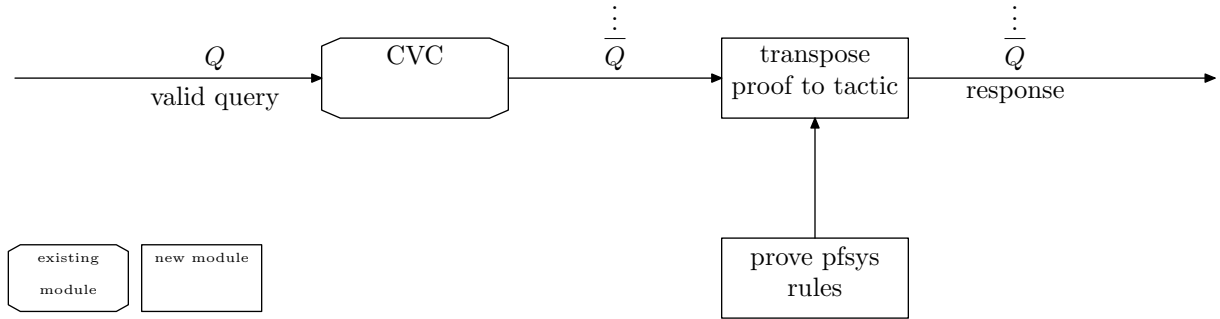


FIGURE 7. Naïve Design

create an interface to an unsound oracle. This interface works by traversing Isabelle’s internal representation of a theorem, writing a corresponding CVC query to a temporary file, calling CVC, and checking whether the result is “yes”.

My next step was to transpose proofs generated by CVC. I suspected the integration would not be as easy as Figure 7, but was not aware of the specific complications. I implemented scanner and parser modules to handle the proof language. In the absence of a specification for the proof language, I had to revise those modules each time a new language feature appeared in CVC’s output.

The rules module and proof transposition module are shown in Figures 7. Proofs of `pfsys`’s rules are encoded in the rules module: as it loads, Isabelle uses these proofs to verify `pfsys`’s rules. The transposition function traverses the tree built by the parser. The rough idea is that for each rule encountered in the tree, this function calls the corresponding Isabelle rule to construct a tactic, though the details sometimes vary (e.g. §4.2.5).

Currently the transposer handles enough rules to reveal problems with this naïve approach. These two steps are currently not integrated, due to the diversion of dealing with semantic mismatches (§4.2).

4.1.2. *Initial Difficulties.* A number of issues made initial progress on this project very slow.

There was a unexpectedly large degree of reverse engineering. The only existing document on the proof production is Aaron Stump’s Ph.D. thesis [Stu02]. Moreover, neither the logic, `pfsys`, nor the output of CVC were meant for humans to read.

LF is more difficult to use than most logical systems, partly because of dependent types. Before I was able to understand the proofs output by CVC I needed to do some exercises to learn the system. I found it very fruitful to work through some material from an introductory logic text [HR00, chapter 1].

Ideally, when a programmer alters a module he also writes a few, simple inputs to test his changes. I was not able to do that with the proof transposer. Hand-coding proofs in LF (beyond basic exercises) is not an effective use of time due to there being a large number of details required, and the ease of making errors (recall the comparison of Figures 4 and 5). The specific implementation, Flea, is also user-hostile: A single term and its type may be quite long, and any error results in the type-checker failing; In this case Flea provides no diagnostics other than reporting “fail”, so debugging is slow. Other implementations of LF might be easier to use, but switching was not an option since `pfsys` makes heavy use of non-standard extensions to LF.

The only other source of tests is CVC, but CVC proofs have two drawbacks. First, they are too large to casually examine. For instance the proof of  $P \rightarrow (Q \rightarrow R), P \rightarrow Q \vdash P \rightarrow R$  uses 79 rule instances (14 distinct rules), each involving explicit mention of the formulas involved (see appendix 8.2). Second, due to the redundancy of the rules in `pfsys`, an attempt to construct a query whose proof used a particular rule often resulted in the proof using other, equivalent rules. For instance, to get another example of `case_splitting` rule, I made an enumeration type of three colors, asserted that a property held for each color, and queried whether the property held in general<sup>6</sup>. The proof instead invoked two other rules: `distinct_ctor` which, in this case states that red, green and blue are different; and `fd_rule` which, in this case states that given four terms of type color, two must be equal. Due to my inability to construct tests, I add rules on an as-needed basis.

**4.2. Semantic Mismatches.** The logics of Isabelle and CVC differ. In this section I discuss these differences, and the work I have done to overcome them.

The major problem is the treatment of partial functions. We will examine division by zero in detail. Other partial functions within the scope of the project can be dealt with similarly.

