Source-Level Proof Reconstruction for Interactive Theorem Proving

Lawrence C. Paulson and Kong Woei Susanto

Computer Laboratory, University of Cambridge, England
LP15@cam.ac.uk, kongwoei@gmail.com

Abstract. Interactive proof assistants should verify the proofs they receive from automatic theorem provers. Normally this proof reconstruction takes place internally, forming part of the integration between the two tools. We have implemented source-level proof reconstruction: resolution proofs are automatically translated to Isabelle proof scripts. Users can insert this text into their proof development or (if they wish) examine it manually. Each step of a proof is justified by calling Hurd's Metis prover, which we have ported to Isabelle. A recurrent issue in this project is the treatment of Isabelle's axiomatic type classes.

1 Introduction

Interactive theorem proving is notoriously labour intensive. Researchers are actively developing many verification technologies that are fully automatic, such as model checkers and SMT solvers. However, interactive theorem provers are unrivalled for their rich assertion languages that support quantifiers, recursive definitions and set notation. Interactive theorem provers also excel at managing elaborate hierarchies of formal developments. Much recent work concerns adding automation to interactive theorem provers to increase their users' productivity.

Correctness is paramount when external tools are coupled to interactive theorem provers. Although most verification tools appear to be highly reliable, errors can easily be introduced in the interface code, which translates problems from the interactive prover to the automatic tool. The translations themselves can be unsound: Meng and Paulson [9] show that using a compact translation improves the external tool's success rate, but admits the possibility of unsound proofs. Finally, many interactive provers adhere to the LCF philosophy [2] that all inferences must be checked by a small proof kernel. For all of these reasons, the interactive prover must check the automatic tool's output.

This paper concerns proof reconstruction at the source level. Rather than checking the proof behind the scenes, we deliver an actual piece of proof script. Source-level reconstruction is particularly appropriate when the external tool needs large computational resources: the user can write the proof script with the help of a powerful, multi-core workstation, and later replay it on an elderly laptop. A visible proof script also gives the highest possible confidence of correctness, as the reasoning is broken down into small steps that can be examined by hand.

Our particular interest lies in automatic theorem provers (ATPs) for first-order logic. We have elsewhere described [11] an interface between Isabelle and resolution theorem provers. Until now, we have been able to call external provers from Isabelle but could not do much with the result. We can now perform proof reconstruction by parsing the output of any ATP that delivers proofs in TSTP format [21]. Each inference is justified by a call to Hurd's Metis prover [5].

Paper outline. We begin (§2) by presenting the background material: Isabelle and our project to link it with automatic theorem provers. We then describe our experience of porting Metis to Isabelle (§3). There follows a lengthy presentation of how we generate single-step proof scripts (§4), with additional examples (§5). We finally give brief conclusions (§6).

2 Background

Isabelle/HOL [14] is an interactive theorem prover for higher-order logic, built upon the Isabelle logical framework [16]. (Henceforth, we shall use Isabelle and Isabelle/HOL synonymously.) Isabelle has been used for countless projects, such as the mechanization of the prime number theorem by Avigad et al. [1]. Isabelle's version of higher-order logic has many similarities with that used in HOL4 [15] and HOL Light [3]. All are based on polymorphic simple type theory, without subtyping or dependent types. Polymorphism is expressed by free type variables with implicit universal quantification, so a theorem like `rev (rev xs) = xs` is universally quantified both in the list `xs` and in the anonymous type of its elements. None of these tools offer explicit quantification over type variables [7].

2.1 Order-sorted Polymorphism

Order-sorted polymorphism [12] distinguishes Isabelle's version of higher-order logic from other versions. In Isabelle, a type may belong to any finite number of *type classes*. This idea, which gives a controlled treatment of overloading, originates with Wadler and Blott [22]. It is particularly powerful in an interactive theorem prover, where type classes can be specified using axioms [24]. Finite types, for example, can be characterized by an axiom stating that there exists a list enumerating all of the type's elements. We can then prove individual types to be finite by exhibiting such lists. We can even prove that certain type constructors, such as Cartesian product, preserve the property of finiteness; the proof requires exhibiting a construction that enumerates the elements of the product given enumerations of its component types.

Orderings provide other examples of type classes. The class `order` of partial orderings consists of all types equipped with a reflexive, anti-symmetric and transitive relation $\leq$. The class `linorder` of linear orderings extends `order` with the axiom $x \leq y \vee y \leq x$. To prove that the type of integers belongs to class `linorder`, we define a suitable instance of $\leq$ and prove suitable instances of the four axioms. The definition of $\leq$ on lists can refer to $\leq$ on list elements. The lexicographic ordering on lists yields a partial order if the list elements are

partially ordered and a linear order if they are linearly ordered. We declare such facts to Isabelle as follows:

```
instance list :: (order) order
  ...

instance list :: (linorder) linorder
  ...
```

Here, ... designates the proofs of the type class axioms, which typically refer to the corresponding properties of the element type. Paulson has shown how axiomatic type classes allow the various numeric types such as the integers, rationals and reals to be formalized without proving separate instances of algebraic properties for each type [17].

