Proofs and reconstructions

Nik Sultana[1], Christoph Benzmüller[2], and Lawrence C. Paulson[1]

[1] Computer Lab, Cambridge University
[2] Dept of Mathematics and Computer Science, Freie Universität Berlin

Abstract. Implementing proof reconstruction is difficult because it involves symbolic manipulations of formal objects whose representation varies between different systems. It requires significant knowledge of the source and target systems. One cannot simply re-target to another logic. We present a modular proof reconstruction system with separate components, specifying their behaviour and describing how they interact. This system is demonstrated and evaluated through an implementation to reconstruct proofs generated by Leo-II and Satallax in Isabelle/HOL, and is shown to work better than the current method of rediscovering proofs using a select set of provers.

Keywords: proof reconstruction, higher-order logic, abstract machines

1 Introduction

The case for interfacing logic tools together has been made countless times in the literature, but it is still an important research question. There are various logics and tools for carrying out formal developments, but practitioners still lament the difficulty of reliably exchanging mathematical data between tools.

Writing proof-translation tools is hard. The problem has both a theoretical side (to ensure that the translation is adequate) and a practical side (to ensure that the translation is feasible and usable). Moreover, the source and target proof formats might be less documented than desired (or even necessary), and this adds a dash of reverse-engineering to what should be a system integration task.

We suggest that writing such tools can be made easier by relying on a suitable modular framework. Modularity can be used to *isolate* the translation of different kinds of formulas, inferences, and logics from one another. This has significant practical benefits. First, the translations can be developed separately. Second, if the reconstruction of an inference fails, it does not affect the reconstruction of other inferences in the proof. This makes it easier to localise debugging efforts. Third, it improves usability. The diversity between proof systems means that inference-specific code can hardly ever be reused to reconstruct proofs from other theorem provers. Thus, proof reconstruction is difficult to scale to reconstruct proofs from different systems. The framework carves out the functionality that *can* be reused between systems. This code is often fairly general, and used to store and query formulas, inferences, and their metadata. We believe that

this divide-and-conquer approach is crucial to ease the implementation of proof reconstruction for different systems.

In this paper we propose a framework structured as a compiler. The compiler's target is specified as an abstract proof-building machine, which captures essential features of the target logic. This framework is designed to be efficient and extensible. Both compiler and abstract machine are implemented as an extension of the Isabelle/HOL proof assistant [14], to import proofs produced by the Leo-II [4] and Satallax [7] theorem provers.

Paper structure. The series of functions applied to a proof in our framework is outlined in §2. Our abstract model of a proof translator is described in §3, before returning to describe our framework in more detail in §4. Our implementation is described and evaluated in §5, before contrasting with related work in §6. We conclude in §7 with a description of what we learned from this project.

2 Reconstruction workflow

Proof reconstruction consists of a series of steps, or workflow, applied to some representation of a proof. As a result of this workflow, a proof in a *source* logic is transformed into a proof (of the same theorem) in the *target* logic.

Before giving a detailed description of the workflow in later sections, we summarise our framework by outlining what needs to be implemented at each step of the translation. If an implementation of the framework already exists, then this description summarises what needs to be added or changed to translate proofs between different theorem provers.

1. **Parse the proof.**
2. **Interpret the logical signature.** We use a mapping from *types* and *constants* of the source logic, into types and constants of the target logic. This mapping might not be total if the source language cannot be fully interpreted in the target language. This mapping is lifted to map terms and formulas from the source to target. If one of the logics is not typed, then suitable encodings could be used [5].
3. **Analyse and transform the proof.** We often want to change the representation of the proof before translating it, to remove redundant information, or restore information that was not included when the proof's representation was produced.
4. **Generate a trace.** We linearise the proof into a series of inferences. These inferences are changed into instructions to an abstract proof-building machine, which we describe in the next section.
5. **Emulate inferences.** There are two kinds of *interpretations* at play when translating proofs. The first kind was encountered in step 2, when we interpreted expressions, mapping them from the source to the target language. The second kind of interpretation, which we call *emulation*, involves interpreting *inferences* of formulas, from the source to the target logic. As a result of emulation, we generate a finite set of admissible rules in the target logic.

This set forms a calculus that will be complete for the purpose of translating the source proof into the target logic.

6. **Play the trace.** This is done on the abstract machine, and supported by the emulated rules, to yield a proof in the target logic.

To use our framework, one must implement each of these steps. To handle a new source language we must change steps 1-5. Step 6 provides an interface to the target logic, in the form of an abstract proof-building machine. This machine is an intermediate target in our framework, between the source and target logics. We describe this machine next before describing the rest of the workflow in detail.

3 Cut machines

The key observation of our approach is that while proof search abhors the cut rule [2], proof translation benefits from it. We describe a simple abstract machine for mapping proofs from one logic into another. It serves as an abstract model of proof translation. A similar method was used by de Nivelle [10] to describe the generation of proof terms that validate clausifications. It is also inspired by how generic proof checking is done in Isabelle [15].

The purpose of using such a machine in our framework is to isolate the source and target logics. We believe that this will make it easier to modify or repurpose the front-end and back-end of implementations of the framework, to reconstruct proofs from, or to, different logics. Such modifications would not affect other parts of the framework; this is inspired by how compilers are structured.

