# Comprehensive Exam Report 2 - Relating Higher Order and First-Order Logics

Arjun Viswanathan

## 1 Introduction

In this report, we will discuss the integration of automatic theorem provers (ATPs) with interactive theorem provers (ITPs). We will particularly focus on the ATP category of satisfiability modulo theories (SMT) solvers. ITPs are highly reliable tools with a small proof kernel. They require the user to write elaborate proofs to formalize properties, and provide little automation. ATPs, on the other hand automatically prove formulas without any user intervention. Users would benefit from having the reliability of ITP results with the automation capabilities of ATPs. However, SMT solvers have an enormous code base, and due to the lack of a small, verifiable kernel, are susceptible to bugs. Thus, in a collaboration with an SMT solver, an ITP's trust-base would be extended to include the solver's. To avoid compromising the proof kernel of the ITP, SMT solvers can produce proofs of their results. We explore how proof producing SMT solvers can guide ITPs in finding proofs for formulas - a process called *proof search*. To assist ITPs in the automation of proof search, the proof object from the SMT solver guides the proof search in the ITP via a process called *proof reconstruction*. Reconstruction can be implemented as a meta-procedure. For example, reconstruction of SMT solver proofs in Isabelle/HOL are done external to the prover's logic by Sledgehammer [6]. Here, the external ATP's proof guides *Metis*, Sledgehammer's internal ATP in its proof search. The final proof is found by *Metis* and the SMT solver's proof is only used to prune the search space, and fill in holes. In SMTCoq [11], this reconstruction is expressed within the ITP's logic by a process called *computational reflection*. Here, the SMT solver's logic is embedded in Coq's logic, and then a proof from the SMT solver is directly used to obtain the proof of a formula in Coq. While this reduces the proof obligation within Coq, it incurs an overhead of checking the SMT solver's proof, which is typically large, inside Coq, making it a potentially time consuming process. We look at the details involved in both these approaches of integrating SMT solvers with ITPs.

## 2 Preliminaries

Proof calculi are specified in this document using declarative rules of the form

$$\frac{P_1 \quad ... \quad P_n}{C}\ rule\_name$$

which specifies that if the premises $P_1, \ \ldots, \ P_n$ hold, then the conclusion $C$ holds.

A goal in Coq is proved by manipulating a *proof context* that initially consists of the entire goal, that can be broken down into smaller parts. Coq offers functions called *tactics* that help

manipulate the proof context and find proof terms for goals. We refer to some Coq tactics and explain them inline.

In both Coq and Isabelle/HOL, lists are constructed using square brackets (`[ ]`) and their elements are separated using semicolons (`;`). Arrays are constructed using curly braces (`{ }`) and their elements separated using commas (`,`).

## 2.1 Many-Sorted First-Order Logic

Many-sorted first-order logic (MSFOL) extends first-order logic with types (or sorts). We present the syntax of MSFOL in this section and the semantics are presented in [3]. Syntactically, the components of MSFOL are sorts $\sigma$, terms $t$, and formulas $\phi$. Sorts are atomic entities that represent types. Function types — $(\sigma_1, ..., \sigma_n) \to \sigma$ — and relation types — $(\sigma_1, ..., \sigma_m)$ are defined over sorts, and are types of functions and predicates. Terms $t$ and formulas $\phi$ are specified as:

$$t := x^{\sigma} \mid f^{(\sigma_1,...,\sigma_n)\to\sigma}(t_1, ..., t_n)$$
$$\phi := \bot \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \forall x^{\sigma}.\phi \mid t_1 = t_2 \mid P^{\sigma_1,...,\sigma_m}(t_1, ..., t_m)$$

Terms are either sorted variables, or applications of sorted function symbols to terms. Formulas are constants or logical connectives applied to other formulas, quantified formulas, equality over terms, or predicates symbols applied to terms. Connectives $\top$, $\vee$, $\to$, $\leftrightarrow$, and the existential quantifier $\exists$ can be specified using the connectives and quantifiers mentioned above.

An *atomic formula* is one of $\bot$, $t_1 = t_2$, $P^{\sigma_1,...,\sigma_m}$, and a *literal* is an atomic formula or it's negation. A *clause* is a disjunction of literals. All formulas can be converted to conjunction normal form (CNF) which is a conjunction of clauses. For ease of notation, a formula in CNF is also represented as a set of clauses within curly braces. $\bot$, the symbol representing false, is overloaded to also represent the empty clause, which is trivially unsatisfiable (unlike the empty set, or the empty conjunction which is trivially satisfiable).
The resolution rule simplifies two clauses to a new clause as follows:

$$\frac{\phi_1 \vee ... \vee \phi_n \vee \chi \quad \neg\chi \vee \psi_1 \vee ... \vee \psi_m}{\phi_1 \vee ... \vee \phi_n \vee \psi_1 \vee ... \vee \psi_m}\ resolution$$

The following is an instance of the resolution rule.

$$\frac{a \vee \neg b \quad b \vee c}{a \vee c}$$

Given that two clauses hold, and a *pivot*, which is a literal that occurs with opposite polarities in each clause ($b$ in the above example), the resolution rule concludes that the combination of the two clauses without either occurrence of the pivot holds.

A formula as defined above is represented in Coq using Coq's `Prop` type (for propositions). Coq terms of type `bool` (for Booleans) are represented as propositions by equating them to the Boolean value `true`. We differentiate these from propositions in Coq by referring to them as *Boolean goals*. For instance, the following proposition specifies one of the conjunction elimination rules ($\forall\ a\ b,\ a \wedge b \to a$) in Coq as a proposition.

```
∀ a b :  Prop, a ∧ b -> a
```

An equivalent Boolean goal is:

```
∀ a b :  bool, implb (a && b) a = true
```

SMTCoq is able to prove such Boolean goals and a fragment of propositional goals in Coq. The limits of SMTCoq's abilities in proving propositional and Boolean goals are discussed in report 3. Here, we study how SMTCoq proves Boolean goals.

### 2.2 Higher-Order Logic

We specify the syntax of higher-order logic and delegate the explanation of semantics to a reference [9]. HOL consists of types $\tau$ and terms $t$.

$$\tau := \alpha \mid \kappa^n\ \tau_1 ... \tau_n$$
$$t := x^\tau \mid c^\tau \mid t_1\ t_2 \mid \lambda x^\tau . t$$

Types $\tau$ are either type variables $\alpha$ or applications of type constructors $\kappa^n$ to $n$ types ($n$ is usually omitted). Particular types of interest are the function type — formed by applying the arrow type constructor $\to^2$ to other types, the Boolean type — `Bool` (or $\texttt{Bool}^0$), the type of natural numbers — `Nat`, and integers — `Int`. Note that `Bool` is the type of Booleans in higher-order logic (used in Sec. 4), different from `bool` which is the type of Booleans in Coq (used in Sec. 3). Terms are typed variables, typed constants, applications of terms to terms, or typed $\lambda-$ abstractions. We have the usual Boolean constants representing logical connectives and quantifiers. For instance, logical negation, $\neg^{\texttt{bool}\to\texttt{bool}}$, universal quantification, $\forall^{\alpha\to\texttt{bool}\to\texttt{bool}}$, and polymorphic equality, $=^{\alpha\to\alpha\to\texttt{bool}}$. Type annotations for terms are also often omitted when understood from context.