Axiomatic type classes are powerful, but they complicate the task of using external verification tools to prove Isabelle subgoals. For one thing, we must ensure that the automatic tools can reason with type classes as well as Isabelle can. Isabelle automatically applies its database of *instance* declarations to specific cases; for example, it can automatically recognize that lists of lists of integers belong to class *linorder*. Also, Isabelle automatically applies forgetful inclusions, for example that *linorder* is a subclass of *order*.

Incorrect treatment of type classes compromises soundness. Hurd [5] reports that types can be ignored entirely when integrating HOL4 with an automatic theorem prover. He describes a potential unsoundness by exhibiting a polymorphic "Russell constant" R such that $RR = \neg(RR)$ if types are ignored. (With types, the fatal equation cannot even be written.) He uses proof reconstruction to detect unsoundness, which he says "occurs in less than 1% of all HOL subgoals". Omitting type information is unthinkable in the presence of axiomatic type classes. They govern the properties that may be assumed of Isabelle's numerous overloaded functions, such as orderings and the arithmetic operators. Type information is necessary to stop the ATP from assuming, for example, that the subset relation is a linear ordering. Such errors are far more likely than the automatic discovery of Russell's paradox. Although unsound proofs cannot survive proof reconstruction, they can prevent sound proofs from being found.

We have described our first-order formalization of type information elsewhere [11].

- Types are first-order terms. A type operator taking n operands is simply an n-ary function symbol, while type variables are first-order variables. For example, the type of lists of integers might be `tc_list(tc_int)`.
- Type classes are first-order predicates. For example, the claim that the type of booleans is finite might be `class_finite(tc_bool)`. To express an *instance* declaration, for example that the Cartesian product of two finite types is itself finite, we use the obvious implication.
- Type class inclusions, such as that every linear ordering is a partial ordering, are formalized as implications between two atomic formulas.

As we are using resolution theorem provers, we use *clause form*: disjunctions of possibly negated atomic formulas. The formalization of the type class

hierarchy is close to clause form already, as it consists of implications between atomic formulas. Since resolution uses proof by contradiction, the conjecture to be proved is negated before it is translated into clauses; existing theorems being used as lemmas are also translated. In both cases, we have to formalize class constraints on the type variables appearing in a clause. (For examples, see §4.2 below.)

- Because the conjecture has been negated, the implicit universal quantification over its type variables becomes existential. Type variables in the conjecture are Skolemized, yielding new constants. Class constraints on these type variables become additional facts, such as `class_finite(t_a)`.
- Type variables in a lemma remain universally quantified. Type constraints in a clause take the form of additional negative literals, with the intuitive meaning such as "if `class_finite(T)` then ...".

Although this formalization is straightforward, it complicates the translations between higher-order logic and first-order logic. Proof reconstruction has to take first-order clauses that contain minimal type information and somehow recover as much of it as possible. The parts of the proof that relate to the type system must then be deleted, as they represent reasoning that Isabelle carries out implicitly. References to specific literals in a clause must be adjusted to account for such deletions.

2.2 The Interface Between Isabelle and ATPs

The Isabelle-ATP interface is designed to give users push-button assistance. When invoked, it examines its entire database of approximately 7000 theorems, selecting several hundred that appear to be relevant to the problem [10]. If the problem contains higher-order features such as functions or booleans being passed as arguments, then it is translated into a first-order form [9]. An automatic theorem prover is then invoked: currently, either E [19], SPASS [23] or Vampire [18]. The prover is given a considerable amount of processor time, perhaps 60 seconds per subgoal. Because this is much longer than users may want to wait, the prover runs in the background, allowing the user to continue working interactively. Responses from the ATP, whether successful or not, are reported when they appear.

The first working version of this system [11] could occasionally reconstruct Isabelle proofs, but it was problematical. It worked by reading the output of SPASS and emulating each SPASS inference rule in Isabelle. SPASS used many inference rules that differed slightly from one another. The information it delivered was incomplete, omitting details such as which subterm of which literal had been rewritten. Finally, SPASS re-ordered the literals in the clauses it was given.

In view of these difficulties, we decided instead to base proof reconstruction on Hurd's Metis prover, described below. We would parse an ATP's output merely to extract the names of the lemmas used; then, we would deliver that list

of lemmas to Metis. This was a sensible division of labour: leading ATPs can cope with problems that contain hundreds of clauses; Metis is relatively easy to integrate with interactive theorem provers.