A *cut machine* is defined in terms of two features:

1. The machine's **state** consists of a tuple $(\rho,\ \sigma,\ F)$, where ρ is a finite set of ground assumption sequents (that can include axioms of a particular theory). Symbol σ represents a stack of proof *subgoal* sequents, and F is the goal formula. Proving all the subgoals is sufficient for proving the goal. The proofs presented to the cut machine are translated to the target system. Translating the proofs of all the subgoals is sufficient for translating the proof of the goal.

2. **Instructions** given to the machine may consist of the following:
   - 'PROVE F' states that F is the goal formula. A goal formula may only be set once per proof.
   - 'CUT r' applies the sequent $r \in \rho$ to the stack of subgoals in the machine's state. This will be described in more detail below.
   - 'END' asserts that a machine is in a terminal state. A terminal state is one where the subgoal stack σ is empty. The goal formula F in that state has been shown to follow from ρ using the instructions presented to the machine, which result in a proof in the target logic.

We call them 'cut machines' because they mainly rely on applying instances of the Cut rule to splice together inferences. These are inferences in the target logic that emulate the inferences made in the source logic proof. Splicing together

the emulated inferences produces a proof in the target logic. Specifically, consider the instruction 'CUT r', where $r \in \rho$ is a sequent, such that $r = \frac{A_1,\ldots,A_n}{B}$. (We overload the rule notation to express sequents, since the resulting notation is more pleasant to read. We use the symbol $\vdash_\rho$ to denote the finite proof system contained in ρ.) Then 'CUT r' can be interpreted as the following rule:

$$\frac{\vdash_\rho \frac{A_1,\ldots,A_n}{B} \qquad \vdash_\rho A_1 \qquad \ldots \qquad \vdash_\rho A_n}{\vdash_\rho B}$$

Let the symbol $\rhd$ represent the single-step transition relation between states. We will use '$-$' to describe an empty stack, and right-associative ':' to describe the push operation. The formal semantics of the machine's instructions are as follows:

$$\textsc{Prove}\ F: \quad (\rho,\ -,\ \mathsf{True}) \rhd (\rho,\ F,\ F)$$

$$\textsc{Cut}\ r: \quad (\rho,\ B:\sigma,\ F) \rhd (\rho,\ A_1:\ldots:A_n:\sigma,\ F) \quad \text{where } r \in \rho \text{ and } r = \frac{A_1,\ldots,A_n}{B}$$

$$\textsc{End}: \quad (\rho,\ -,\ F) \rhd (\rho,\ -,\ F).$$

A *cut program* consists of a finite sequence of instructions. A *well-formed* cut program consists of a single PROVE instruction, zero or more CUT instructions, and finally a single END. An *initial state* consists of any state of the form $(\rho,\ -,\ \mathsf{True})$. A *terminal state* consists of any state of the form $(\rho,\ -,\ F)$.

A cut program describes the proof of some statement $\vdash_\rho F$ in the source logic, and the cut machine uses this description to build a proof in the target logic. Note that a cut program without a PROVE instruction only describes the tautology $\vdash_\rho \mathsf{True}$. A cut machine running a well-formed cut program can get stuck in two ways: (i) when executing 'CUT r' if $r \notin \rho$ or if the conclusion of r does not match the top element in σ, or (ii) when executing END if the machine is not in a terminal state.

3.1 Validating the model

Use of the model relies on the assumption that ρ contains all the rules needed by the cut program. The finite set ρ contains a restricted inference system, consisting of inference rules in the target logic. This set is a parameter to the model, and the generation of these rules takes place externally—this will involve emulating the inference rules of the source logic in terms of the target logic, as described in §4.4.

A cut program that does not get stuck is called *well-going*. Provided that a suitable ρ exists, the model has the following properties. Provided it is given a well-going cut program, the cut machine has the following invariant: if the subgoals are valid, then the goal is valid too. We can also show that if a cut program reaches a terminal state then its proof goal is valid. Thus a well-going cut program always produces a theorem in the target logic. Moreover, this can be verified by inspecting a proof in the target logic.

3.2 Using the model

This section will describe how this model interacts with the workflow described in §2. Let L_1 represent a source logic, and L_2 represent a target logic. 'Logic' here is used to describe essentially the syntactical features of a logic: the syntax of its formulas, and the formation rules of its proofs. To use the model we require three functions:

1. A mapping from formulas of L_1 into formulas of L_2, such that semantics is preserved. We rely on the interpretation of formulas for this, mentioned in point 2 in §2, and described further in §4.3.
2. A mapping from *inferences* in the source proof to inferences in the target logic. We call this mapping an *emulation*. This was mentioned in point 5 in §2, and will be described further in §4.4.
   The resulting inferences are not necessarily primitive inferences—they could be admissible rules. These rules make up the contents of ρ, one of the parameters of the machine described in this section.
3. A compiler that takes proofs encoded in L_1 and produces a cut program. This was mentioned in point 4 in §2, and will be described further in §4.5, which includes example output of such a compiler.