---

<sup>6</sup>Aaron Stump suggested this as a way of invoking case-splitting

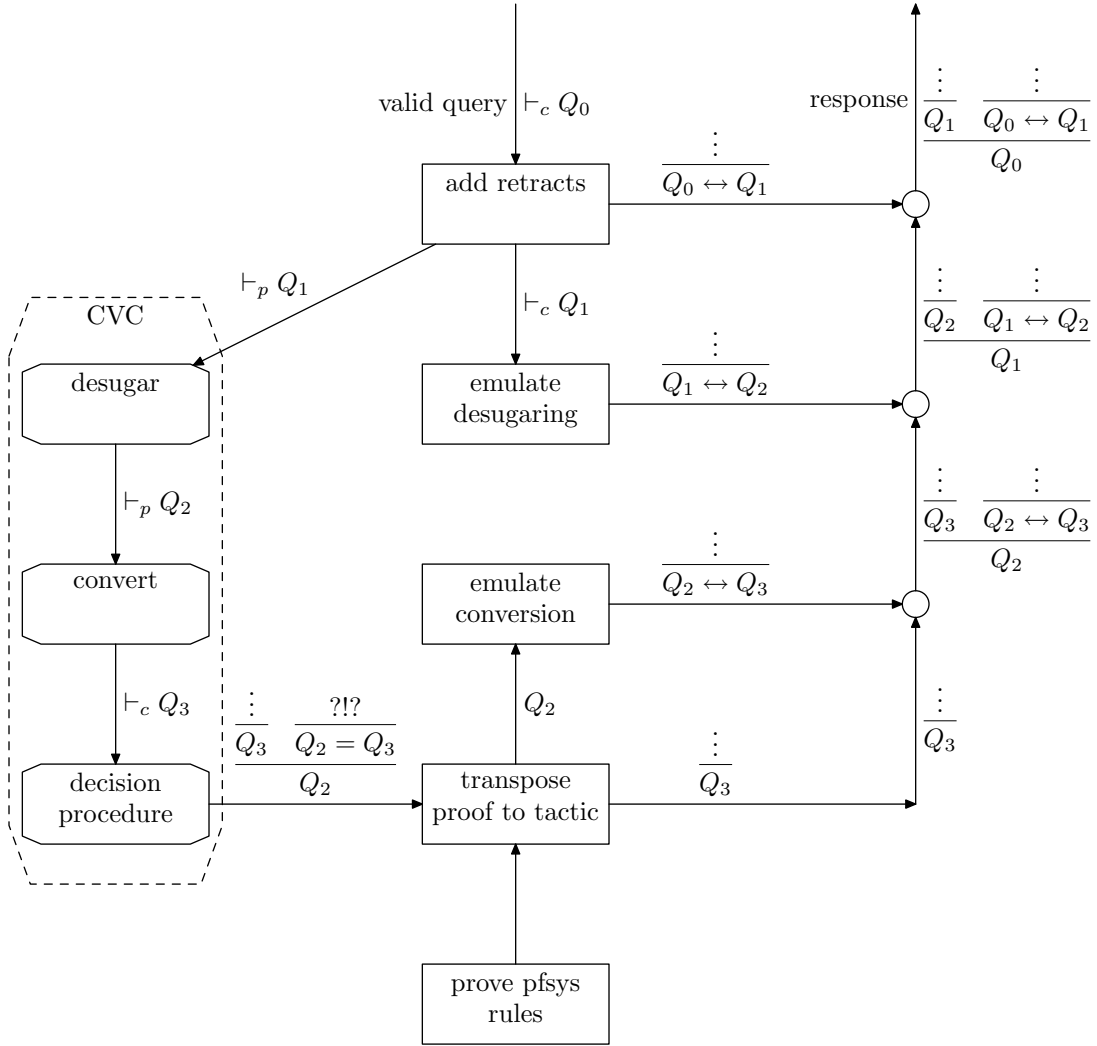


FIGURE 8. Design of Interface

Minor problems include the generality of `pfsys`'s rules, and syntactic sugaring in CVC's input language.

4.2.1. *The Design of the Interface.* Figure 8 shows my proposed interface between Isabelle and CVC. As we go through the challenges, we will see how they motivate this design.

Briefly, the reason for this design is that CVC does some preprocessing of our query: First it translates the input language to its internal language (**desugaring**, §4.2.2); Then it translates from the logic of partial functions to classical logic by generating TCCs (**converting**, §2.4). This conversion is a problem since our queries are *already* in classical logic. There is no way to bypass these steps in CVC, so to compensate: First, to ensure the meanings in

classical logic and logic of partial functions coincide, add retracts to the original query,  $Q_0$ , meanwhile proving that the resulting query,  $Q_1$ , is equivalent (in classical logic). Second prove  $Q_1 \leftrightarrow Q_2$  by stepping through the desugaring process within Isabelle; Third, discard the flawed proof of  $Q_2 \leftrightarrow Q_3$ , and instead prove it by stepping through the conversion within Isabelle; Fourth, transpose the proof of  $Q_3$  to Isabelle tactics; Finally, combine the results for a proof of the initial query  $Q_0$ .

4.2.2. *Desugaring the Input Language.* CVC's versions of  $\wedge$ ,  $\vee$ ,  $\neg$  and  $\rightarrow$  seem to correspond with Isabelle's. However, examining the proofs revealed that CVC's are syntactic sugar for if-then-else,  $n$ -ary conjunction, and  $n$ -ary disjunction. The proofs that CVC provides are for the *desugared* formula, not the input formula. In Figure 8,  $Q_1$  is given to CVC, which desugars it and proves  $Q_2$ .