## 3 SMTCoq

SMTCoq is a skeptical cooperation between the Coq proof assistant, and SAT and SMT solvers, implemented as a Coq plugin. We will maintain a focus on the SMT solver integration of SMTCoq, noting that many features are shared with the SAT solver integration.

ATPs like SMT solvers are susceptible to bugs due to the large code-bases used to support their automation. ITPs like Coq have a small trustable proof kernel which would be compromised if they were to trust external results. In a collaboration with SMT solvers, to avoid extending Coq's trust-base, SMTCoq requires the solvers to be proof-producing, and uses Coq's computational capabilities to lift their proofs up to Coq proofs, in a process called computational reflection.

This is illustrated in Figure 1 using a running example that is elaborated in the rest of the section. The running example is of a propositional formula with no theory literals. This could be proved by a SAT solver. SMT solvers are able to deal with more expressive formulas than the one presented in the example here, but for simplicity, we don't involve theory reasoning in the example. An example involving theory literals is used in Report 3 to demonstrate how SMT solvers perform theory reasoning and produce proofs.

SMTCoq can be invoked within Coq using the `smt` tactic. Before involving the SMT solver, SMTCoq uses Coq tactics to extract the Boolean term from a Boolean goal of this form:

`forall` $Q$`, B = true.`

where $Q$ is a vector of universally quantified variables that appear in `B`, and `B` is the Boolean term that SMTCoq extracts. It performs this extraction using the `intros` tactic in Coq that instantiates the universal quantifiers by introducing the variables into the proof context.

STEP 1: Coq tactic invokes SMTCoq to prove theorem:

```
Theorem F : forall (a b c : bool), (b && negb c) ||
  (a && negb b) || negb a || c = true.
Proof.
  smt.
```

↓

STEP 2: SMTCoq extracts Boolean term `B` from `F` and creates a deep embedding `f` of `B`.

`B: (b && negb c) || (a && negb b) || negb a || c`

↓

STEP 3: SMT solver proves unsatisfiability of $\neg B$ (the SMT encoding of $\neg$`B`) and returns proof `t`. This step is expanded in Figure 2

↓

STEP 4: Call to SMTCoq's checker checks that `t` proves unsatisfiability of $\neg$`f`:

$c :=$ `checker f t = true`

↓

STEP 5: Reflection lemma reflects deep embedding to shallow embedding $\mathbb{I}$(`f`) via call to `checker` from Step 4:

`checker_correct f t (refl_equal` $c$ `true)`

↓

STEP 6: Proof of `F` is completed in Coq:

```
Theorem F : forall (a b c : bool), (b && negb c) ||
  (a && negb b) || negb a || c = true.
Proof.
  smt.
Qed.
```

Figure 1: SMTCoq's journey in proving example Boolean formula `F`

STEP 3.1: $B$ is negated:

$$B : (b \wedge \neg c) \vee (a \wedge \neg b) \vee \neg a \vee c$$
$$\neg B : \neg((b \wedge \neg c) \vee (a \wedge \neg b) \vee \neg a \vee c)$$

↓

STEP 3.2: $\neg B$ after CNF conversion is checked for unsatisfiability by the SMT solver:

$$\{\neg b \vee c, \neg a \vee b, a, \neg c\}$$

↓

STEP 3.3: SMT solver produces proof `t` of unsatisfiability of $\neg B$:

$$\dfrac{\dfrac{\dfrac{\neg b \vee c \qquad \neg a \vee b}{\neg a \vee c} \qquad a}{c} \qquad \neg c}{\bot}$$

Figure 2: The SMT Solver's role in helping SMTCoq prove example formula $F$.

**Example 3.1.** In Step 1 of Figure 1, SMTCoq is invoked via the `smt` tactic on a Boolean goal (`F`) representing the formula $\forall\ a\ b\ c,\ (b \wedge \neg c) \vee (a \wedge \neg b) \vee \neg a \vee c$. The Boolean term `B` is extracted in Step 2. □

SMTCoq implements a checker for SMT solver proofs (Step 4). It encodes the language of SMT solvers in Coq using two different embeddings. Given a logical framework X, and a logical framework Y, in which we want to represent the language of X, there are two ways to achieve this representation, and both of these are necessary for reflection:

- *Shallow Embedding:* Use types and terms of Y that correspond to those of X, to represent terms of X. A shallow embedding of SMT formulas in Coq is Coq's `bool` type of Booleans.
- *Deep Embedding:* Define custom types in Y that correspond to those of X, and define inside Y, the meaning of terms of these custom types with respect to shallow terms. A deep embedding of SMT formulas is defined in SMTCoq as the type `form` in SMT solvers, and the interpretation function $\mathbb{I}$ (referred to in Step 5) expresses the meaning of terms in the deep embedding with respect those in the shallow embedding.

**Example 3.2.** The Boolean term `B` from Step 2 of Figure 1 is the shallow embedding of SMT formula $B$ from Step 3.1 of Figure 2. `f` from Figure 1 has type `form`, and is the deep embedding of SMT formula $B$. □

SMT solvers refutationally prove input formulas — given a formula $F$, the solver proves that the negation of $F$ ($\neg F$) is unsatisfiable, which is equivalent to proving that $F$ is valid. To do this,

the SMT solve converts $\neg F$ into conjunction or clausal normal form (CNF) which is a conjunction of clauses. It then uses a combination of a SAT solver and multiple theory solvers to reduce the set of input clauses to the empty clause via resolution and other rules of inference. Thus, deriving the empty clause, which represents the most trivial inconsistency in CNF, from the negation of the input clauses proves that it is unsatisfiable. A proof from an SMT solver is a tree with the clauses (from the negation of the input formula) and theory lemmas — clauses that are valid in particular theories — at the leaves and $\bot$ at the root. The theory lemmas are backed by subproofs which come from the respective theory solver. Our example doesn't involve theory lemmas, but examples of these are presented in Report 3. While the actual SMT solver proof typically also consists of proof steps justifying CNF conversion steps, we omit the CNF conversion phase from our presentation. The role of the SMT solver from Figure 1 is elaborated in Figure 2. The SMT solver's proof creation is detailed in Report 3.

**Example 3.3.** In Figure 2, SMT formula $B$ (corresponding to Boolean term `B` from Figure 1) is negated in Step 3.1, and converted to CNF in Step 3.2. Finally, in Step 3.3, the unsatisfiability of $\neg B$ is proven by a refutation tree with the clauses from the CNF in Step 3.2 at the leaves, and $\bot$ at the root. □

Given a deep embedding of formula $B$, and a proof of unsatisfiability of $\neg B$ from the SMT solver, SMTCoq's proof checker checks that the proof proves the unsatisfiability of $\neg B$. If the checker succeeds, SMTCoq is able to reflect the formula in the deep embedding to its shallow embedding. This is done via a correctness lemma for the checker (also called the reflection lemma), which proves that for any formula in the deep embedding, and a proof of the unsatisfiability of its negation from the SMT solver, if the checker returns `true`, then the formula in the shallow embedding holds. A proof of $B$ in the shallow embedding is obtained by instantiating the correctness lemma with $B$. Step 5 from Figure 1 shows the instantiation of the reflection lemma with the right objects to obtain the proof of our example formula. The interpretation function $\mathbb{I}$ transforms formulas in the deep embedding to their shallow embeddings given a valuation $\nu$ of the free variables in them, so $\forall\ \nu,\ \mathbb{I}_\nu(\texttt{f})$ is exactly the term required to close the proof of theorem `F` in Coq. Steps 4 to 6 are covered in detail in the rest of the section.