If the functions above are total and preserve semantic properties, then any proof in the source logic can be translated into a proof in the target logic. The translation can be carried out by running the cut program on an implementation of the cut machine.

3.3 Extending the model

Reliance on the cut rule gives this framework its generality. A cut machine can be specialised by lifting features of the source logic to the level of the machine. This involves extending the definition of the machine and its instruction set. The lifted feature would then be simulated at the machine level, like the Cut instruction, rather than relying on opaque derivations in ρ.

This can be useful for features such as *splitting* [21]. Recall that splitting is a rule scheme used in clausal calculi to make clauses smaller. We will base the description of splitting on the implementation of this concept in Leo-II. Without loss of generality, we will look at an example starting with a binary clause $\{A, B\}$ such that A and B do not share free variables. We can split this clause into singleton clauses $\{A\}$ and $\{B\}$, but separate refutations must be obtained for each element of the split—that is, $\{A\}$ cannot be used in a refutation derived from $\{B\}$, and vice-versa.

Using the current definition of the machine, such a rule could be used outside the machine to populate ρ (remember that ρ is a parameter to the model) with the rule $\frac{A \vee B}{\mathsf{False}}$. We would then use this rule via Cut as before.

Instead, we could modify the machine's definition to *lift* the rule to the machine level, to specialise the machine to support splitting.

Logically, this is the following rule:

$$\frac{\vdash_\rho \frac{A}{\mathsf{False}} \qquad \vdash_\rho \frac{B}{\mathsf{False}} \qquad \vdash_\rho A \vee B}{\vdash_\rho \mathsf{False}}$$

The semantics of the new instruction $\textsc{Split}(A \vee B)$ is:

$$\left(\rho,\ \frac{C}{\mathsf{False}} : \sigma,\ F\right) \quad \triangleright \quad \left(\rho,\ \frac{C \wedge A}{\mathsf{False}} : \frac{C \wedge B}{\mathsf{False}} : \frac{C}{A \vee B} : \sigma,\ F\right)$$

Such a machine has been implemented for interpreting Leo-II proofs in Isabelle/HOL. Interpreting Satallax proofs only relies on the basic machine, without splitting.

4 Framework

Our approach to proof reconstruction is made up of two phases: the *shunting* and *emulation* of inferences. The first phase (steps 1-4 in §2) transforms a proof and generates a cut program, while the second phase (step 5 in §2) assists in the execution of this program. The second phase populates the set ρ that will be used when executing the cut program (step 6 in §2). Executing the cut program will yield a proof in the target logic.

The two phases are related, but have different purposes:

- The *shunting* of inferences involves (globally) meaning-preserving transformations being applied to a proof, to facilitate its reconstruction.
- Emulation maps inferences of one calculus to chains of inferences in another calculus. In our implementation, the inferences made by Leo-II and Satallax are emulated as Isabelle/HOL-admissible rules.

It is advantageous to separate the two phases since some details of one can be encapsulated from the other. Furthermore, the emulation of each inference rule takes place independently of the others. Failure to reconstruct an inference will mean that we cannot reconstruct the entire proof, but would *not* affect the reconstruction of other inferences in the proof. This isolation in emulation is advantageous since it localises debugging, and could allow humans to assist in reconstructing inference rules that currently cannot be emulated by the implementation.

We will concretise our description of the framework to a specific proof being translated between two specific logics: from the classical higher-order logic clausal calculus of Leo-II to the classical higher-order logic of Isabelle/HOL. Despite their conceptual similarity, non-trivial manipulation is required to have the proofs of Leo-II checked by Isabelle/HOL: some information needs to be pruned away, and other information reconstructed, as will be explained below. Despite the specificity of this explanation, this method is applicable to other varieties of formal logic, such as the higher-order tableau calculus used by Satallax.

For a running example, let us take the TPTP problem SEU553ˆ2. In this problem, we use individuals, whom we represent by the type symbol ι, to model sets of elements. The powerset function therefore has the type $\iota \to \iota$. The problem conjectures that if two arbitrary sets, A and B, containing individuals, are equal, then their powersets are equal too. This is formalised as follows:

$$\forall A : \iota, B : \iota.\ A = B \ \longrightarrow\ \mathsf{powerset}\, A \ = \ \mathsf{powerset}\, B$$

Leo-II proves this to be a theorem. Its proof output is shown in Figure 1, and rendered as a graph in Figure 2.

4.1 Proof generation

Böhme and Weber [6] recommend that proofs intended for reconstruction should be sufficiently detailed to facilitate this task. We came to appreciate the validity of their advice based on our experience with different versions of Leo-II's proof output. By default, Leo-II proofs may contain instances of *compound rules*, such as those for clausification and unification. Using compound rules often results in shorter proofs since the details of member inferences are elided. Unfortunately, this loses information that can be very expensive to recompute. This is described in more detail and quantified in the first author's dissertation [19]. Fortunately Leo-II can be instructed to expand compound rules into primitive inferences in its proof output. We found this to be essential. Satallax does not use compound rules in its proofs.