The desugaring translations are:  $\neg P$  becomes  $P \leftrightarrow F$ ;  $P \rightarrow Q$  becomes (**if**  $P$  **then**  $Q$  **else**  $T$ ); binary connectives  $\wedge$  and  $\vee$  becomes instances of  $n$ -ary conjunction and disjunction, respectively. The problem with negation and implication is that the proof does not match the query. The problem with conjunction and disjunction is that the  $n$ -ary connectives are more flexible than binary connectives. For instance they allow flattening  $P_1 \wedge (P_2 \wedge P_3)$  to  $\bigwedge_{i=1}^3 P_i$ , for which there is no binary analog.

The solution is to do a parallel desugaring within Isabelle, and in tandem prove the equivalence of the two forms  $Q_1$  and  $Q_2$ . Modeling the desugared forms requires embedding if-then-else and arbitrary conjunctions and disjunctions within Isabelle.

4.2.3. *Partial Functions.* Partial functions appear commonly with algebraic data types. For instance lists are built with two constructors, **cons** and **nil**. The constructor **cons** has selectors **head** and **tail**, which should not be applied to **nil**. In general, an  $n$ -ary constructor,  $C$ , has  $n$  selectors to access subcomponents, but should only be applied to terms built with  $C$ . In CVC a mis-applied selector is undefined, while in Isabelle it's assigned an unknown, arbitrary value.

Division is a partial function. Consider the formula  $1/0 = 1/0$ : in CVC it is false, since  $1/0$  is undefined; while in Isabelle it is true by reflexivity. Thus any proof that depends upon this formula's falsity cannot be transposed to an Isabelle tactic. We will initially focus on

division, since it is the single impediment to progress in handling arithmetic. The proposed solution extends to algebraic data types.

After learning about TCCs it is apparent that any solution to partial functions involves a nontrivial translation of queries. An approach is to use a **retract**, to effectively extend a partial function to a total one. Given an inclusion  $S \subseteq B$ , the function  $r : B \rightarrow S$  is a retract if its restriction to  $S$  is the identity. For instance, the function  $r : \mathbb{Q} \times \mathbb{Q} \rightarrow \mathbb{Q} \times (\mathbb{Q} - \{0\})$  defined by

$$r(x, y) = \begin{cases} (x, y) & y \neq 0 \\ (a_x, b_x) & y = 0, \text{ where } b_x \neq 0 \end{cases} \quad (6)$$

is a retract ( $a_x, b_x$  are arbitrary, possibly depending on  $x$ ). Now, abbreviating  $\frac{x}{y}$  by  $d(x, y)$ , the Isabelle semantics for both  $d(x, y) = d(x, y)$  and  $d(r(x, y)) = d(r(x, y))$  are the same. In Figure 8 adding retracts to  $Q_0$  yields  $Q_1$ . In tandem with adding retracts we require a tactic to prove  $Q_0 \leftrightarrow Q_1$ .

Encoding retracts in CVC's input language exploits two facts. First, there are simple tests to tell if division (or a selector) is defined on an input: test whether the second argument is 0 (whether the term is built with the correct constructor, respectively). Second, CVC treats uninterpreted functions as total. The trick is to assign an uninterpreted function to the composition of the retract and the partial function, then use if-then-else to determine whether the input is in the functions domain: if so, return the original function; otherwise return the uninterpreted function. For instance, for division

$$x/y \mapsto (\mathbf{if} \ y = 0 \ \mathbf{then} \ arb(x) \ \mathbf{else} \ x/y) \quad (7)$$

where  $x$  and  $y$  may be any expression, and where  $arb(x)$  models  $d(r(x, 0))$  in the case that  $d$  is undefined.

To handle proofs involving the retracts requires an addition to the transposition stage: As the proof tree is traversed, each occurrence of the uninterpreted function is replaced by the original partial function. For division:

$$arb(x) \mapsto x/0 \quad (8)$$



case, another for the inductive step. Figure 9 (c) shows the proof required to perform one instance of `subst_and` within Isabelle. The difference is that “ $\dots$ ” in (b) is part of the rule while “ $\dots$ ” in (c) is part of the proof tree.

The transposer also performs the replacement of rule schemata with sequences of rule applications.

**4.3. Debugging.** The exercise of translating proofs from CVC to Isabelle has had the side-effect of debugging Flea’s language and CVC’s proof generation. I have found a number of unsoundnesses (appendix 8.3). Most have obvious work-arounds. One, the ability to prove both  $1/0 = 1/0$  and  $1/0 \neq 1/0$  within `pfsys`, caused considerable grief, and helped me to better understand CVC, and how to handle TCCs. This last bug also led to the decision to discard the right subproof generated by CVC.

## 5. RELATED WORK

**5.1. High-Assurance Work.** Sean McLaughlin is working to integrate HOL-Lite, another fully-expansive proof assistant, with CVC-Lite in support of the FlySpeck program [Hal]. Of all related work, his project appears the most similar to mine in its high-assurance and its transposition of the oracle’s proofs. No other details are available at present.