**Guarantees** SMTCoq is *sound* — when it proves a formula in Coq, the formula is valid and thus the proof can be closed (using `Qed.`, as in Figure 1, Step 6), but not *complete* — when it fails to prove a formula, we can't be certain that the formula isn't valid. This is because, the underlying SMT solver cannot decide certain formulas - given a formula, it could either return `sat` (satisfiable), `unsat` (unsatisfiable), or `unknown` ('don't know'). Additionally, SMTCoq only works for *universal goals* of the form:

```
forall Q, B = true.
```

where $Q$ is a vector of universally quantified variables, and `B` is a Boolean term, and these goals, when negated and sent to the SMT solvers are quantifier-free. It cannot yet reason about goals that contain quantifiers when translated. The theories it currently supports are the theories of equality over uninterpreted functions (EUF), linear integer arithmetic (LIA), bit-vectors (BV), and arrays with extensionality (AX).

### 3.1 Checker

SMTCoq's checker checks the deep embedding of SMT formulas in Coq against the proof trees constructed by the SMT solver. Recall that a deep embedding is an inductive type written in Coq representing SMT formulas. In the following, we present a simplified version of SMTCoq's deep embedding. The actual embedding uses Coq's machine integers to implement sharing of terms and optimize space, but an unoptimized and simpler version is presented below.

`form` is the type of SMT formulas:

```
Inductive form : Type :=
  | Fatom (_ : atom)
  | Ftrue
  | Ffalse
  | Fnot (_ : form)
  | Fand (_ : array form)
  | For (_ : array form)
  | Fimp (_ : array form)
  | Fxor (_ _ : form)
  | Fiff (_ _ : form)
  | Fite (_ _ _ : form)
```

where a formula can be an atom, `true`, `false`; a negation of a formula; a conjunction, disjunction, or implication of any number of formulas; an exclusive-or or an equivalence of two formulas; or, an if-then-else of three formulas.

`atom` is the type of SMT atoms (or terms):

```
Inductive atom : Type :=
  | AppIntr (_ : op) (_: list atom)
  | AppUnintr (_ : int) (_: list atom).
```

which can be applications of either interpreted functions (from a theory) or uninterpreted functions to zero (constants) or more atoms. Uninterpreted functions are parameterized by machine integers while interpreted ones are encoded as follows:

```
Inductive op : Type :=
  | Zcst (_ : Z) | Zle | Zlt | Zplus | ...
  | Eq (_ : type).
```

Some of the interpreted integer functions and the equality function are shown in `op`. Similarly, there are interpreted functions from each theory.

**Example 3.4.** Some examples of atoms are:

- `AppUnintr 1 []` for $x$, where $x$ has index 1.
- `AppUnintr 2 []` for $y$, with index 2 for $y$.
- `AppUnintr 0 [AppUnintr 1 []]` for the term $f(x)$ where $f$ has index 0.
- `AppIntr Zplus [(AppIntr (Zcst 5) []); (AppUnintr 2 [])]` for the term $5 + y$.

□

For convenience of representing CNF formulas in SMTCoq, a `clause` data structure encodes a disjunction of literals as a list of its disjuncts and a `state` represents the entire formula in CNF as an array of the respective conjuncts. We can now specify the deep embedding of the formula from our running example in Figure 1.

**Example 3.5.** The deep embedding of

$$B : (b \wedge \neg c) \vee (a \wedge \neg b) \vee \neg a \vee c$$

from our running example is `f`:

```
For {(Fand {Fatom (AppUnintr 1 []), Fnot (Fatom (AppUnintr 2 []))}),
     (Fand {Fatom (AppUnintr 0 []), Fnot (Fatom (AppUnintr 1 []))}),
     Fnot (Fatom (AppUnintr 0 [])), Fatom (AppUnintr 2 [])}
```

and the deep embedding of $\neg B$ (in CNF), $\neg$`f` is the encoding of the set $\{\neg b \vee c, \neg a \vee b, a, \neg c\}$ from Step 3.2 of Figure 2 as the following `state`:

```
{[Fnot Fatom (AppUnintr 1 []); Fatom (AppUnintr 2 [])],        (* ¬b ∨ c *)
 [Fnot Fatom (AppUnintr 0 []); Fatom (AppUnintr 1 [])],        (* ¬a ∨ b *)
 [Fatom (AppUnintr 0 [])], [Fnot Fatom (AppUnintr 2 [])]}      (* a, ¬c *)
```

where $a, b$, and $c$ are indexed as $0, 1$, and 2. □

Given deep embedding `f` of type `form` and proof object `t` of type `T`, the checker returns `true` if it is able to check that `t` is a proof of unsatisfiability of $\neg$`f`, and `false` otherwise.

```
checker :  form → T → bool
```

It is sound — when it returns `true`, the input formula is unsatisfiable, but not complete — when it returns `false` the input formula is not necessarily satisfiable.

**Proof Certificates** Although all SMT solvers (that interface with SMTCoq) provide a refutation tree as a proof as illustrated in Step 3.3 of Figure 2, their proof formats vary. CVC4 uses LFSC [15] which is a meta-logic that allows the specification of proof calculi that have declarative rules with computational components called side conditions, and veriT uses a proof language similar to SMT-LIB [4]. SMTCoq has its own proof certificate format for SMT solver proofs and it translates proofs from all solvers to this certificate format. `T` is the type of these certificates that are checked by the checker. Note that these translators are outside Coq's trusted kernel, but are kept small, and tested using a large set of benchmarks. Moreover, they don't reduce SMTCoq's soundness guarantees since a mistranslated proof will not check.

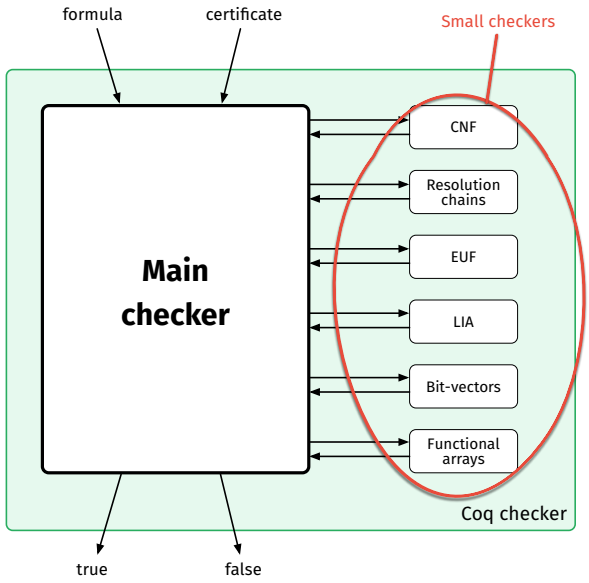


Figure 3: SMTCoq's `checker` shown as the "Main checker" which interacts with various small checkers.