4.2 Formula interpretation

After the proof is parsed, its logical signature—consisting of a set of types, and a set of constants—is extracted. The signature is interpreted in the target logic—in this case, it consists of the types ι and $\iota \to \iota$, and the constants $\mathsf{powerset}$, $\mathsf{sK1}_{\mathrm{A}}$, and $\mathsf{sK2}_{\mathrm{SY2}}$, following the signature described on lines 1-3 in the proof shown in Figure 1. The constants $\mathsf{sK1}_{\mathrm{A}}$ and $\mathsf{sK2}_{\mathrm{SY2}}$ do not appear in the problem's formulation, because they are Skolem constants [11], and they are scoped in the proof, not in the original problem.

After interpreting the signature, the formulas contained in inferences (lines 4-29) are interpreted in the target logic relative to this signature. This leaves us with a skeleton of the proof consisting of the inferred formulas, but so far does not include the inferences themselves, other than metadata—such as the names of inference rules, and their parameters (e.g., which inferences they derive from).

This step is identical for both Satallax and Leo-II proofs encoded in TPTP. The approach will differ significantly for the two provers in the next steps, before converging again when the cut programs (resulting from their proofs) are executed.

```
1    thf( tp_powerset , type, (powerset: ($i>$i))).
2    thf( tp_sK1_A , type, ( sK1_A : $i)).
3    thf( tp_sK2_SY2 , type, ( sK2_SY2 : $i)).
4    thf( 1 , conjecture, (![A:$i,B:$i]: ((A = B) => ((powerset@A) = (powerset@B)))),
5  ★   file('SEU553^2.p',powerset__Cog)).
6    thf( 2 , negated_conjecture, ((![A:$i,B:$i]: ((A = B) => ((powerset@A) = (powerset@B))))=$false)),
7  ★   inference(negate_conjecture,[status(cth)],[1])).
8    thf( 3 , plain, (((![A:$i,B:$i]: ((A = B) => ((powerset@A) = (powerset@B))))=$false)),
9  ★   inference(unfold_def,[status(thm)],[2])).
10   thf( 4 , plain, (((![SY2:$i]: (( sK1_A = SY2) => ((powerset@ sK1_A ) = (powerset@SY2))))=$false)),
11 ★   inference(extcnf_forall_neg,[status(esa)],[3])).
12   thf( 5 , plain, ((((( sK1_A = sK2_SY2 ) => ((powerset@ sK1_A ) = (powerset@ sK2_SY2 )))=$false)),
13 ★   inference(extcnf_forall_neg,[status(esa)],[4])).
14   thf( 6 , plain, (((( sK1_A = sK2_SY2 )=$true)),
15 ★   inference(standard_cnf,[status(thm)],[5])).
16   thf( 7 , plain, ((((powerset@ sK1_A ) = (powerset@ sK2_SY2 ))=$false)),
17 ★   inference(standard_cnf,[status(thm)],[5])).
18   thf( 8 , plain, (((~ ((powerset@ sK1_A ) = (powerset@ sK2_SY2 )))=$true)),
19 ★   inference(polarity_switch,[status(thm)],[7])).
20   thf( 9 , plain, ((( sK1_A = sK2_SY2 )=$true)),
21 ★   inference(clause_copy,[status(thm)],[6])).
22   thf( 10 , plain, (((~ ((powerset@ sK1_A ) = (powerset@ sK2_SY2 )))=$true)),
23 ★   inference(clause_copy,[status(thm)],[8])).
24   thf( 11 , plain, ((((powerset@ sK1_A ) = (powerset@ sK2_SY2 ))=$false)),
25 ★   inference(extcnf_not_pos,[status(thm)],[10])).
26   thf( 12 , plain, (($false)=$true)),
27 ★   inference(fo_atp_e,[status(thm)],[9,11])).
28   thf( 13 , plain, ($false),
29 ★   inference(solved_all_splits,[solved_all_splits(join,[])],[12])).
```

Fig. 1. Leo-II's proof of SEU553^2. Leo-II encodes its proofs in TPTP format [20], where each line is structured as follows: `language(id, role, fmla, annotation)`, where the annotation is optional. Here the language is 'THF' [3], used to encode formulas in higher-order logic. Some information in the proof has been marked up: the grey-boxed text, such as `tp_powerset` and `1`, are unique identifiers for inference steps; and the boxed text, consisting of `sK1_A` and `sK2_SY2` are Skolem constants. Lines prefixed by a star ★ are annotation lines, and the <u>underlined</u> words in those lines are names of inference rules used by Leo-II.