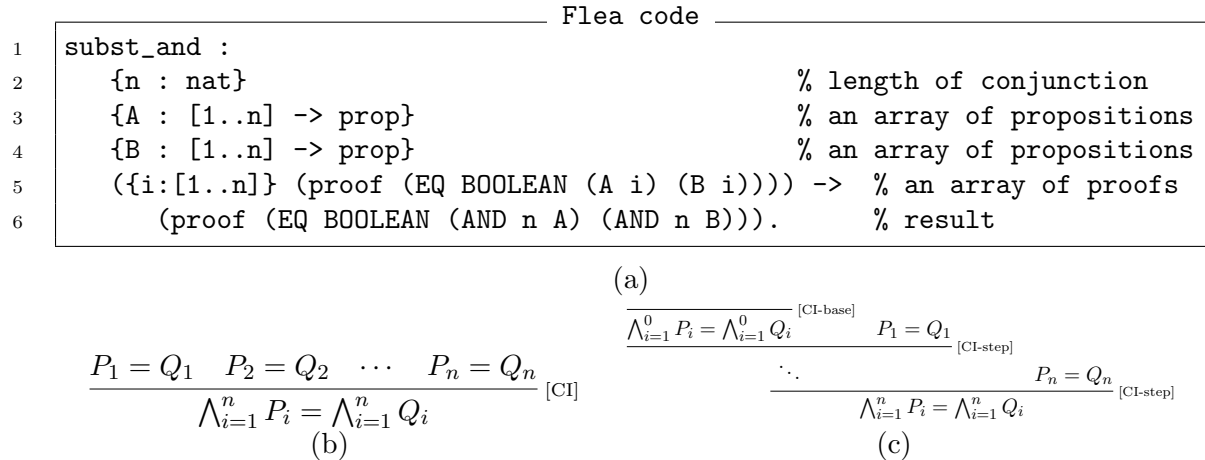


FIGURE 9. The `subst_and` rule

$$\frac{\bigwedge_{i=1}^{n-1} P_i = \bigwedge_{i=1}^{n-1} Q_i \quad P_n = Q_n}{\bigwedge_{i=1}^n P_i = \bigwedge_{i=1}^n Q_i} \text{[CI-step]} \quad \frac{}{\bigwedge_{i=1}^0 P_i = \bigwedge_{i=1}^0 Q_i} \text{[CI-base]}$$

FIGURE 10. Inductive Rules to Model `subst_and`

There has been some work on implementing decision procedures within fully-expansive proof assistants. Boulton has implemented the Nelson-Oppen framework in HOL [Bou95]. Norrish implemented full Presburger arithmetic in HOL [Nor03], and Chaieb and Nipkow [CN03] adapted the work to Isabelle.

These have been many attempts to integrate proof assistants with automated theorem provers for first-order logic. Kumar et al. [KKS91] and Hurd [Hur99] integrated HOL with first-order predicate logic provers. Paulson specifically designed Isabelle's classical reasoner to be integrated [Pau99]. Paulson and Hurd are currently working on a successor to this tool [Pau03a].

**5.2. Other Work.** There have been other integrations of theorem provers with decision procedures. A state-of-the-art work is the Prototype Verification System (PVS), which is integrated with ICS[For03]. Boyer and Moore [BM88] integrated full Presburger arithmetic into their heuristic theorem prover. Heilmann's duration calculus package for Isabelle uses SVC as a trusted oracle for rational arithmetic [Hei99, chapter 8].

The Extended Static Checker for Java (ESC/Java) [FLL<sup>+</sup>02] uses the Simplify combined decision procedure. It provides support for detecting errors such as dereferencing null pointers, accessing beyond array bounds, race conditions, and deadlocks. ESC/Java is neither sound, nor complete, yet it can effectively find bugs (with a few being spurious). It is automated, though the user must document loop invariants and pre- and post-conditions.

## 6. FUTURE WORK

The project is far from complete. Only fragments of the design sketched in §4.2.1 have been implemented. Much work is needed to realize the plan presented in §3, and even then it does not take full advantage of CVC. I also need to re-evaluate whether to continue using CVC.

**6.1. Implementation.** I believe am now in a position to implement the plan in §4.2.1. The existing code only implements parts of the naïve design, however I anticipate that most of the work will be straight-forward programming.

6.1.1. *Efficiency.* It is too early in the project for efficiency to be a priority. CVC, without proofs, is one of the fastest combined decision procedures. This speed suggests that the use of fast mode together with sound mode described in §3.1.1 is adequate for most purposes.

For sound mode, the use of retracts may cause slowdowns. Each use of if-then-else in a division may cause a case split. CVC stores its proofs as directed acyclic graphs (DAGs) to share commonly occurring subproofs, so CVC's proof size is probably immune to these splits. My implementation currently expands each DAG to tree, which causes an exponential slowdown in the worst case.

There are two constant-factor speedups. First, Isabelle and CVC currently communicate by writing to files. Using pipes should be faster. Second, Isabelle's manipulation of proofs depends upon parsing strings. There may be a more direct interface.

## 6.2. Taking Advantage of CVC's Other Capabilities.

6.2.1. *Better Error Messages.* Providing proofs with affirmative answers suggests the idea of providing counter-examples with negative examples. When most of Isabelle's tactics fail, they provide no information about what went wrong. CVC can provide the set of assumptions that make the query false. The information is not as specific as a counter-example, but it can still help the user.

6.2.2. *CVC's Other Theories.* CVC also handles records, tuples, and arrays. The proof translation for these theories should be straightforward, after adding algebraic data types and functions. The unclear part is which features in Isabelle they correspond to.