**Checker Architecture** To prove the unsatisfiability of a formula, the SMT solver operates in phases. It converts the formula into CNF and performs some simplications on it. It then tries to satisfy the formula propositionally, by having a SAT solver run on what can be considered a propositional abstraction of the theory literals. It does this in a cycle where the theory solvers give feedback by adding the negation of the satisfying model to the abstraction of the formula, if the refinement of the model isn't satisfied in a theory. Each of these phases constitute *steps* of the SMT proof that SMTCoq has to check. A step can modify the current `state` (representation of the formula) while maintaining its (un)satisfiability. If the initial formula is unsatisfiable, then after running all steps, the final `state` is expected to be the empty clause $\bot$. Currently, SMTCoq has steps that perform resolution, conversion of formulas to conjunction normal form (CNF), SMT solver simplifications, and one for each theory. Each step is independently checked by a small checker. Each small checker simplifies the `state` via the step it performs, and the main checker (`checker`) ultimately checks that the input is finally transformed to $\bot$. The architecture of the checker is illustrated in Figure 3. The phases of SMT solver's operation and the corresponding steps of the SMT solver's proof are elaborated in Report 3.

## 3.2 Reflection Lemma

The previous section describes a deep embedding of SMT formulas in Coq and a checker for refutation proofs of their negations (Step 4 of the running example in Figure 1). This infrastructure alone serves as an external checker for SMT solver proofs in Coq. SMTCoq, however, is able to do more than this, as demonstrated in Step 5 and Step 6. In addition to checking SMT solver proofs, it can lift these up to Boolean formulas in Coq and prove such formulas automatically in Coq with the assistance of the SMT solver. In principle, this is done by proving the correctness of `checker`

with respect to `bool` formulas, that is to say that for every formula (`form`) that `checker` is able to successfully check against its proof from the SMT solver, the corresponding `bool` formula holds. The entities that these deeply embedded formulas must correspond to are the shallow embeddings of the formulas, and the correspondence itself is defined by the interpretation function. The shallow embedding, interpretation function, and the correctness lemma are discussed in what follows.

1. The *shallow embedding* of SMT formulas is Coq's `bool` type of Booleans. The advantage of a shallow embedding in a logical framework is that the semantics of formulas in the embedding are defined by the framework, while this is done, for instance, using the interpretation function $\mathbb{I}$ for deep embedding `form`. As with the deep embedding, Boolean formulas may have atoms containing terms from the theories of equality, integers, bit-vectors, and arrays. Boolean equality and the Coq $\mathbb{Z}$ [7] type of integers are sufficient to encode the first two, while SMTCoq defines custom types for bit-vectors and arrays (and as a consequence, also the semantics of these types). The respective shallow embeddings of the deeply embedded terms from Example 3.4 are simply `x`, `y`, `f(x)` and `5 + y` with `f, x,` and `y` correctly typed.

2. The *interpretation function* has the following type.

$$\mathbb{I} : \texttt{form} \to \texttt{val} \to \texttt{bool}$$

   The interpretation function maps the `form f` to its shallow embedding, given a valuation $\nu$ of free variables in `f`. $\nu$ specifies the assignment of values to the free variables in `f`. We use shorthand $\mathbb{I}_\nu(\texttt{f})$ for $\mathbb{I}$ `f` $\nu$. The interpretation function is recursively defined and while it is not fully specified here, we illustrate it using an example below.

3. The *reflection lemma*, or the reflection principle, is a theorem in Coq, proving the checker correct in terms of the shallow embedding.

$$\texttt{checker\_correct} : \forall\ \texttt{(f : form) (t : T) checker f t = true} \to \forall\ (\nu : \texttt{val}),\ \mathbb{I}_\nu(\texttt{f}) = \texttt{true}$$

   If a checker is able to check some formula `f` in the deep embedding against some proof certificate `t` from the SMT solver (it checks that `t` proves the unsatisfiability of $\neg$`f`), then `checker_correct`, proves that the interpretation of `f` (in the shallow embedding) holds, for any valuation of the variables in `f`, thus proving `f` valid.

   It is called the reflection lemma, because it uses computational reflection to shift the burden of the proof to a computation — the computation done by the checker. Since `checker_correct` is proved for all formulas and certificates, it can be instantiated with any particular formula and certificate. The only thing it requires, to prove the interpretation of the formula, is that the checker returns `true`. Since SMT solver proofs are often large, this is a heavy computation. The reflection mechanism reduces proving to computations via Coq's reduction mechanism. Coq's Calculus of Inductive Constructions (CIC) is a $\lambda-$calculus that has a reduction mechanism for terms. The reduction rules form a strongly normalizing system. The calculus's *conversion rule* allows a term to have multiple types, as long as the types have the same normal forms. For instance, for some predicate $P$ over natural numbers, a term that has type $P(10)$ also has type $P(5 * 2)$ and $P(20 - 10)$. Due to the conversion

rule, computations can be used in Coq's reasoning and proof terms can be found simply by computing normal forms of types.

Given deep embedding `f` of a formula, and SMT proof certificate `t` of $\neg$`f`, the proof of $\forall\ \nu,\ \mathbb{I}_\nu(\texttt{f}) = \texttt{true}$ is simply:

```
checker_correct f t (refl_equal (checker f t) true)
```

where `refl_equal` (reflexivity of equality) is a tactic that forces the Coq type-checker to perform an equality check between `checker f t` and `true` by reduction. The claim is that the two terms are convertible to each other and since `true` is a simple term of type `bool`, Coq's reduction mechanism is invoked on `checker f t`, which usually consists of checking a large, non-trivial proof certificate. SMTCoq uses various optimizations to make these computations efficient. As mentioned in the previous section, the deep embedding presented there only conceptually resembles SMTCoq's deep embedding. The implementation of SMTCoq uses a non-recursive structure with hash-consing of terms and machine integers to store variables, among other optimization techniques. Similar optimizations are also applied to certificates. Since the reflection lemma is proved for general formulas, each invocation of it only varies in the deep embedding and proof certificate that it is instantiated with. This is a simple instantiation proof in Coq, the overhead being in reducing the call to `checker` for each formula and proof pairing to `true`.

**Example 3.6.** Recall from Example 3.5 that the deep embedding `f` of $B$ from Figure 2 is:

```
For {(Fand {Fatom (AppUnintr 1 []), Fnot (Fatom (AppUnintr 2 []))}),
     (Fand {Fatom (AppUnintr 0 []), Fnot (Fatom (AppUnintr 1 []))}),
     Fnot (Fatom (AppUnintr 0 [])), Fatom (AppUnintr 2 [])}
```

$\mathbb{I}_\nu$(`f`) is recursively defined by first defining the interpretation of all the atoms in `f`.