However, we foresaw that Metis alone would often be insufficient. We performed extensive experiments [8] with 285 first-order problems. We were able to put nearly all of these problems into a minimal form by using an ATP to identify the necessary axioms. These minimal problems were similar to those that Metis would have to prove if we relied on it for proof reconstruction. Metis was unable to prove 8% of them in 10 seconds[1] and even given 60 seconds failed to prove 5%. It is a shame to let proof reconstruction fail for so many problems, especially as they are likely to be the most difficult ones. Another objection is that it is wasteful to use an ATP merely as a relevance filter when it delivers full proofs. We decided to parse these proofs and reconstruct them line by line, but instead of emulating an ATP's specific inference rules, we would justify each line by calling Metis. That project is the main subject of this paper.

3 Porting Metis to Isabelle

Hurd has written his Metis prover [5] in order to add further automation to HOL4. He has already shown [4] how difficult it is to harness existing ATPs for this purpose: ambiguities in their output complicate proof reconstruction. Although Metis cannot compete with the best ATPs, it includes a full implementation of the superposition calculus, and its performance is respectable [6].

Metis expresses proofs using five simple inference rules, designed for easy emulation in any interactive proof assistant for higher-order logic.[2]

$$\frac{}{A_1 \vee \ldots \vee A_n} \text{ axiom } [A_1, \ldots, A_n]$$

The *axiom* rule constructs an axiom. It takes as its argument a list of literals and returns their disjunction. The corresponding theorem must already be known to the proof assistant; axioms in the resolution proof correspond to uses of existing theorems as lemmas.

$$\frac{}{L \vee \neg L} \text{ assume } L$$

The *assume* rule takes as argument a literal L and returns the theorem $L \vee \neg L$.

$$\frac{A_1 \vee \ldots \vee A_n}{A_1[\sigma] \vee \ldots \vee A_n[\sigma]} \text{ instantiate } \sigma$$

The *instantiate* rule instantiates free variables. It takes two arguments, a substitution and a theorem, returning the appropriate instance of the theorem.

$$\frac{A_1 \vee \ldots \vee L \vee \ldots \vee A_m \qquad B_1 \vee \ldots \vee \neg L \vee \ldots \vee B_n}{A_1 \vee \ldots \vee A_m \vee B_1 \vee \ldots \vee B_n} \text{ resolve } L$$

[1] on a 2.8GHz Intel Xeon processor

[2] While we were revising this paper, Hurd released Metis 2.0. The set of inference rules in this version is simpler than its predecessor; we describe the new version.

The *resolve* rule takes a literal L and two theorems. The resulting theorem contains all literals of the first theorem other than L and all literals of the second theorem other than $\neg L$.

$$\frac{}{\neg(s = t) \vee \neg L \vee L[t]} \text{ equality } (L, p, t)$$

The *equality* rule takes a literal L, a path p and a term t. Writing s for the term denoted by path p and $L[t]$ for the result of replacing this occurrence by t, it yields a theorem stating that if $s = t$ and L then $L[t]$.

Unification is not required: all substitutions are supplied explicitly using the instantiate rule. The idea is that the Metis performs the unifications and proof reconstruction merely has to perform the supplied instantiations. This description makes it sound like integration with Isabelle ought to be easy, but we encountered some difficulties.

We began by modifying the Metis sources to compile with Isabelle. We then did the easy half of the integration: allowing Isabelle to call Metis, passing Isabelle theorems as axioms. We then turned to the difficult half, proof reconstruction, using the rules above. Most of the rules were straightforward; the difficult one was instantiate, and ironically we would have preferred to do our own unifications. Reconstructing type information missing from the resolution proof requires us to ignore any supplied instantiations that do relate to types, instead performing type inference to generate the correct type instantiations.

Converting Metis terms into Isabelle terms required type inference and proved to be surprisingly complicated. We finally got proof reconstruction working for first-order problems without types. Adding translations for higher-order problems and a treatment of types and type classes required another five months before proof reconstruction became robust. Among our problems was the need to adjust path quantifiers to account for the deletion of type literals. We also discovered several bugs (none affecting soundness) in Isabelle's forward proof mechanisms. A major difference between Isabelle and HOL4 is that Isabelle works almost entirely by backward proof, while HOL4 ultimately reduces everything to forward proof.

4 Producing Single-Step Proofs

The Metis proof method gives us a new way to reconstruct resolution proofs generated by another ATP. As in our first attempt [11], we translate each clause in the resolution proof to an Isabelle formula. Rather than emulate the ATP's specific inference rules, however, we prove each clause by calling Metis, passing it the inference rule's premises. Rather than parse the output of SPASS, we parse TSTP format [21], a recently introduced language for proof communication. TSTP format is easy to parse and is precisely documented.[3] E can output TSTP format and we have reason to hope that SPASS and Vampire will support it eventually.