6.3. **Other Theories.** CVC does not handle quantification. This is a problem for interpreted functions outside of CVC's domain, for instance the greatest common divisor of two integers (gcd). Since these functions are outside CVC's domain they would have to be passed as uninterpreted functions, but without properties such as  $\forall x, y. \text{gcd}(x, y) = \text{gcd}(y, x)$ . The only way to pass such information to CVC is to instantiate it. No matter how many instantiations are passed to CVC, information is lost. In this case CVC will be incomplete, and the challenge is to minimize the incompleteness.

In such a case it should be possible to make CVC have a **dialog** with Isabelle. If CVC is otherwise unable to prove a property involving  $\gcd(u, v)$ , it may ask Isabelle what else it knows about  $\gcd(u, v)$ , Isabelle would assert  $\gcd(u, v) = \gcd(v, u)$ , and CVC would try to continue using this new fact. It is not clear how to do this effectively.

**6.4. Switch to CVC-Lite.** Before continuing it may be opportune to switch to CVC-Lite, the successor of CVC. CVC-Lite’s capabilities for proof production are currently basic, but the designers are willing to accommodate proof consumers. The benefits of CVC-Lite are: it is supported, while CVC is not<sup>7</sup>; it generates natural deduction proofs, which are simpler to understand; it has a more direct interface to proof objects; it has some support for quantifiers and nonlinear arithmetic. The drawbacks are: it appears that CVC-Lite also require queries to be expressed in a different language than they are processed (in this case a 3-valued logic) so a work-around may still be needed; all the proof rules are different from CVC’s.

I was aware of CVC-Lite when I started, but due to time constraints of the RPE and the uncertain schedule for CVC-lite’s development I was unable to risk using it.

## 7. CONCLUSIONS

The main benefit of this project is a much better understanding of the requirements to soundly integrate Isabelle with CVC, and a realistic design for the interface. One frustrating lesson is that tools designers, in this case the CVC team, should not require the use of an unnatural interface to their tools: My design would have been much simpler if I could have bypassed their input language that uses the logic of partial functions.

Acknowledgments: Thanks to John Matthews for initial research ideas, many helpful discussions, and for comments on earlier drafts. Thanks to Nathan Linger and Perry Wagle for helpful comments. Thanks to Aaron Stump and Sergey Berezin for responses to questions posted to various CVC and CVC-Lite mailing lists.

---

<sup>7</sup>It is a shame that the thorough scrutiny required for proof translation is not being applied toward a supported tool

## REFERENCES

- [BM88] R. S. Boyer and J. Strother Moore. Integrating decision procedures into heuristic theorem provers: A case study with linear arithmetic. *Machine Intelligence*, 11:83–124, 1988.
- [Bou95] R. J. Boulton. Combining decision procedures in the HOL system. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 75–89, Aspen Grove, UT, USA, September 1995. Springer–Verlag.
- [CN03] Amine Chaieb and Tobias Nipkow. Generic proof synthesis for presburger arithmetic. Technical report, October 2003.
- [FLL<sup>+</sup>02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245. ACM Press, 2002.
- [For03] Formal Methods Program. Formal methods roadmap: PVS, ICS, and SAL. Technical Report SRI-CSL-03-05, Computer Science Laboratory, SRI International, Menlo Park, CA, October 2003.
- [Gor00] Michael Gordon. From LCF to HOL: a short history. In G. Plotkin, Colin P. Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.
- [Hal] Thomas C. Hales. The flyspeck project fact sheet.  
<http://www.math.pitt.edu/~thales/flyspeck/>.
- [Hei99] Søren T. Heilmann. *Proof Support for Duration Calculus*. PhD thesis, Technical University of Denmark, January 1999.
- [HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.
- [HR00] Michael R A Huth and Mark D Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2000.
- [Hur99] Joe Hurd. Integrating Gandalf and HOL. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Proceedings of the 12<sup>th</sup> International Conference on Theorem Proving in Higher Order Logics (TPHOLs’99)*, volume 1690 of *Lecture Notes in Computer Science*, pages 311–321, Nice, France, September 1999. Springer–Verlag.
- [KKS91] R. Kumar, T. Kropf, and K. Schneider. Integrating a first-order automatic prover in the HOL environment. In M. Archer, J.J. Joyce, K.N. Levitt, and P.J. Windley, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 170–176, Davis, California, 1991. IEEE Computer Society Press.
- [NO79] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