- $\mathbb{I}_\nu$(`Fatom (AppUnintr 0 [])`) $= \nu$(`a`)
- $\mathbb{I}_\nu$(`Fatom (AppUnintr 1 [])`) $= \nu$(`b`)
- $\mathbb{I}_\nu$(`Fatom (AppUnintr 2 [])`) $= \nu$(`c`)
- $\mathbb{I}_\nu$(`Fnot (Fatom (AppUnintr 1 []))`) $=$ `negb` ($\mathbb{I}_\nu$(`Fatom (AppUnintr 1 [])`)) $=$ `negb`($\nu$(`b`))
- $\mathbb{I}_\nu$(`Fnot (Fatom (AppUnintr 2 []))`) $=$ `negb` ($\mathbb{I}_\nu$(`Fatom (AppUnintr 2 [])`)) $=$ `negb`($\nu$(`c`))

We then need the interpretation of conjunctions and disjunctions:

- $\mathbb{I}_\nu$ `(Fand {`$\texttt{F}_1$`, ..., `$\texttt{F}_n$`})` $= \mathbb{I}_\nu(\texttt{F}_1)$ `&&` ... `&&` $\mathbb{I}_\nu(\texttt{F}_2)$
- $\mathbb{I}_\nu$ `(For {`$\texttt{F}_1$`, ..., `$\texttt{F}_n$`})` $= \mathbb{I}_\nu(\texttt{F}_1)$ `||` ... `||` $\mathbb{I}_\nu(\texttt{F}_2)$

where && is the infix notation for the conjunction operator over bools in Coq and || is the infix notation for the disjunction operator over bools. Finally, we can define the interpretation of the entire deep embedding of f:

$$\mathbb{I}_\nu(\texttt{f}) \;=\; (\nu(\texttt{b})\ \texttt{\&\&}\ \texttt{negb}\ \nu(\texttt{c}))\ \texttt{||}\ (\nu(\texttt{a})\ \texttt{\&\&}\ \texttt{negb}\ \nu(\texttt{b}))\ \texttt{||}\ \texttt{negb}\ \nu(\texttt{a})\ \texttt{||}\ \nu(\texttt{c})$$

Assuming an arbitrary valuation $\nu$ gives us exactly the shallow embedding of $B$, or the Boolean term B:

```
(b && negb c) || (a && negb b) || negb a || c
```

for arbitrary a, b, and c.

In Step 4, a call to the checker checks that certificate t proves the unsatisfiability of ¬f. This call is made when the reflection lemma is instantiated with f and t in Step 5, and the refl_equal tactic forces Coq to convert checker f t = true to true = true. If the call to the checker is convertible, that is, the checker is successful, then the reflection lemma produces the proof of $\forall\ \nu,\ \mathbb{I}_\nu(\texttt{f})$ which is the necessary proof term required to complete the proof (Step 6). For this example, the call to the checker is indeed successful. □

**Small Checker Correctness** The checker divides certificates into steps that can be independently checked by small checkers. Similarly, the correctness lemma of the checker also depends on the correctness lemma of each small checker. Recall that small checkers modify the state via steps that they perform and eventually reduce the state to ⊥ if the certificate indeed proves unsatisfiability. Thus, the correctness lemma of a small checker guarantees that it performs sound steps - that is, the step doesn't change the satisfiability or unsatisfiability of the state. It is straighforward to compose these proofs of correctness of the small checkers, into that of checker (although it is not straightforward to prove the small checkers correct).

**Modularity** The division of a proof into steps and the division of the main checker into small checkers for these steps contribute to a modular architecture where it is fairly straightforward to add new steps (like new theories) to SMTCoq without breaking the rest of the system — one need only extend the step type, write a checker for the step, and prove it correct. The certificate format for SMT proofs also makes it convenient to integrate a new SMT solver with SMTCoq — one need only write a translator from the proof format of the SMT solver to the certificate format. As a testament to the utility of SMTCoq's modularity, it's initial iteration had veriT as the only SMT solver integrated, and EUF and LIA as the only theories. It was extended with support for CVC4 and for the theories of bit-vectors and arrays [8].

## 4 Sledgehammer

Isabelle [14] is an LCF-style system that provides a meta-logic which can be instantiated with other logics. Isabelle/HOL [13], one of the most popular Isabelle instantiations, implements a classical higher-order logic.

Sledgehammer is an Isabelle/HOL component that uses external ATPs to enhance Isabelle/HOL with proof automation. Initially, these ATPs only included resolution provers [12]. The work by Bohme et al. [6] involved extending Sledgehammer to incorporate SMT solvers [3] and this work

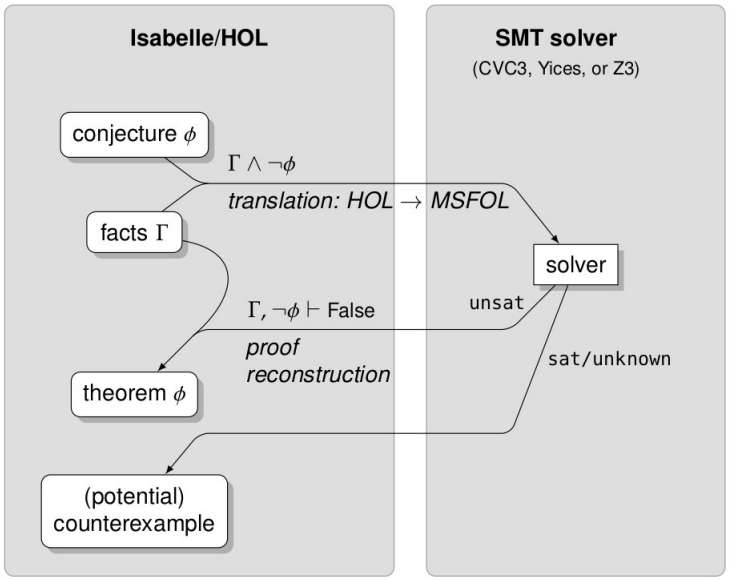


Figure 4: SMT solvers' integration with Isabelle/HOL

will be our focus for the rest of this section. As with SMTCoq, the SMT solvers produce a proof object which is then reconstructed within Isabelle/HOL by Sledgehammer, using Sledgehammer's internal ATP — Metis [10]. The proof object essentially guides the inference steps of the proof within Isabelle/HOL.

The integration of Sledgehammer with SMT solvers is illustrated in Figure 4. Given a conjecture $\phi$ in Isabelle/HOL, Sledgehammer selects a set of facts $\Gamma$ that might be relevant to proving $\phi$ and sends the formula $\Gamma \wedge \neg\phi$ to the SMT solver to check for unsatisfiability. If the SMT solver finds the formula to be satisfiable, it returns a satisfying model, which may serve as a counterexample of the conjecture. If the SMT solver is able to refute the conjecture (i.e., conclude the unsatisfiability of its negation), it returns a proof. The proof tree (that proves $\bot$ from the input) is then reconstructed in Isabelle/HOL via a bottom-up traversal of the tree, where for each node, Metis attempts to prove a corresponding theorem in Isabelle/HOL. Unlike SMTCoq, this reconstruction process occurs external to Isabelle/HOL's logic — an external procedure guides the reconstruction of nodes. However, a theorem is added to Isabelle/HOL by Metis, only if it is provable by Isabelle/HOL's LCF kernel. Thus, this process doesn't extend the trust base of the ITP. Since Isabelle/HOL implements a higher-order logic (HOL), which is more expressive than the SMT solver's many-sorted first order logic (MSFOL), the entirety of the formula $\Gamma \wedge \neg\phi$ may not be understandable to the SMT solver. Parts of the formula, however, may be fully first-order (FOL is a subset of HOL). Bohme et al. extended Sledgehammer with a translation from higher-order to first-order logic, so that it can reason about the remainder of the formula. This translation is discussed in the rest of this section.

### 4.1 Translation

Sledgehammer's SMT solver integration (Bohme et al.) performs a translation of HOL formulas to MSFOL formulas. $[\![\ ]\!]$ is the translation function that maps both HOL types and terms to MSFOL sorts and terms, respectively. The translation is sound — given a formula $F = \Gamma \to \phi$ in HOL, if $[\![F]\!]$ is refutable ($[\![\Gamma \wedge \neg\phi]\!]$ is unsatisfiable) in MSFOL, then $F$ is valid in HOL — but not complete — if $F$ is valid in HOL, $[\![F]\!]$ is not necessarily refutable in MSFOL. The translation might not carry over enough information about certain HOL formulas so that the their refutability can be concluded in MSFOL, which is expected, since HOL is a much more expressive logic than MSFOL.