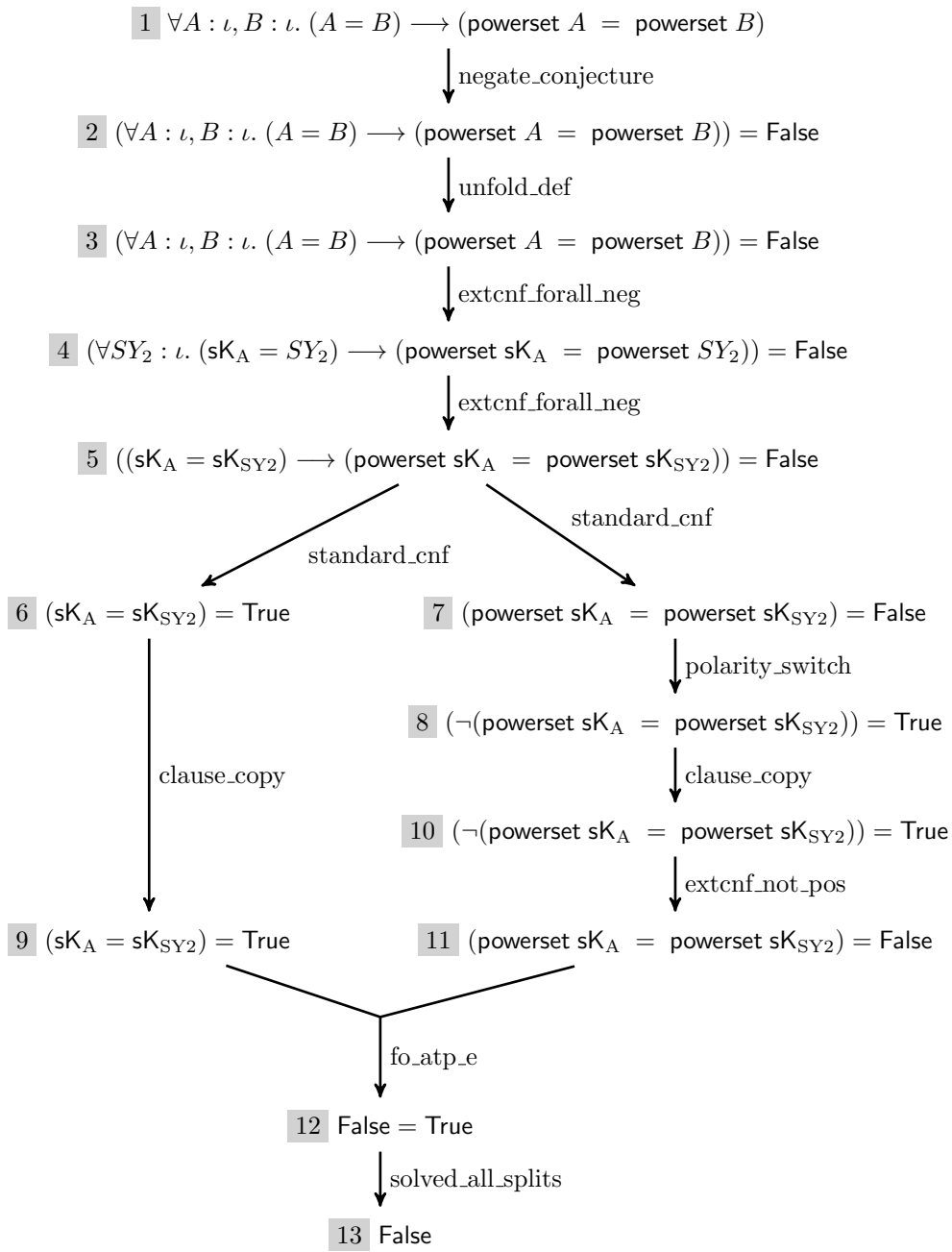

Fig. 2. Graph for Leo-II's proof of SEU553ˆ2. Vertices consist of formulas derived during the proof, except for the topmost formula, labelled 1, which is obtain from the problem's formulation. The numeric labels adjacent to formulas, such as 1, are used by Leo-II to uniquely identify clauses it generates. We use these labels to index clauses during proof reconstruction. These labels correspond to the labels in Figure 1. Note that this proof contains inferences that do not materially advance the proof. We can see this between formulas 6 and 9, and between formulas 7-11. A simple static analysis could erase formulas 9 and 8-11, and adjust the edges from 6 and 7 to point to 12.

4.3 Proof analysis and transformation

Some preprocessing and transformation of inferences is carried out on the proof skeleton. The inferences form a directed acyclic graph where the vertices are formulas, and the edges are labelled with inference names. The proof from Figure 1 is shown as such a graph in Figure 2.

The analysis and transformation of the inferences might be done to simplify the proof, or to facilitate further analyses or transformations. There are three proof transformations that we found to be useful for processing Leo-II proofs:

1. *Eliminating redundant parts of the proof.* Occasionally Leo-II includes redundant chains of inferences that do not materially contribute to the proof. We can see two examples of this in Figure 2, as explained in its caption.
2. *Extracting subproofs related to splitting.* Recall from §3.3 that each subproof of a split yields a refutation, and the set of inferences made during a subproof of a split is kept apart from the inferences made in other splits. Each subproof is used to construct a lemma that produces a premise to the disjunction-elimination rule.
3. *Separating instantiation from other inferences.* Leo-II sometimes overloads inferences with instantiation of variables. This makes it harder to emulate an inference. Rather than emulating this complex behaviour, we transform the proof to extract instantiations into separate inferences. This allows us to handle instantiations and other inferences separately.

A useful transformation in Satallax consisted of inlining assumption formulas to produce the actual inferences made at each step. Satallax does not explicitly encode an inference formula. Instead, each proof line produced by Satallax refers to formulas involved in that inference, and the formulas are stored separately. Combining these to give the actual inference is straightforward.