- [Nor03] Michael Norrish. Complete integer decision procedures as derived rules in HOL. In D. Basin and B. Wolff, editors, *Proceedings of the 16<sup>th</sup> International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *Lecture Notes in Computer Science*, pages 71–86. Springer–Verlag, 2003.
- [Pau89] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.
- [Pau99] L.C. Paulson. A Generic Tableau Prover and its Integration with Isabelle. *Journal of Universal Computer Science*, 5(3):73–87, March 1999.
- [Pau03a] Lawrence C. Paulson. Automation for interactive proof, 2003. Grant proposal.  
Available at <http://www.cl.cam.ac.uk/~lcp/Grants/automation.html>.
- [Pau03b] Lawrence C. Paulson. *The Isabelle Reference Manual*, 2003. Part of the Isabelle distribution.
- [SBD02] A. Stump, C. Barrett, and D. Dill. CVC: a Cooperating Validity Checker. In *14th International Conference on Computer-Aided Verification*, 2002.
- [SD99] Aaron Stump and David L. Dill. Generating proofs from a decision procedure. In A. Pnueli and P. Traverso, editors, *Proceedings of the FLoC Workshop on Run-Time Result Verification*, 1999. Trento, Italy.
- [Stu02] Aaron Stump. *Checking Validities and Proofs with CVC and flea*. PhD thesis, Stanford University, August 2002.
- [Szp03] George Szpiro. Mathematics: Does the proof stack up? *Nature*, 424:12–13, July 2003.
- [Wik87] Åke Wikström. *Functional Programming Using Standard ML*. Prentice Hall, 1987.

## 8. APPENDIX

### 8.1. Notation.

$\mathbb{B}$  booleans  $\{T, F\}$   
 $\mathbb{N}$  natural numbers  $\{0, 1, 2, \dots\}$   
 $\mathbb{Z}$  integers  $\{\dots, -2, -1, 0, 1, 2, \dots\}$   
 $\mathbb{Q}$  rationals  $\{\frac{0}{1}, \frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \frac{2}{3}, \dots\}$

Comparative notation across systems:

notion	math	Isabelle	CVC input	pfsys.flea
entailment	$P, Q \vdash R$	$P \implies Q \implies R$	ASSERT P; QUERY Q;	.
implication	$P \rightarrow Q$	$P \dashrightarrow Q$	$P \Rightarrow Q$	(translated to if-then-else)
equivalence	$P \leftrightarrow Q$	$P = Q$	$P \Leftrightarrow Q$	EQ BOOLEAN P Q
negation	$\neg P$	$\bar{P}$	NOT P	(translated to $P = F$ )
conjunction (binary)	$P \wedge Q$	$P \& Q$	P AND Q	(translated to $n$ -ary conjunction)
disjunction (binary)	$P \vee Q$	$P   Q$	P OR Q	(translated to $n$ -ary disjunction)
conjunction ( $n$ -ary)	$\bigwedge_{i=1}^n P_i$	*AND [P1 ... Pn]	.	AND n [[P1 ... Pn]]
disjunction ( $n$ -ary)	$\bigvee_{i=1}^n P_i$	*OR [P1 ... Pn]	.	OR n [[P1 ... Pn]]
exclusive or	.	.	P XOR Q	(translated to $(P = Q) = F$ )
if-then-else	.	*?	IF P THEN x ELSE y ENDIF	ITE type P x y

Items in the Isabelle column are part of Isabelle/HOL, except those with with (\*) which I have defined for this project.

**8.2. Example CVC Proof.** Exercise 1.4.2.c from Huth and Ryan [HR00] asks for a proof that  $P \rightarrow (Q \rightarrow R), P \rightarrow Q \vdash (P \rightarrow R)$ . The proof in their system is quite simple: 1 use of implication introduction, and 3 uses of implication elimination.

Modeling the same problem in CVC, the input is:

```
P,Q,R: BOOLEAN;
ASSERT P=>(Q=>R);  ASSERT P=>Q;  QUERY (P=>R);
DUMP_SIG; DUMP_PROOF;
```

The signature file contains declarations and assertions of the problem, and quite oddly, an axiom:

```
% signature for user-declared variables and types
P : cvc_l_ = ((trm BOOLEAN)).
Q : cvc_l_1 = ((trm BOOLEAN)).
R : cvc_l_2 = ((trm BOOLEAN)).

% Assertions made by the user are given here
_cvc_assumption_0 : cvc_l_3 = ((pf cvc_l_4 = ((ITE BOOLEAN P cvc_l_5 = ((ITE BOOLEAN Q R TRUE)) TRUE))))).
_cvc_assumption_1 : cvc_l_6 = ((pf cvc_l_7 = ((ITE BOOLEAN P Q TRUE))))).
cvc_fd_elements_ : cvc_l_8 = ((fd_has_elements BOOLEAN 2 cvc_l_9 = ([[ FALSE, TRUE ] ] ))).
```