Since MSFOL is a subset of HOL, we can see HOL as a composition of MSFOL-equivalent logic and the rest:

$$\text{HOL} = \text{MSFOL-equiv} + \text{non-MSFOL}$$

The MSFOL-equiv part of HOL has a straightforward transformation to MSFOL:

- Only nullary type constructors from HOL have equivalent MSFOL sorts:

$$[\![\kappa^0]\!] = \sigma$$

- Applications of Boolean connectives, quantifiers, equalities and other predicate symbols (function symbols returning `Bool`) to terms have corresponding MSFOL formulas:

$$\begin{aligned}
[\![False]\!] &\cong \bot \\
[\![\neg t]\!] &\cong \neg[\![t]\!] \\
[\![t \wedge u]\!] &\cong [\![t]\!] \wedge [\![u]\!] \\
[\![t = u]\!] &\cong [\![t]\!] = [\![u]\!] \\
[\![c^{\tau_1 \to ... \to \tau_n \to \texttt{Bool}} t_1 ... t_n]\!] &\cong c^{[\![\tau_1]\!],...,[\![\tau_n]\!]}([\![t_1]\!], ..., [\![t_n]\!]) \\
[\![\forall x^\tau . t]\!] &\cong (\forall x^{[\![\tau]\!]} . [\![t]\!])
\end{aligned}$$

- Variables and function applications (of return type other than `Bool`), including non-functional constants have MSFOL equivalents.

$$\begin{aligned}
[\![x^\tau]\!] &\cong x^{[\![\tau]\!]} \\
[\![c^{\tau_1 \to ... \to \tau_n \to \tau} t_1 ... t_n]\!] &\cong c^{([\![\tau_1]\!],...,[\![\tau_n]\!]) \to [\![\tau]\!]}([\![t_1]\!], ..., [\![t_n]\!])
\end{aligned}$$

The more interesting parts of the transformation deal with translating the non-first-order elements of HOL to MSFOL. These include:

- Type variables $\alpha$ and *compound types* — applications of type constructors $\kappa^n$ with $n > 0$ to types.
- $\lambda$-abstractions such as $\lambda x.t$.
- Variables of functional types and partial applications of functions to arguments. For example, $t^{\tau_1 \to \tau_2 \to \tau}\ t_1$ is a partial application of $t$ which is a variable of type $\tau_1 \to \tau_2 \to \tau$ to argument $t_1$ (of type $\tau_1$) and this application has type $\tau_2 \to \tau$. This is not directly expressible in MSFOL.

The following subsections describe phases of the translation that address the elimination of each of these non-first-order components of HOL.

### 4.1.1 Monomorphization

Recall that HOL types $\tau$ are either type variables $\alpha$ or applications of type constructors $\kappa^n$ to $n$ types. A *monomorphic type* is a type without type variables (e.g. $\kappa^0$, $\kappa^1\kappa^0$, and $\texttt{Bool}^0 \to \texttt{int}^0$ where $\to$ is a type constructor), and a *schematic type* is one with type variables (such as $\alpha \to \texttt{Bool}^0$). Monomorphization involves repeatedly instantiating schematic terms based on a set of monomorphic terms until a fixed point is reached.

The definition of monomorphization requires a description of *instantiation* of schematic entities w.r.t. monomorphic ones. Informally, a monomorphic type $\tau_M$ *matches* a schematic type $\tau_S$ if it or its components can replace the type variables in the schematic type. For example, `Bool` matches $\alpha$; and $\texttt{Bool} \to \texttt{int}$ matches $\texttt{Bool} \to \beta$ since `int` matches $\beta$. Finally, for our running example taken from Bohme et al., $\texttt{Bool} \to \kappa$ matches $\alpha \to \beta$, where $\kappa$ is a monomorphic type. The instantiations are defined recursively as follows.

- Given a monomorphic constant $c^{\tau_M}$, if $t^{\tau_S}$ is a schematic term that contains a schematic constant $c^{\tau_S}$ such that $\tau_M$ matches $\tau_S$, then $c^{\tau_M}$ induces a substitution $\sigma$ on $t^{\tau_S}$, and $\sigma(t^{\tau_S})$ is an instance of $t^{\tau_S}$ w.r.t. to $c^{\tau_M}$. For example, the instance of term

$$(\forall f^{\alpha\to\beta},\ x^{\alpha},\ xs^{\texttt{list}\ \alpha}.\ \texttt{apphd}\ f\ (\texttt{cons}\ x\ xs) = f\ x)$$

  w.r.t. constant

$$(\lambda x^{\texttt{Bool}}.\texttt{if}\ x\ \texttt{then}\ a^{\kappa}\ \texttt{else}\ b^{\kappa})^{\texttt{Bool}\to\kappa}$$

  is

$$(\forall f^{\texttt{Bool}\to\kappa},\ x^{\alpha},\ xs^{\texttt{list}\ \alpha}.\ \texttt{apphd}\ f\ (\texttt{cons}\ x\ xs) = f\ x)$$

  where the instantiation has turned all occurrences of $f$ in the term from $f^{\alpha\to\beta}$ to $f^{\texttt{Bool}\to\kappa}$. The following presents some inline, technical preliminaries about the rest of the terms. `list` $\alpha$ is a compound, polymorphic type of a list of $\alpha$'s where $\alpha$ is a type variable. `list Bool` is the type of lists of Booleans. `cons` is the list constructor of type $\alpha \to \texttt{list}\ \alpha \to \texttt{list}\ \alpha$ and [ ] is the list constructor for empty lists. For example, the integer list containing 1, 2, and 3 (in that order) is represented as `cons` 1 (`cons` 2 (`cons` 3 [ ])). `hd` is a `list` $\alpha \to \alpha$ function that returns the first element of a list and `apphd` is an $(\alpha \to \beta) \to \texttt{list}\ \alpha \to \beta$ function that takes a function $f$ and a list $l$ as input, and applies $f$ to the head of $l$ (or, $f$ (`hd` $l$)).

- The idea of an instance is extended to terms as follows. If $t^{\tau_M}$ is a monomorphic term, then $\sigma(t^{\tau_S})$ is an instance of $t^{\tau_S}$ w.r.t. $t^{\tau_M}$, if $\sigma$, the combination of all substitutions induced by the constants in $t^{\tau_M}$ is defined. This instance might still be schematic. For example, since $x^{\texttt{Bool}}$ is an instance of $x^{\alpha}$ w.r.t. $\texttt{T}^{\texttt{Bool}}$, and $xs^{\texttt{list Bool}}$ is an instance of $xs^{\texttt{list}\ \alpha}$ w.r.t. $[\ ]^{\texttt{list Bool}}$, we can combine the instantiation of the constant in the previous example to obtain that the instantiation of term

$$(\forall f^{\alpha\to\beta},\ x^{\alpha},\ xs^{\texttt{list}\ \alpha}.\ \texttt{apphd}\ f\ (\texttt{cons}\ x\ xs) = f\ x)$$

  w.r.t. the term

$$(\texttt{apphd}\ (\lambda x.\ \texttt{if}\ x\ \texttt{then}\ a\ \texttt{else}\ b)^{\texttt{Bool}\to\kappa}\ (\texttt{cons}\ \texttt{T}\ [\,]))\neq a$$

to be

$$(\forall f^{\texttt{Bool}\to\kappa},\ x^{\texttt{Bool}},\ xs^{\texttt{list Bool}}.\ \texttt{apphd}\ f\ (\texttt{cons}\ x\ xs)=f\ x)$$

- Instantiation can further be extended to sets of schematic terms $S$ and monomorphic terms $M$. An instance of $S$ w.r.t. $M$ is the set $I$ of terms such that each term in $I$ is an instance of some term from $S$ w.r.t. some term from $M$. Since instantiation can produce either monomorphic or schematic terms, $I$ can be partitioned into either as $(I_M, I_S)$.