A useful analysis to carry out on both Leo-II and Satallax proofs involves discovering the definitions of Skolem constants. These definitions are necessary to emulate their ∃-elimination inferences. Both provers' proofs generally contain the declarations of Skolem constants (as can be seen for Leo-II on lines 2 and 3 of Figure 1), but not their definitions. These definitions are implicit in the proof, and can be extracted by analysing the syntax. Skolemisation occurs on lines 4 and 5 in Figure 1, and the equations extracted by our analysis are:

$$\begin{aligned}
\mathsf{sK1}_{\mathrm{A}} &= (\varepsilon A : \iota.\ (\forall B : \iota.\ (A = B) \longrightarrow (\mathsf{powerset}\ A = \mathsf{powerset}\ B)) = \mathsf{False}) \\
\mathsf{sK2}_{\mathrm{B}} &= (\varepsilon B : \iota.\ (\mathsf{sK1}_{\mathrm{A}} = B) \longrightarrow (\mathsf{powerset}\ \mathsf{sK1}_{\mathrm{A}} = \mathsf{powerset}\ B) = \mathsf{False})
\end{aligned}$$

These are then added as axioms to the theory. One might feel justifiably squeamish about adding axioms to a theory, but the axioms concerned here are definitional axioms for Skolem constants.

Note that here we assume that the target theory validates the Axiom of Choice (and can interpret Hilbert's ε operator). This arises from the specific features of our implementation targetting Isabelle/HOL (which is a classical logic) and it is not a feature or requirement of the cut-machine model.

4.4 Emulation of inference rules

We now turn to the inferences themselves. Inferences are emulated to yield admissible rules in the target calculus. Emulation might be implemented using rule schemes or proof-building functions, which could consist of calls to provers whose output we can already reconstruct. This was done previously between Leo-II, Satallax and Isabelle/HOL [18]. For instance, the Leo-II inference described in line 26 of Figure 1 can be emulated by the Isabelle/HOL-admissible rule

$$\frac{(\mathsf{sK1}_{\mathrm{A}} = \mathsf{sK2}_{\mathrm{SY2}}) = \mathsf{True} \ \wedge \ (\mathsf{powerset}\ \mathsf{sK1}_{\mathrm{A}} = \mathsf{powerset}\ \mathsf{sK2}_{\mathrm{SY2}}) = \mathsf{False}}{\mathsf{False} = \mathsf{True}}$$

where, as specified in Figure 1, the first premise is contributed from the conclusion of the inference labelled 9 (occurring in line 20), and the second premise from the inference labelled 11 (line 24). The proof text also indicates that this inference was made using the E theorem prover [17], with which Leo-II cooperates. The resulting rule in Isabelle/HOL is labelled "12", consistent with the name used in the TPTP encoding of the proof. Once all inference rules have been emulated, then the proof skeleton has been extended to include all the information necessary to produce a proof in the target logic.

4.5 Generating a cut program

So far we have imported all of the proof information—consisting of signature, formulas, and inferences—from the source logic into the target logic. We now need to combine the inferences to reconstruct the theorem in the target logic. The proof graph is traversed depth-first, to produce a trace, or cut program, of the proof. Running this on an implementation of a cut machine (described in §3) should produce the reconstructed proof.

For the example proof above, the program consists of 18 instructions, essentially a traversal of Figure 2:

```
[Cut "13", Cut "12", Unconjoin,
 Cut "11", Cut "10", Cut "8", Cut "7",
   Cut "5", Cut "4", Cut "3", Assumed,
 Cut "9", Cut "6",
   Cut "5", Cut "4", Cut "3", Assumed,
 End]
```

4.6 Executing a cut program

The cut program is interpreted according to the semantics given in §3. We use some additional instructions in our implementation. We describe them next; they can both be seen in the example code snippet given above.

The *Unconjoin* instruction eliminates a conjunction, and behaves like the right-conjunction rule in the sequent calculus. This is needed since the proof graph is not a tree in general—it recombines. For example, this command is

applied to step 12, formalised in Isabelle/HOL in §4.4, to break up the conjoined premises into two premises. This step consumes one subgoal, and produces two subgoals. Each subgoal relates to a path to the root of the graph. The root of the graph consists of the conjecture formula. Finally, the *Assumed* instruction discharges a subgoal using an identical element of ρ.

Note that in the program shown earlier, lines 3 and 5 are duplicates. We could extract a lemma that fuses together the contents of line 3 into another admissible rule, then replace line 5 with an application of this lemma. There are different ways of implementing this. One way is a proof analysis that produces commands to create and use a lemma (in which case the machine needs to be extended to interpret these commands). Another is a proof transformation that adds the lemmas to ρ and simply produces a 'Cut' command to use the lemma.

For both Leo-II and Satallax we execute this program on the double-negated conjecture, since they both work by refutation. If all emulation steps are successful, then the execution should yield an Isabelle/HOL theorem corresponding to that proved by Leo-II or Satallax.

5 Implementation

We implemented this system as an extension to Isabelle, and it consists of around 8600 lines (including comments) of Standard ML and Isabelle definitions—such as the formalisation of Leo-II and Satallax inference rules as Isabelle rules.[3] The cut machine described in this paper is implemented as an interface to Isabelle's kernel; ultimately all the reconstructed proofs are validated by Isabelle's kernel.