The following proof was generated by CVC<sup>8</sup>. It is formatted so that: each line is either a rule, an assumption, or a reference to previous subproof; indentation indicates subproofs. In it 14 distinct rules are applied a total of 79 times.

```
((equiv_mp2 cvc_l_11 = ((ITE BOOLEAN P R TRUE)) cvc_l_11
  ((case_splitting BOOLEAN P 2 cvc_l_13 = ([[ FALSE, TRUE ] ] ) cvc_fd_elements_ cvc_l_11
    ([[ ([hyp_0_FALSE : cvc_l_16 = ((pf cvc_l_17 = ((EQ BOOLEAN P FALSE)))]
      ((equiv_mp2 cvc_l_11 TRUE
        true_is_a_thm
        ((eq_trans BOOLEAN cvc_l_11 cvc_l_20 = ((ITE BOOLEAN FALSE R TRUE)) TRUE
          ((subst_ite BOOLEAN cvc_l_22 = ([[ P, R, TRUE ] ] ) cvc_l_23 = ([[ FALSE, R, TRUE ] ] )
            ([[ hyp_0_FALSE
              , cvc_l_25 = ((eq_refl BOOLEAN R))
              , cvc_l_26 = ((eq_refl BOOLEAN TRUE)) ]]) **))
          ((ite_rewrite2 BOOLEAN R TRUE)))))))))
  , ([hyp_1_TRUE : cvc_l_29 = ((pf cvc_l_30 = ((EQ BOOLEAN P TRUE)))]
    ((equiv_mp2 cvc_l_11 R
      ((case_splitting BOOLEAN R 2 cvc_l_13 cvc_fd_elements_ R
        ([[ ([hyp_2_FALSE : cvc_l_35 = ((pf cvc_l_36 = ((EQ BOOLEAN R FALSE)))]
          ((equiv_mp2 R FALSE
            ((case_splitting BOOLEAN Q 2 cvc_l_13 cvc_fd_elements_ FALSE
              ([[ ([hyp_3_FALSE : cvc_l_41 = ((pf cvc_l_42 = ((EQ BOOLEAN Q FALSE)))]
                ((false_implies_anything FALSE
                  ((eq_rewrite3_nd FALSE
                    ((false_implies_anything cvc_l_46 = ((EQ BOOLEAN TRUE FALSE))
                      ((equiv_mp cvc_l_48 = ((ITE BOOLEAN P Q TRUE)) FALSE _cvc_assumption_1
                        ((eq_trans BOOLEAN cvc_l_48 cvc_l_50 = ((ITE BOOLEAN TRUE FALSE TRUE)) FALSE
                          ((subst_ite BOOLEAN cvc_l_52 = ([[ P, Q, TRUE ] ] ) cvc_l_53 = ([[ TRUE, FALSE, TRUE ] ] )
                            ([[ cvc_l_55 =
                              ((true_prop_equals_true P
                                ((equiv_mp cvc_l_30 P
                                  hyp_1_TRUE
                                  ((eq_rewrite2 P))))))
                                , hyp_3_FALSE
                                , cvc_l_26 ]]) **))
                                cvc_l_58 = ((ite_rewrite1 BOOLEAN FALSE TRUE)))))))))))))
                , ([hyp_4_TRUE : cvc_l_60 = ((pf cvc_l_61 = ((EQ BOOLEAN Q TRUE)))]
                  ((false_implies_anything FALSE
                    ((eq_rewrite3_nd FALSE
                      ((eq_symm BOOLEAN FALSE TRUE
                        ((eq_symm BOOLEAN TRUE FALSE
                          ((false_implies_anything cvc_l_46
                            ((equiv_mp cvc_l_68 = ((ITE BOOLEAN Q FALSE TRUE)) FALSE
                              ((equiv_mp cvc_l_70 = ((ITE BOOLEAN Q R TRUE)) cvc_l_68
```

<sup>8</sup>Note that in the paper I used `proof` and `prop` where `pfsys` really uses `pf` and `o`, respectively.

```

((equiv_mp cvc_1_72 = ((ITE BOOLEAN P cvc_1_70 TRUE)) cvc_1_70 _cvc_assumption_0
 (eq_trans BOOLEAN cvc_1_72 cvc_1_74 = ((ITE BOOLEAN TRUE cvc_1_70 TRUE)) cvc_1_70
  ((subst_ite BOOLEAN cvc_1_76 = ([ P, cvc_1_70, TRUE ]) ) cvc_1_77 = ([ TRUE, cvc_1_70, TRUE
    ([ cvc_1_55
      , ((eq_refl BOOLEAN cvc_1_70))
      , cvc_1_26 ]) **))
    ((ite_rewrite1 BOOLEAN cvc_1_70 TRUE))))))
((subst_ite BOOLEAN cvc_1_82 = ([ Q, R, TRUE ]) ) cvc_1_83 = ([ Q, FALSE, TRUE ]) )
 ([ cvc_1_85 = ((eq_refl BOOLEAN Q))
  , hyp_2_FALSE
  , cvc_1_26 ]) **))
((eq_trans BOOLEAN cvc_1_68 cvc_1_50 FALSE
 (subst_ite BOOLEAN cvc_1_83 cvc_1_53
  ([ (true_prop_equals_true Q
    ((equiv_mp cvc_1_61 Q
      hyp_4_TRUE
      ((eq_rewrite2 Q))))))
  , ((eq_refl BOOLEAN FALSE))
  , cvc_1_26 ]) **))
 cvc_1_58)))))))))))))
hyp_2_FALSE)))
, (([hyp_5_TRUE : (pf cvc_1_95 = ((EQ BOOLEAN R TRUE)))]
 (equiv_mp2 R TRUE
 true_is_a_thm
 ((true_prop_equals_true R
 (equiv_mp cvc_1_95 R
 hyp_5_TRUE
 (eq_rewrite2 R)))))) ]))
((eq_trans BOOLEAN cvc_1_11 cvc_1_101 = ((ITE BOOLEAN TRUE R TRUE)) R
 (subst_ite BOOLEAN cvc_1_22 cvc_1_103 = ([ TRUE, R, TRUE ]) )
 ([ cvc_1_55
  , cvc_1_25
  , cvc_1_26 ]) **))
 ((ite_rewrite1 BOOLEAN R TRUE)))))) ]))
((eq_refl BOOLEAN cvc_1_11))).