Now, we can describe monomorphization. A *monomorphization step* for $S$ w.r.t. $M$ maps the pair $(M, S)$ to the pair $(M \cup I_M, S \cup I_S)$. The complete monomorphization of $S$ w.r.t. $M$ is the computation of a least fixed point of monomorphization steps of $S$ w.r.t. $M$. Given a HOL formula $F$, if $(M, S)$ is the partition of its constituents into monomorphic and schematic terms, then monomorphization of $S$ w.r.t. $M$ yields pair $(M', S')$ and we call the conjunction of all terms in $M'$ the monomorphization of $F$. Thus, using the above examples as instantiation steps, we have that monomorphization translates the formula $F$

$$\begin{aligned}F : (\forall f^{\alpha\to\beta},\ x^{\alpha},\ xs^{\texttt{list}\ \alpha}.\ \texttt{apphd}\ f\ (\texttt{cons}\ x\ xs)=f\ x)\ \wedge\\ ((\texttt{apphd}\ (\lambda x.\ \texttt{if}\ x\ \texttt{then}\ a\ \texttt{else}\ b)^{\texttt{Bool}\to\kappa}\ (\texttt{cons}\ \texttt{T}\ [\,]))\neq a)\end{aligned}$$

to the formula $F'$

$$\begin{aligned}F' : (\forall f^{\texttt{Bool}\to\kappa},\ x^{\texttt{Bool}},\ xs^{\texttt{list Bool}}.\ \texttt{apphd}\ f\ (\texttt{cons}\ x\ xs)=f\ x)\ \wedge\\ ((\texttt{apphd}\ (\lambda x.\ \texttt{if}\ x\ \texttt{then}\ a\ \texttt{else}\ b)^{\texttt{Bool}\to\kappa}\ (\texttt{cons}\ \texttt{T}\ [\,]))\neq a).\end{aligned}$$

The following are some limitations of this process:

1. There have to be monomorphized terms in a formula, to guide the monomorphization process, so it seems like this process would fail with formulas that contain only schematic terms.

2. The monomorphization process could be non-terminating. In other words, the first component of the pair yielded by monomorphization — the set of monomorphic terms — could be infinite. For example, consider $S = \{c^{\alpha} \wedge c^{\kappa\ \alpha}\}$ and $M = \{c^{\kappa_0}\}$. $\kappa_0$ matches $\alpha$, so the instance $c^{\kappa_0} \wedge c^{\kappa\ \kappa_0}$ is added to $M$ after a monomorphization step. Now, $\kappa\ \kappa_0$ matches $\alpha$, so the instance $c^{\kappa\ \kappa_0} \wedge c^{\kappa\ \kappa\ \kappa_0}$ is added to $M$, and so on. Since the proof of a formula, if it existed, would be finite, most monomorphic terms would be irrelevant. Finding the finite subset of necessary monomorphic terms is undecidable [5], but Bohme et al. use heuristic methods to overapproximate this set. They limit the number of monomorphization steps and the number of monomorphic terms generated with the expectation that monomorphic terms that contribute to proofs are typically generated early in the monomorphization process.

3. The monomorphization process is described as a syntactic process. Semantic steps in the process aren't described, and if it doesn't indeed involve any semantic pruning of translations, the number of monomorphization steps might be impractically large. For example, from the formula in the running example above ($F$), $x^{\alpha}$ could match $a^{\kappa}$ and be instantiated to $x^{\kappa}$. While syntactically, this would check out, semantically, given that $f^{\alpha\to\beta}$ is instantiated to $f^{\texttt{Bool}\to\kappa}$, and that $f$ is applied to $x$, $x$ has to have type `Bool` and this can be achieved by matching it with $x^{\texttt{Bool}}$.

4. Monomorphization is incomplete and the instantiations are done heuristically. This means that if the SMT solver is not able to prove the monomorphized version of a formula, a different monomorphization could very well be provable. Thus, the monomorphization of a problem would have to be done smartly, and sometimes repeated multiple times to be successful.

### 4.1.2 Lambda-Lifting

$\lambda$-abstractions represent anonymous or unnamed functions, which aren't allowed in MSFOL. These abstractions are removed from HOL formulas by a $\lambda$-*lifting* process, which uses a fresh constant as a name for the abstraction and adds a quantified formula specifying it's behavior. Concretely,

$$[\![t[\lambda x^{\tau}.u]]\!] \cong (t[(\lambda x^{\tau}.u) \mapsto c] \wedge (\forall x^{\tau}.\ c\ x = u))$$

The notation $t[x]$ represents a term $t$ with a sub-term $x$, and $t[x \mapsto y]$ is the term obtained by substituting all occurrences of $x$ by $y$ in $t$. $c$ is specified in MSFOL as an uninterpreted function with sort $\tau \to \kappa$ where $\kappa$ is the sort of $u$. For instance, $(\lambda x^{\mathtt{Int}}, x + 1)\ 5 = 6$ is translated to $(c\ 5 = 6) \wedge (\forall x^{\mathtt{Int}}.\ c\ x = x + 1)$. $\lambda-$lifting faithfully preserves the semantics of the function, and thus it maintains soundness. Quantified formulas are usually a source of undecidability for SMT solvers, so these lifted formulas contribute to the incompleteness of the translation.

### 4.1.3 Explicit Applications

For higher-order constants occurring multiple times with a variable number of arguments in a problem, the minimal number of arguments used with the constant in the problem is considered as the arity of the constant, and any additional arguments are expressed explicitly, with the help of a function symbol $\mathtt{app}^{(\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_2}$ that is defined as:

$$\mathtt{app}\ \ t_1\ t_2 = t_1\ t_2$$

If $c$ is a constant that occurs with at least $n$ arguments, then occurrences of applications of $c$ are translated as:

$$[\![c\ t_1\ ...\ t_n\ u_1\ ...\ u_m]\!] \cong \mathtt{app}\ (...\ (\mathtt{app}\ \ (c\ t_1\ ...\ t_n)\ u_1)\ ...)\ u_m$$

where $m$ could be 0, in which case there are no applications of `app`. For example, if $f^{\mathtt{Int} \to \mathtt{Int}}$ occurs twice in a formula, once partially applied to no arguments as $f$, and once fully applied as $(f\ 0)$, then since the minimal number of arguments to $f$ is zero, we have:

$$[\![f]\!] \cong f$$
$$[\![f\ 0]\!] \cong \mathtt{app}\ \ f\ 0$$

`app` is a higher-order function symbol so both the function ($f$ in the above example) and `app` have to be encoded in the SMT solver as constructors. This can also be done using arrays which are essentially functional types which can be passed around, since arrays in SMT-LIB [2] (the standard for SMT solvers) can have any index and value types, and map elements of the index type to the value type. Bohme et al. don't specify what technique they use to encode the `app` symbol, but it is reasonable to guess that they do it using SMT-LIB arrays.