The preceding sections described the design of the framework and its components, and how it was implemented to import Leo-II and Satallax proofs into Isabelle/HOL. These are the limitations of the implemented prototype:

1. Currently we do not handle the rules for the Axiom of Choice in both Leo-II and Satallax. Emulating those rules is a natural extension to this work.
2. Recall that Leo-II collaborates with E to find a refutation. In our implementation, E's proofs are re-found by using Metis [12], whose proofs can already be reconstructed in Isabelle/HOL. Reconstructing E's proofs separately, and reconstructing hybrid Leo-II+E proofs, are discussed elsewhere [19].
3. Our Satallax reconstruction code currently does not support the use of axioms in proofs. Supporting axioms is logically straightforward: using an axiom involves drawing it from ρ and adding a Cut step for that axiom.

5.1 Evaluation

A set of test proofs was first obtained by running Leo-II 1.6 for 30 seconds on all THF problems in the TPTP 5.4.0 problem set. In these experiments, Leo-II cooperated with version 1.8 of the E theorem prover. Leo-II produced 1537

[3] All the code can be downloaded from `http://www.cl.cam.ac.uk/~ns441/files/frocos_2015_code.tgz`

proofs. We used a repository version of Isabelle2013, the versions of Metis and Z3 packaged for Isabelle2013, and the experiments were done on a 1.6GHz Intel Core 2 Duo box, with 2GB RAM, and running Linux.

The translator was then run with a timeout of 10 seconds. By using Metis to emulate E, 1262 (82.1%) proofs were reconstructed entirely. If we treat E as an oracle (i.e., assuming E is sound, and that we have a perfect reconstruction for its proofs), then the number of reconstructed proofs increases to 1442 (93.8%). Currently, in Sledgehammer [16]—Isabelle's interface for external provers—Leo-II proofs are reconstructed by refinding using Metis and Z3 [9]. On the same problem set, Metis and Z3 were able to reconstruct 57.3% and 68.9% of the proofs respectively. Our scripts and data for this evaluation are available online.[4]

To evaluate the reconstruction of Satallax proofs we ran Satallax 2.8 in proof-generating mode on all THF problems in the TPTP 6.1.0 problem set, with a 30-second timeout for each problem. This set contains 3036 THF problems, 2458 of which are classified as theorems. Satallax provided proofs for 1860 THF problems. After removing proofs that involve axioms (because of the limitation described in §5 point 3) we are left with 1383 proofs.

Our reconstruction code was then run on each of these proofs, with a time-out of 10 seconds, and succeeded in reconstructing 1149 proofs (82%). On the same problem set, Metis and Z3 were able to reconstruct 51% and 67% of the proofs respectively. We used a repository version of Isabelle2014, and ran these experiments on a 2GHz Intel Core i7 machine, with 16GB RAM, and running OSX. Our scripts and data for this evaluation are available online.[5]

6 Related work

There is a fairly large literature on proof translation and reconstruction. We focus on two recent projects that are similar in spirit and setup to ours. A more detailed survey is given in the first author's dissertation [19].

Keller [13] uses an extension of the Calculus of Constructions (as implemented in Coq) as the host logic for theorems proved by SAT and SMT solvers. She develops a trusted SMT proof checker and uses it to check proofs produced by other SMT solvers, or to interpret those proofs in Coq's logic. Keller's checker follows the design of an SMT solver: it mediates between theory-specific solvers to refute a conjecture. The theory-specific solvers in Keller's work consist of theory-specific decision-procedures implemented using Coq. In order to use Keller's system, a proof from an SMT solver must first be translated into a form that can be processed by the trusted checker. (We think that this is not unlike the generation of cut programs, at least in spirit.) This is done as a preprocessing step, and this realises an embedding of the source calculus in the target calculus. The pure logic component of SMT is identical to that of SAT: proofs consist of refutations expressed using resolution. This means that the pure logic component of Keller's system is much simpler than the systems developed in our work: for

[4] `http://www.cl.cam.ac.uk/~ns441/files/recon_framework_results.tgz`
[5] `http://www.cl.cam.ac.uk/~ns441/files/frocos_2015_eval.tgz`

one thing, SMT lacks quantification. In a related respect, Keller's system is more complex than the systems developed in our work, since Keller's system supports a range of theories—as is expected in SMT. Currently, the state of the art in higher-order theorem proving does not interpret any theories other than equality [1].

Chihani et al. [8] describe an architecture based on higher-order logic programming for checking proof certificates. These certificates can take different forms, depending on the proof calculus of the source logic they relate to. Checking these certificates involves interpreting them into a form that can be checked as a proof in LKU, a linear logic they devised.

The system designed by Chihani et al. consists of three components: The *kernel* is an implementation of LKU's proof system; the *client* is the proof-producing theorem prover, which encodes its proofs in some format chosen by the authors of the theorem prover; and *clerks and experts* are two types of agent-like functions that carry out proof construction in LKU. They correspond to the two phases of proof-construction in a focussed proof system. This component serves to interpret the proof certificate into an LKU proof, that can then be checked by the kernel.