[3] http://www.cs.miami.edu/~tptp/TPTP/SyntaxBNF.html

4.1 TSTP Format

TSTP (Thousands of Solutions from Theorem Provers) format defines a language for communicating proofs. A resolution prover typically produces a list of proof lines, each containing a clause and a justification referring to previous lines. The final line contains a contradiction. A proof line in TSTP format has the following syntax:

```
cnf(<name>,<formula_role>,<cnf_formula><annotations>).
```

Here is a description of these items:

- The `<name>` is a symbol identifying the formula. Although identifiers are permitted, all ATPs that we know of use positive integers.
- The `<formula_role>` is `axiom` for an axiom clause or `negated_conjecture` for clauses arising (even indirectly) from the negation of the conjecture.
- The `<cnf_formula>` is the formula itself. As we use clause form, it is a disjunction of literals. The disjunction symbol is `|` and the empty clause explicitly contains the literal `$false`.
- Any number of `<annotations>` may follow. These describe the provenance of the formula, for example as a named clause in a given file or proved from previous lines. Proof justifications can vary from one ATP to the next, but we only need to identify references to previous proof lines.

4.2 A Small Example

Let us examine the proof of a simple goal, one of our standard test problems. It arises in the middle of an interactive proof; the remaining goal is to prove $0 \leq f(x) + (-lb(x))$ from the two assumptions

$$\forall y.\, lb(y) \leq f(y) \quad \text{and} \quad \forall y.\, f(y) \leq lb(y) + g(y).$$

The proof is trivial. Ignore the second assumption and in the first one instantiate y by x; then, move $lb(x)$ across the inequality. Given the equivalent clause form, E immediately generates a 22-line resolution proof. Below we present about half of these lines to illustrate some issues involved in the translation. We have reformatted the lines and shortened some names to improve clarity.

The proof refers to specific axioms from the input file. The first axiom is $X - X = 0$.

```
cnf(216,axiom,
    (c_minus(X,X,X3)=c_HOL_Ozero(X3) |
     ~class_OrderedGroup_Oab__group__add(X3)),
    file('BigO__bigo_bounded2_1', cls_right__minus__eq_1)).
```

The variable $X3$ ranges over types, and the second literal restricts type $X3$ to belong to the class *OrderedGroup.ab_group_add* (Abelian groups). This type class was implicit in the original problem through its use of overloaded operators such as addition. As subtraction (`c_minus`) and zero (`c_HOL_Ozero`) are polymorphic,

they carry type information as an additional argument. We have to undo this translation to obtain an Isabelle formula.

The next two axioms express inclusions between type classes. They state that elements of *Ring_and_Field.ordered_idom* (ordered integral domains) also belong to the classes *OrderedGroup.ab_group_add* (Abelian groups) and *OrderedGroup.pordered_ab_group_add* (partially ordered Abelian groups). These axioms cannot be expressed in Isabelle, so references to them must be deleted from proofs.

```
cnf(343,axiom,
    (class_OrderedGroup_Opordered__ab__group__add(X3) |
     ~class_Ring__and__Field_Oordered__idom(X3)),
    file('BigO__bigo_bounded2_1', clsrel_Ring__and__Field_123)).
cnf(350,axiom,
    (class_OrderedGroup_Oab__group__add(X3) |
     ~class_Ring__and__Field_Oordered__idom(X3)),
    file('BigO__bigo_bounded2_1', clsrel_Ring__and__Field_923)).
```

Negating the test problem generates an additional conjecture clause: one asserting type information about the type variable `'b`, which is implicit in the subgoal. As remarked above in §2.1, negating a polymorphic subgoal creates an existentially quantified type variable that turns into a Skolem constant. This type, `t_b`, belongs to the class of ordered integral domains. The other conjecture clause shown refers to the assumption $\forall y.\, lb(y) \leq f(y)$.

```
cnf(335,negated_conjecture,
    (class_Ring__and__Field_Oordered__idom(t_b)),
    file('BigO__bigo_bounded2_1', tfree_tcs)).
cnf(336,negated_conjecture,
    (c_lessequals(v_lb(X3),v_f(X3),t_b)),
    file('BigO__bigo_bounded2_1', cls_conjecture_0)).
```

Now the proof begins. The first two lines follow type class inclusions, deducing that the type variable `'b` belongs to the class of Abelian groups. During the translation, such proof steps must be deleted.

```
cnf(366,negated_conjecture,
    (class_OrderedGroup_Opordered__ab__group__add(t_b)),
    inference(spm,[status(thm)],[343,335,theory(equality)])).
cnf(367,negated_conjecture,
    (class_OrderedGroup_Oab__group__add(t_b)),
    inference(spm,[status(thm)],[350,335,theory(equality)])).
```