```

**8.3. Unsoundness of pfsys, and related problems.** The most surprising result of this project was finding pfsys’s unsoundness.

The most serious of these, for this project, is the indiscriminate mixing of logics described in §8.3.1. A number of minor (fixable) problems follow.

A few of pfsys’s rules are unsound in the sense that it is possible to hand-code a proof of false which exploits them. It may be that CVC never takes advantage of these exploits: for instance CVC might always check that  $i \neq j$  before using the rules of §8.3.2. In other words, the system may not be unsound if we insist that Flea only be used with proofs output by CVC, and we verify that CVC never exploits the problems found.

However, the modularity of the system is broken. One goal of using Flea is to have an independent verification of CVC’s results. With each additional bug, this independence is further eroded, as validity depends upon knowing that CVC does not exploit that bug. Verification of Flea now requires auditing the 150,000 lines of CVC code.

**8.3.1. Mixing Classical Logic and the Logic of Partial Functions.** The contradictory statements  $1/0 \neq 1/0$  and  $1/0 = 1/0$  are both proved in the same example (CVC input: QUERY NOT  $1/0=1/0$ ;). They seem to originate from  $\vdash_p 1/0 \neq 1/0$  and  $\vdash_c 1/0 = 1/0$  in the intended proof, however pfsys does not distinguish between  $\vdash_c$  and  $\vdash_p$ , and it uses the same term, EQ BOOLEAN, to model meta-equality, and boolean equality in both classical logic and the logic of partial functions.

This flaw is an unsoundness. It has the greatest affect on this project since it is not obvious how to work around it. Discarding the right subtree of CVC’s proof, and reconstructing it from knowledge of CVC’s internal workings seems to be the best workaround.

My current hypothesis is that the logic of partial functions and meta-equality only appear *near* the root of the proof tree. Specifically:

- the formula at the root is in the logic of partial functions,
- every node on the left child of the root is in classical logic,
- each path from the root, through the right child, and on to a leaf has an occurrence of the rule `insert_tcc`, moreover:
  - every node between the root and this rule is in the logic of partial functions
  - the node at this rule has a mix of classical logic and the logic of partial functions connected with a meta-equality
  - every node between this rule and the leaf is in classical logic

The authority on how equality is used are the designers of the system. I have reported the bug to CVC's bug-list, and asked for clarification, but have yet to receive a reply.

8.3.2. *Unstated preconditions:  $i \neq j$ .* The problem is that a number of rules are missing a necessary precondition that their arguments be distinct. This is a minor flaw that should be easy to work around. It should also be easy fix in CVC, if the maintainers choose to do so.

The first is `and_or_drop_duplicate` which states that in an  $n$ -ary conjunction or disjunction, if the  $i^{\text{th}}$  and the  $j^{\text{th}}$  entries are identical then the  $j^{\text{th}}$  entry may be dropped. The intent is to reduce a formula like  $P \wedge Q \wedge P$  to  $P \wedge Q$ . However, without the precondition that  $i \neq j$  we could erroneously reduce  $F \wedge T$  to  $T$  by dropping the  $F$ . `pfsys` omits this precondition.

The second is `elim_tester_pos3` which states that in an algebraic data type, if the  $i^{\text{th}}$  tester is applied to the  $j^{\text{th}}$  constructor (which, in turn is applied to anything) then the result is false. This rule should only apply if  $i \neq j$ , but again there is no such check.

Two other rules, `arrcons_duplicate` and `adt_occurs_check`, appear to have the same problem.

The solution for the transposer is to generate the extra condition  $i \neq j$ , and to automatically discharge it. The values  $i$  and  $j$  should be constants, so discharging will be trivial. If these oversights are ever exploited, the proof translation will fail, resulting in a completeness error. However, it is unlikely that CVC will exploit this oversight.

8.3.3. *Axioms Generated On-the-fly.* The problem encountered was that one of the `pfsys` rules, `case_splitting` which handles case analysis, has the peculiar feature that important data are dynamically generated. For instance, when case-splitting is used on a proposition the assertion is made that the booleans consist of the values  $T$  and  $F$ , in that order.

For booleans, this is a trivial problem. For cases other than booleans, the translator will fail, as I cannot anticipate how this rule will be invoked.

However, this erodes the claim that Flea independently verifies proofs. Now an auditor has to examine CVC to ensure that it always generates enumeration axioms correctly (in addition to auditing the Flea source and axioms).

8.3.4. *Arbitrary Rationals.* This is not an unsoundness, but it is an oddity that I have no idea how to work around, if it ever arises.

Embedded deeply in the proof of  $\neg(\frac{1}{0} = \frac{1}{0})$  is the fraction  $\frac{139316248}{139316096}$ . The validity of the proof appears independent of this value: all attempts to change it to another number resulted in a proof that still type-checked. Upon careful examination, it appears that this number is somehow equated to  $\frac{1}{0}$ , however with a dependent type that is a function of this fraction.

My current conjecture is that this number has been arbitrarily assigned to the expression  $\frac{1}{0}$  to make division total, and the type keeps the value from being used in any essential way in the main proof. I suspect such terms will not arise once retracts are added to the queries.