Together, clerks and experts seem to constitute an embedding of the source logic into a fragment of LKU. The clerks and experts seem to carry out a similar role to that of Keller's preprocessor, described above. (Recall that a different preprocessor might be needed for each theorem prover whose proofs we want to import.) In our work, this role is carried out by the emulations of inference rules, described in §4.4.

Using Keller's approach does not require the embedding to be described in the host system (i.e., Coq in that case). In contrast, the approach taken by Chihani et al., and us, do require this: emulation takes inferences in the source logic and produces inferences in the target logic.

7 Conclusions

A modular framework for translating proofs between two logics, based on cut machines, can accomplish efficient and robust proof reconstruction. However, detailed proofs are essential; otherwise, reconstructing proofs requires excessive search, which can be very expensive. Our implementation of the framework outperforms the existing method for reconstructing Leo-II and Satallax proofs in Isabelle/HOL.

Modularity is achieved partly by breaking up the translation process into steps (such as interpreting formulas, and emulating inferences) but also by using an abstract proof-building machine to mediate between the source and target logics. Our modular design is intended to facilitate reuse and modification. The extent to which this is the case remains to be seen, but we are encouraged that similar concepts were used by Keller and Chihani et al.

Acknowledgments. We thank Trinity College, Cambridge University Computer Lab, Cambridge Philosophical Society, and DAAD (the German Academic Exchange Service) for funding support. The anonymous reviewers and Chad Brown provided feedback, for which we are grateful.

References

1. Christoph Benzmüller. *Equality and Extensionality in Higher-Order Theorem Proving*. PhD thesis, Naturwissenschaftlich-Technische Fakultät I, Saarland University, 1999.
2. Christoph Benzmüller, Chad E. Brown, and Michael Kohlhase. Cut-Simulation and Impredicativity. *Logical Methods in Computer Science*, 5(1:6):1–21, 2009.
3. Christoph Benzmüller, Florian Rabe, and Geoff Sutcliffe. THF0 – The Core TPTP Language for Classical Higher-Order Logic. In P. Baumgartner, A. Armando, and D. Gilles, editors, *International Joint Conference on Automated Reasoning*, number 5195 in Lecture Notes in Artificial Intelligence, pages 491–506. Springer, 2008.
4. Christoph Benzmüller, Frank Theiss, Lawrence C. Paulson, and Arnaud Fietzke. LEO-II – A Cooperative Automatic Theorem Prover for Higher-Order Logic. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *International Joint Conference on Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 162–170. Springer, 2008.
5. Jasmin C. Blanchette. *Automatic Proofs and Refutations for Higher-Order Logic*. PhD thesis, Institut für Informatik, Technische Universität München, 2012.
6. Sascha Böhme and Tjark Weber. Designing Proof Formats: A User's Perspective. In Pascal Fontaine and Aaron Stump, editors, *International Workshop on Proof Exchange for Theorem Proving*, pages 27–32, 2011.
7. Chad E. Brown. Satallax: An Automated Higher-Order Prover. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *International Joint Conference on Automated Reasoning*, volume 7364 of *Lecture Notes in Computer Science*, pages 111–117. Springer, 2012.
8. Zakaria Chihani, Dale Miller, and Fabien Renaud. Foundational Proof Certificates in First-Order Logic. In Maria Paola Bonacina, editor, *Conference on Automated Deduction*, volume 7898 of *Lecture Notes in Computer Science*, pages 162–177. Springer, 2013.
9. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
10. Hans de Nivelle. Extraction of Proofs from Clausal Normal Form Transformation. In *Computer Science Logic*, volume 2471 of *Lecture Notes in Computer Science*, pages 584–598. Springer, 2002.
11. Gilles Dowek. Skolemization in Simple Type Theory: the Logical and the Theoretical Points of View. In C. Benzmüller, C. E. Brown, J. Siekmann, and R.Statman, editors, *Festschrift in Honour of Peter B. Andrews on his 70th Birthday*, Studies in Logic and the Foundations of Mathematics. College Publications, 2009.
12. Joe Hurd. First-Order Proof Tactics in Higher-Order Logic Theorem Provers. In Myla Archer, Ben Di Vito, and César Muñoz, editors, *Design and Application of Strategies/Tactics in Higher Order Logics*, number CP-2003-212448 in NASA Technical Reports, pages 56–68, September 2003.

13. Chantal Keller. *A Matter of Trust: Skeptical Communication Between Coq and External Provers*. PhD thesis, École Polytechnique, June 2013.
14. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.
15. Lawrence C. Paulson. *Isabelle – A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.
16. Lawrence C. Paulson and Jasmin C. Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In *International Workshop on the Implementation of Logics*. EasyChair, 2010.
17. Stephan Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.
18. Nik Sultana, Jasmin C. Blanchette, and Lawrence C. Paulson. LEO-II and Satallax on the Sledgehammer test bench. *Journal of Applied Logic*, 2012.
19. Nikolai Sultana. *Higher-order proof translation*. PhD thesis, Computer Laboratory, University of Cambridge, 2015. Available as Tech Report UCAM-CL-TR-867.
20. Geoff Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.
21. Christoph Weidenbach. Combining superposition, sorts and splitting. In John A. Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, pages 1965–2013. MIT Press, 2001.