The next proof step creates an instance of $X - X = 0$ for type `'b`, deleting the second literal. As this step still concerns type information, it too must be deleted.

```
cnf(1968,negated_conjecture,
    (c_minus(X,X,t_b)=c_HOL_Ozero(t_b)),
    inference(spm,[status(thm)],[216,367,theory(equality)])).
```

After several steps, omitted here, E nears the conclusion. E takes its time: the last three steps each express contradiction. Naturally such repeated steps must be deleted.

```
cnf(4421,negated_conjecture,
    ($false | $false),
    inference(rw,[status(thm)],
        [inference(rw,[status(thm)],[4420,2575,theory(equality)]),
                  336,theory(equality)])).
cnf(4422,negated_conjecture,
    ($false),
    inference(cn,[status(thm)],[4421,theory(equality)])).
cnf(4423,negated_conjecture,
    ($false),
    4422,['proof']).
```

4.3 The Translation Method

Our system translates a TSTP file into an Isabelle proof by parsing it, reconstructing type information, removing redundancy and discarding steps that have no place in Isabelle's inference system. If the original subgoal contained higher-order features such as function variables, then its first-order counterpart will contain an explicit "apply" operator and "is-true" predicate [9]; these must also be removed.

We represent abstract syntax by a data structure for n-ary trees with integers in the leaves and strings labelling the branch nodes. Here is the corresponding ML datatype declaration:

```
datatype stree = Int of int | Br of string * stree list;
```

Isabelle provides top-down parsing primitives that we use to parse CNF lines. We have written additional code to translate TSTP identifiers (which must be strictly alphanumeric) back to the Isabelle ones from whence they came. For example, `_0` in a TSTP identifier corresponds to a full stop (`.`) in an Isabelle identifier. We first remove the prefix indicating what class of identifier it is: `c_` for constant, `tc_` for type constructor, `T_` for type variable, etc. If the ATP generates its own variables, such as `X3`, then we have to guess what they stand for. Note that variables in a resolution proof always start with a capital letter.

Next, we identify type information. Although type class information must be deleted from proofs, it must be noted so that the Isabelle terms we generate are well-typed. Polymorphic constants must be reconstructed. Given a constant, we look up its definition. If its type scheme contains n type variables, then its last n arguments represent types. Type literals in a clause help us by specifying the type classes of some type variables.

```
cnf(2542,negated_conjecture,
    (c_plus(c_HOL_Ozero(t_b),X3,t_b)=X3 |
     ~class_OrderedGroup_Oab__group__add(t_b)),
    inference(spm,[status(thm)],[147,1968,theory(equality)])).
```

For example, the type variable *'b* in the clause above belongs to the type class *OrderedGroup.ab_group_add*; this determines the types of the constants 0 and + in the clause. In other clauses, the type class of *'b* is *Ring_and_Field.ordered_idom*.

Having type class constraints for all type variables allows us to generate the corresponding instances of constants, which in turn should allow us to reconstruct terms correctly. Our experience confirms this, even with 100-step proofs involving many type classes.

For each clause, we separate the type literals from the others. The remaining literals, those meaningful to Isabelle, are formed into a disjunction. Type inference is then performed using the stored type class information. The disjunction is finally generalized over its free variables. If no "real" literals are present, then the clause contains type information only; we represent it differently from an empty clause, which represents contradiction.

In order to re-assemble the proof, we examine the supplied annotations, extracting all the integers they contain. With E, all integers in annotations refer to previous proof lines. (If an ATP used integers for other purposes, then we would need to identify and delete them.) We go through three phases.

1. Axiom references are deleted. Axioms represent either type information or known Isabelle theorems. The latter can be designated by their Isabelle identifiers, so it would be pointless to include them as proof lines. At this stage we also eliminate duplicate proofs of an assertion; these typically differ in type literals only.
2. The proof is then purged of chains of reasoning relating to types. This step is repeated until the deletions stop.
3. Finally, some proof lines are removed in order to reduce the proof length. Removal of a line involves replacing all references to it by references to its antecedents. The proof tree becomes shorter but bushier.

The user can control the factor by which proof lines are combined to shorten the proof. If the factor is n, then only every nth line is retained. Polymorphic lines must be removed because Isabelle does not have type quantifiers; such lines typically involve instantiating a polymorphic lemma to other polymorphic types. After removing all polymorphic lines, we still have a proof because the final line is monomorphic: it is simply *False* and has type *bool*.