Using the `app` symbol for partially applied interpreted constants in MSFOL, such as logical connectives, would result in terms that aren't well-typed. Instead, these are $\eta-$expanded and then

$\lambda-$lifted, after converting the partially applied term to a fresh constant. For example, for the partially applied conjunction ($\wedge$, used in prefix notation here), a partial application is translated as follows.

$$\begin{aligned} [\![t[\wedge\ x]]\!] &\cong [\![t[\lambda y.\ \wedge\ x\ y]]\!] && (\eta - expansion) \\ &\cong t[c] \wedge (\forall y.\ c\ y = \wedge\ x\ y) && (\lambda - lifting) \end{aligned}$$

where $c$ is the fresh constant for $\wedge\ x$ in the translated term.

#### 4.1.4 Erasure of Compound Types

A type constructor $\kappa^n$ with $n > 0$ is applied to $n$ types $\tau_i$. After monomorphization, each $\tau_i$ is monomorphic, so $\kappa^n\ \tau_1\ ...\ \tau_n$ is a monomorphic, compound type. It is represented as a fresh nullary type constructor $\kappa_0^n$ with the same interpretation as $\kappa^n\ \tau_1\ ...\ \tau_n$, which can now be represented in MSFOL.

$$[\![\kappa^n\ \tau_1\ ...\ \tau_n]\!] \cong \kappa_0^n$$

Some examples are:

$$[\![\texttt{Bool} \to \kappa]\!] \cong \kappa_1$$
$$[\![\texttt{list Bool}]\!] \cong \kappa_2$$

At the time of writing of Bohme et al. (2012), SMT solvers didn't support sorts parameterized by other sorts, but as of SMT-LIB version 2.5 [1] (2015), a parametric sort exists in SMT-LIB-compliant solvers, and these could alternatively be used to encode compound types.

While this is a straightforward translation step syntactically, notice that SMT solvers need extra help via premise selection to solve constraints involving these types. Integers are axiomatized in the SMT solver as the theories of linear and non-linear integer arithmetic, and Isabelle/HOL integers are made to correspond to these in the translation, so when a formula containing integers is sent to the SMT solver, it can perform integer reasoning. However, SMT solvers have no axiomatization of, say, lists of Booleans. It is crucial that when `list Bool` is translated to $\kappa_2$, the premise selector of Sledgehammer selects the right lemmas about Boolean lists to give to the SMT solver so it can reason about them. So, in this step, completeness depends heavily on premise selection.

#### 4.1.5 Interpreted and Uninterpreted Constants

As mentioned in the previous section, certain types in Isabelle/HOL correspond to certian sorts in the SMT-LIB standard. Bohme et al. exploit the axiomatizations of these sorts by translating constants and functions of these types from HOL to their corresponding MSFOL equivalents - these include linear arithmetic functions over integers and reals and bit-vector functions. This allows the SMT solver to rely on the decision procedures for these theories and not just the premises provided by Sledgehammer, to prove the conjecture.

The rest of the constants are translated as uninterpreted constants, and this is a source of incompleteness — the SMT solver only knows as much about an uninterpreted constants as the additional constraints (premises) will tell it. If the premise selector of Sledgehammer doesn't capture the constraints needed to prove something about an uninterpreted constant, the SMT solver will not be able to prove it. Even with interpreted functions, if the functions belong to an undecidable theory, the SMT solver will not necessarily be able to prove it.

### 4.2 Integration of Solver Response

Recall that given a conjecture $\phi$, Sledgehammer selects a set of facts $\Gamma$ and sends the formula $\Gamma \wedge \neg\phi$ to the SMT solver, after translating it to fully first-order using the translation described above. Given $[\![\Gamma \wedge \neg\phi]\!]$, the SMT solver may return either 'sat', 'unsat', or 'unknown'. If the SMT solver finds $[\![\Gamma \wedge \neg\phi]\!]$ to be satisfiable (sat), it could return a satisfying model; either the original conjecture is valid in Isabelle/HOL and the translation failed to capture enough information about the conjecture to help the SMT solver conclude unsatisfiability, or, the conjecture is invalid, and the satisfying model returned by the SMT solver is a counterexample. An 'unknown' result from the SMT solver is inconclusive. If the SMT solver finds the translated formula to be unsatisfiable, then the conjecture is provable in Isabelle/HOL. There are 3 possible integrations of this result:

1. Trust the SMT solver as an oracle, and declare the conjecture to be proved in Isabelle/HOL. This is not ideal since it extends the trust-base of Isabelle/HOL.

2. Use the SMT solver as a relevance filter by asking the SMT solver for an unsat-core — a subset of $\Gamma$ that by itself entails $\phi$. This would help Metis use a smaller set of premises when it proves the conjecture in Isabelle/HOL.

3. Reconstruct the proof from the SMT solver in Isabelle/HOL by having Metis replay the inferences of the proof tree.

Bohme et al. use the third integration with proofs from the Z3 SMT solver, where for every inference node in the proof tree from Z3, Metis tries to prove a theorem in Isabelle/HOL.

## References


[1] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available at `www.SMT-LIB.org`.

[2] C. Barrett, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[3] C. Barrett and C. Tinelli. *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing, Cham, 2018.

[4] F. Besson, P. Fontaine, and L. Thery. A flexible proof format for SMT: a proposal. 2011.

[5] F. Bobot and A. Paskevich. Expressing polymorphic types in a many-sorted language. In C. Tinelli and V. Sofronie-Stokkermans, editors, *Frontiers of Combining Systems*, pages 87–102, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[6] S. Böhme. *Proving Theorems of Higher-Order Logic with SMT Solvers*. PhD thesis, Technische Universität München, 2012.

[7] P. Crégut. The Coq Z Library. `coq.inria.fr/library/Coq.Numbers.BinNums.html`.

[8] B. Ekici, G. Katz, C. Keller, A. Mebsout, A. J. Reynolds, and C. Tinelli. Extending smtcoq, a certified checker for SMT (extended abstract). In J. C. Blanchette and C. Kaliszyk, editors, *Proceedings First International Workshop on Hammers for Type Theories, HaTT@IJCAR 2016, Coimbra, Portugal, July 1, 2016*, volume 210 of *EPTCS*, pages 21–29, 2016.

[9] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, USA, 1993.

[10] J. Hurd. First-order proof tactics in higher-order logic theorem provers. pages 56–68.

[11] C. Keller. *A Matter of Trust: Skeptical Communication Between Coq and External Provers. (Question de confiance : communication sceptique entre Coq et des prouveurs externes)*. PhD thesis, École Polytechnique, Palaiseau, France, 2013.

[12] L. Kovács and A. Voronkov. First-order theorem proving and vampire. In N. Sharygina and H. Veith, editors, *Computer Aided Verification*, pages 1–35, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[13] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[14] L. C. Paulson. Isabelle: The next 700 theorem provers. *CoRR*, cs.LO/9301106, 1993.

[15] A. Stump, D. Oe, A. Reynolds, L. Hadarean, and C. Tinelli. SMT proof checking using a logical framework. *Formal Methods Syst. Des.*, 42(1):91–118, 2013.