We output the proof as an Isar structured script [13] of a simple form: a series of assertions and justifications. The script begins with the proof method *neg_clausify*, which negates the subgoal and converts it into clauses. All assertions must be explicit in Isar proofs, so the conjecture clauses that are used must be introduced explicitly using Isar *assume* declarations. Each intermediate clause of the proof is introduced using a *have* declaration and proved by calling Metis with its antecedents. Isar requires the final line to be declared using *show*; the assertion is always *False*.

```
proof (neg_clausify)
fix x
assume 0: "⋀y. lb y ≤ f y"
assume 1: "¬ (0::'b) ≤ f x + - lb x"
have 2: "⋀X3. (0::'b) + X3 = X3"
  by (metis diff_eq_eq right_minus_eq)
have 3: "¬ (0::'b) ≤ f x - lb x"
  by (metis 1 compare_rls(1))
have 4: "¬ (0::'b) + lb x ≤ f x"
  by (metis 3 le_diff_eq)
show "False"
  by (metis 4 2 0)
qed
```

Note that E's line numbers, which ranged into the thousands, have been renumbered starting from zero. References to axiom clauses are replaced by the corresponding Isabelle theorem references, such as *le_diff_eq* or *compare_rls(1)*.

As of this writing, the default output contains a huge amount of type information, most of which is redundant. We have switched this off for our examples in order to improve readability. Some problems require type information, but we should generate no more than is necessary. This merely requires some bookkeeping to ensure that (variable, type) and (type variable, class) pairs only appear once.

```
proof (neg_clausify)
fix x
assume 0: "Y ⊆ X ∨ X = Y ∪ Z"
assume 1: "Z ⊆ X ∨ X = Y ∪ Z"
assume 2: "(¬ Y ⊆ X ∨ ¬ Z ⊆ X ∨ Y ⊆ x) ∨ X ≠ Y ∪ Z"
assume 3: "(¬ Y ⊆ X ∨ ¬ Z ⊆ X ∨ Z ⊆ x) ∨ X ≠ Y ∪ Z"
assume 4: "(¬ Y ⊆ X ∨ ¬ Z ⊆ X ∨ ¬ X ⊆ x) ∨ X ≠ Y ∪ Z"
assume 5: "⋀V. ((¬ Y ⊆ V ∨ ¬ Z ⊆ V) ∨ X ⊆ V) ∨ X = Y ∪ Z"
have 6: "LOrder.sup Y Z ≠ X ∨ ¬ X ⊆ x ∨ ¬ Y ⊆ X ∨ ¬ Z ⊆ X"
  by (metis 4 sup_set_eq)
have 7: "Z ⊆ x ∨ LOrder.sup Y Z ≠ X ∨ ¬ Y ⊆ X"
  by (metis 3 sup_set_eq Un_upper2 sup_set_eq 1 sup_set_eq)
have 8: "Z ⊆ x ∨ LOrder.sup Y Z ≠ X"
  by (metis 7 Un_upper1 sup_set_eq 0 sup_set_eq)
have 9: "LOrder.sup Y Z = X ∨ ¬ Z ⊆ X ∨ ¬ Y ⊆ X"
  by (metis equalityI 5 sup_set_eq Un_upper2 sup_set_eq Un_upper1 sup_set_eq
Un_least sup_set_eq)
have 10: "Y ⊆ x"
  by (metis 2 sup_set_eq Un_upper2 sup_set_eq 1 sup_set_eq Un_upper1 sup_set_eq
0 sup_set_eq 9 Un_upper2 sup_set_eq 1 sup_set_eq Un_upper1 sup_set_eq 0 sup_set_eq)
have 11: "X ⊆ x"
  by (metis Un_least sup_set_eq 9 Un_upper2 sup_set_eq 1 sup_set_eq Un_upper1
sup_set_eq 0 sup_set_eq 8 9 Un_upper2 sup_set_eq 1 sup_set_eq Un_upper1 sup_set_eq
0 sup_set_eq 10)
show "False"
  by (metis 11 6 Un_upper2 sup_set_eq 1 sup_set_eq Un_upper1 sup_set_eq 0 sup_set_eq
9 Un_upper2 sup_set_eq 1 sup_set_eq Un_upper1 sup_set_eq 0 sup_set_eq)
qed
```

Fig. 1. Proof of First Example

5 Two Larger Examples

The example presented above was trivial. Here, we consider two examples that involve lengthy proofs. These illustrate the appearance of non-trivial proofs and the effect of the reduction factor.

```
proof (neg_clausify)
fix c x
assume 0: "⋀X1. |h X1| ≤ c * |f X1|"
assume 1: "c ≠ (0::'a)"
assume 2: "¬ |h x| ≤ |c| * |f x|"
have 3: "⋀X3. |0::'a| = |X3| * (0::'a) ∨ ¬ (0::'a) ≤ (0::'a)"
  by (metis abs_mult_pos mult_cancel_right1)
have 4: "|(1::'a) * (0::'a)| = - ((1::'a) * (0::'a))"
  by (metis abs_of_nonpos mult_le_cancel_right2 max.f_below_strict_below.below_refl
max.f_below_strict_below.below_refl)
have 5: "c = (0::'a) ∨ c < (0::'a)"
  by (metis linorder_antisym_conv2 2 abs_of_nonneg linorder_linear 0)
have 6: "(0::'a) ≤ (1::'a)"
  by (metis zero_le_square AC_mult.f.commute mult_cancel_left1)
have 7: "⋀X3. (0::'a) = |X3| ∨ X3 ≠ (0::'a)"
  by (metis abs_minus_cancel neg_equal_iff_equal 4 mult_cancel_right1 3 mult_cancel_right1
max.f_below_strict_below.below_refl mult_cancel_right1 minus_equation_iff 3
mult_cancel_right1 max.f_below_strict_below.below_refl)
have 8: "⋀X1 X3. (0::'a) * |X1| = |X3 * X1| ∨ X3 ≠ (0::'a)"
  by (metis abs_mult 7)
have 9: "⋀X1 X3. X3 * X1 = (0::'a) ∨ X3 ≠ (0::'a)"
  by (metis zero_less_abs_iff 8 mult_cancel_left1 abs_not_less_zero 3 mult_cancel_right1
max.f_below_strict_below.below_refl)
have 10: "⋀X3. X3 * (1::'a) = (0::'a) ∨ |X3| ≠ (0::'a) ∨ ¬ (0::'a) ≤ (1::'a)"
  by (metis abs_eq_0 abs_mult_pos mult_cancel_right1 AC_mult.f.commute)
have 11: "⋀X3. ¬ |X3| < (0::'a) ∨ ¬ (0::'a) ≤ (1::'a)"
  by (metis abs_not_less_zero abs_mult_pos mult_cancel_right1 AC_mult.f.commute)
have 12: "⋀X3. X3 = (0::'a) ∨ ¬ |X3| ≤ (0::'a) ∨ ¬ (0::'a) ≤ (1::'a)"
  by (metis abs_le_zero_iff abs_mult_pos mult_cancel_right1 AC_mult.f.commute
mult_cancel_right1 AC_mult.f.commute)
have 13: "⋀X3. X3 * X3 = (0::'a) ∨ ¬ X3 ≤ (0::'a) ∨ ¬ (0::'a) ≤ X3"
  by (metis 12 6 abs_mult abs_mult_self AC_mult.f.commute mult_le_0_iff)
have 14: "⋀X3. |h X3| ≤ (0::'a) ∨ ¬ |f X3| ≤ (0::'a)"
  by (metis 0 10 mult_cancel_right1 AC_mult.f.commute 6 abs_mult AC_mult.f.commute
9 mult_eq_0_iff abs_mult_self 13 abs_ge_zero abs_mult_pos mult_cancel_right1
AC_mult.f.commute 6)
have 15: "⋀X3. |h X3| < (0::'a) ∨ ¬ c < (0::'a) ∨ ¬ (0::'a) < |f X3|"
  by (metis order_le_less_trans 0 mult_less_0_iff)
show "False"
  by (metis 15 5 1 11 6 2 abs_of_nonpos 2 abs_of_nonneg linorder_linear 0 7
mult_cancel_right1 14 linorder_not_le 12 6 linorder_not_le)
qed
```

Fig. 2. Proof of Second Example

Our first example involves proving a logical equivalence:

$$X = Y \cup Z \iff Y \subseteq X \wedge Z \subseteq X \wedge (\forall V.\, Y \subseteq V \wedge Z \subseteq V \rightarrow X \subseteq V)$$

E produces a TSTP proof consisting of 53 lines. These include 6 conjecture clauses. The resulting single-step proof contains a further 26 steps. Increasing the reduction factor to 4 results in a proof consisting of 6 steps (Fig. 1). The series

of claims is almost legible, especially if we note that `LOrder.sup` is equivalent to union.

Our second example involves proving $|h(x)| \leq |c| \times |f(x)|$ from the assumptions $\forall x. |h(x)| \leq c \times |f(x)|$ and $c \neq 0$. The original hand proof is six lines long, and involves first proving the lemma $c \times |f(x)| \leq |c| \times |f(x)|$ using the monotonicity of multiplication. However, E produces a 303-step proof. The single-step Isabelle proof contains 61 steps (including 3 conjecture clauses), which decreases to 16 steps if we set the reduction factor to 4 (Fig. 2). It is clear that this proof could be further simplified. For example, claim 6 proves $0 \leq 1$, which is already a theorem in Isabelle. We should process these proofs to make them as simple and as natural as possible.

6 Conclusions

Translating the output of an automatic theorem prover into a fragment of proof script has many advantages. Correctness can be checked automatically, but the script is also open to manual inspection. User can insert these scripts into their proof developments, eliminating the need to run ATPs a second time. There are some rough edges at present, but the system reliably translates proofs that are hundreds of lines long.

We have ported Hurd's Metis prover to Isabelle to use as the basis of these proof scripts. In most cases, a single call to Metis suffices to reproduce the proof. Sometimes, however, Metis fails or a more detailed proof description is required. We have therefore implemented a translation from the TSTP output format to Isar structured proof scripts. E produces TSTP output and we expect other ATPs to follow suit. The work can similarly be ported to other interactive provers, since the Isar scripts we generate are simple sequences of assertions justified by Metis calls.

The idea of an interactive prover generating its own proof scripts is promising. It opens up many new avenues for research.

Acknowledgements. The research was funded by the EPSRC grant GR/S57198/01 *Automation for Interactive Proof*. The first implementation of proof reconstruction in our system is due to Claire Quigley. Jia Meng also made many contributions to our work. Joe Hurd offered much advice on Metis. The TPHOLs referees made many valuable comments.

References

1. Jeremy Avigad, Kevin Donnelly, David Gray, and Paul Raff. A formally verified proof of the prime number theorem. *ACM Transactions on Computational Logic*, in press.
2. Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation.* LNCS 78. Springer, 1979.

3. John Harrison. HOL Light: A tutorial introduction. In Mandayam K. Srivas and Albert John Camilleri, editors, *Formal Methods in Computer-Aided Design: FMCAD '96*, LNCS 1166, pages 265–269. Springer, 1996.
4. Joe Hurd. Integrating Gandalf and HOL. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics: TPHOLs '99*, LNCS 1690, pages 311–321. Springer, 1999.
5. 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 NASA/CP-2003-212448 in NASA Technical Reports, pages 56–68, September 2003.
6. Joe Hurd. Metis performance benchmarks. http://gilith.com/software/metis/performance.html, 2004.
7. Thomas F. Melham. The HOL logic extended with quantification over type variables. *Formal Methods in System Design*, 3(1-2):7–24, 1994.
8. Jia Meng and Lawrence C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. In Sutcliffe et al. [20], pages 53–69.
9. Jia Meng and Lawrence C. Paulson. Translating higher-order problems to first-order clauses. In Sutcliffe et al. [20], pages 70–80.
10. Jia Meng and Lawrence C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *Journal of Applied Logic*, in press.
11. Jia Meng, Claire Quigley, and Lawrence C. Paulson. Automation for interactive proof: First prototype. *Information and Computation*, 204(10):1575–1596, 2006.
12. Tobias Nipkow. Order-sorted polymorphism in Isabelle. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 164–188. Cambridge University Press, 1993.
13. Tobias Nipkow. Structured Proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, LNCS 2646, pages 259–278. Springer, 2003.
14. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002. LNCS Tutorial 2283.
15. Michael Norrish and Konrad Slind. The HOL system description. On the Internet at http://hol.sourceforge.net/, 2007.
16. Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.
17. Lawrence C. Paulson. Organizing numerical theories using axiomatic type classes. *Journal of Automated Reasoning*, 33(1):29–49, 2004.
18. Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Communications*, 15(2):91–110, 2002.
19. Stephan Schulz. System description: E 0.81. In David Basin and Michaël Rusinowitch, editors, *Automated Reasoning — Second International Joint Conference, IJCAR 2004*, LNAI 3097, pages 223–228. Springer, 2004.
20. Geoff Sutcliffe, Renate Schmidt, and Schulz Schulz, editors. *FLoC'06 Workshop on Empirically Successful Computerized Reasoning*, volume 192 of *CEUR Workshop Proceedings*, 2006.
21. Geoff Sutcliffe, Jürgen Zimmer, and Stephan Schulz. TSTP data-exchange formats for automated theorem proving tools. In W. Zhang and V. Sorge, editors, *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, number 112 in Frontiers in Artificial Intelligence and Applications, pages 201–215. IOS Press, 2004.

22. Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *16th Annual Symposium on Principles of Programming Languages*, pages 60–76. ACM Press, 1989.
23. Christoph Weidenbach. Combining superposition, sorts and splitting. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 27, pages 1965–2013. Elsevier Science, 2001.
24. Markus Wenzel. Type classes and overloading in higher-order logic. In Elsa L. Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics: TPHOLs '97*, LNCS 1275, pages 307–322. Springer, 1997.