Apply2Isar: Automatically Converting Isabelle/HOL Apply-Style Proofs to Structured Isar

Sage Binder[1]
University of Iowa, Iowa City, IA, USA

Hanna Lachnitt
Stanford University, Stanford, CA, USA

Katherine Kosaian
University of Iowa, Iowa City, IA, USA

Abstract

In Isabelle/HOL, declarative proofs written in the Isar language are widely appreciated for their readability and robustness. However, some users may prefer writing procedural "apply-style" proofs since they enable rapid exploration of the search space. To get the best of both worlds, we introduce Apply2Isar, a tool for Isabelle/HOL that automatically converts apply-style proofs to declarative Isar. This allows users to write complex, possibly fragile apply-style proofs, and then automatically convert them to more readable and robust declarative Isar proofs. To demonstrate the efficacy of Apply2Isar in practice, we evaluate it on a large benchmark set consisting of apply-style proofs from the Isabelle Archive of Formal Proofs.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Proof Assistants, Isabelle/HOL, Isabelle/ML, Isabelle/Isar, Proof Refactoring

Digital Object Identifier 10.4230/LIPIcs.ITP.2026.18

Supplementary Material
Software (Source Code): `https://github.com/sagebinder/apply2isar` [6]
archived at `swh:1:dir:8753f0c693cdc7159375110e6dd9d977e62984a8`

Funding This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract FA8750-24-2-1001. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of DARPA.

Acknowledgements Many thanks to Fabian Huch for his valuable feedback on the paper and his Isabelle/ML guidance. We also thank Andrew Marmaduke, Kartik Sabharwal, Shweta Rajiv, and Apoorv Ingle for useful discussions and helpful feedback on the paper. For help with specific technical questions via the Isabelle Zulip channel, we thank Jonathan Julián Huerta y Munive, Kevin Kappelmann, and Lukas Bartl. Finally, we thank the anonymous ITP reviewers for their valuable feedback and comments.

Declaration about GenAI/LLM use We did not use any generative AI in implementing `Apply2Isar` or in writing this paper.

1 Introduction

The interactive theorem prover Isabelle [22] allows two primary styles for constructing proofs: apply-style proof scripts and structured Isar proofs. Apply-style proofs are *procedural* proofs that generally work backward, starting from the desired goal(s) and invoking methods to

[1] Corresponding author

either discharge open goals or produce new proof obligations – this essentially builds a proof tree. Importantly, in an apply-style proof, the user must interactively step through the proof to view the intermediate proof states. Structured Isar, on the other hand, is *declarative* in that the user explicitly writes intermediate goals and the proof body generally moves forward (though one can provide an initial backward reasoning step), referencing previous goals to prove new goals. Moreover, structured Isar supports various proof structures that allow complex non-linear chains of reasoning to be presented in a more organized and human-readable way. This improved expressivity is one reason that it is now generally preferred over apply-style proofs by the Isabelle community, including by the *Archive of Formal Proofs* (AFP) [1], which is a large and actively-maintained online repository of Isabelle proof developments.

Isar's declarative format was designed with readability and flexibility in mind, drawing inspiration from informal mathematical traditions and high-level programming languages [28], as well as the declarative language of the Mizar proof assistant [20]. It succeeds on many fronts: explicit intermediate goals help clarify the proof outline and flow; the forward reasoning is often clearer to follow; and the multitude of reasoning patterns allow proofs to more closely match the reasoning one might find in a mathematical text. Additionally, compared to apply-style proofs, structured Isar arguably provides inherent robustness benefits: its declarative nature ensures that a proof's overall structure, once written, remains fixed even when method behaviors change (which can happen, for instance, when simplification rules are added or removed, or when a referenced lemma is refactored). If a structured Isar proof does break, it is usually easy to identify and repair broken steps, even for a user who is unfamiliar with the details of the proof (such as an AFP editor).[2]

In contrast to structured Isar, apply-style proofs can be difficult to repair: if a step in an apply-style proof suddenly yields a different (unintended) proof state, an error may not occur until much later in the proof, which can make it hard to identify which method invocation is the culprit. Even once the problematic step is identified, how it should be repaired may not be clear because the correct (intended) proof state is not a fixed property of the proof.

Despite the advantages of structured Isar, apply-style proofs are sometimes easier to *write* since they directly manipulate the proof state and let Isabelle automatically manage current goals, simplify expressions, apply lemmas with unification, and so on. Apply-style proofs also enable rapid exploration of the search space because different methods can quickly be tried with immediately visible results. We explain the distinction between apply-style proofs and structured Isar in more depth in Section 2.

Even in structured Isar proofs, Isabelle/HOL's Sledgehammer automation sometimes performs better on goals that have first been simplified by `simp` or `auto` [7], which can lead to brittle proof patterns if not refactored. Thus, a user might want to write an apply-style proof first, then convert it to structured Isar for greater readability. To do a direct translation, the user must step through the apply-style proof and copy each set of intermediate goals along with the corresponding method application. Alternatively, the user may use the insight gained from writing the apply-style proof to write a new structured Isar proof from scratch. Both methods are labor-intensive. To alleviate this labor and achieve the best of both worlds, we introduce `Apply2Isar`, a tool implemented in Isabelle/ML that automatically converts Isabelle apply-style proofs to structured Isar proofs. We give a motivating example of the potential brittleness of apply-style proofs and an overview of `Apply2Isar` in Section 3. As

[2] Isabelle/HOL's powerful automation tools (like `try0` and `sledgehammer`) are often extremely effective at fixing broken steps in structured proofs.

apply-style proofs can admit complex behavior that does not always translate neatly to structured Isar, we encountered various design challenges in implementing `Apply2Isar`, which we discuss in Section 4. To demonstrate the efficacy of `Apply2Isar` in practice, we evaluated it on thousands of proofs from the AFP, where it produced total or partial translations in 95%–99% of test cases. See Section 5 for a detailed discussion of this evaluation. In Section 6, we discuss future work, including how ideas from related proof refactoring tools may be integrated into `Apply2Isar`. We conclude in Section 7.

2 Preliminaries

The Isabelle tutorial provides detailed explanations of the various proof commands [21], and the Isar reference manual explains the language in depth [26]. We review only basic concepts here, and introduce further concepts throughout the paper as needed.

Isar is a proof language that enables users to write sequences of proof commands in an interactive manner. Methods can be used by commands to operate on pending goals. New methods can be built up from other methods via combinators. Methods do not allow direct goal addressing, but either refer to the first goal or to all goals.[3] While seemingly similar to methods, tactics are programs written directly in ML, the programming language in which Isabelle is implemented. In the early days of Isabelle, the user would use ML functions, including tactics, directly. Nowadays, tactic scripts are a thing of the past for end-users who only interact with the Isar environment.

The Isar virtual machine (Isar/VM) keeps track of nodes consisting of the pending goal, the proof context, and the linguistic mode. There are two such modes that are of particular interest to us. The first is the *prove* mode, which allows the user to refine goals, e.g., with the `apply m` command, which applies the proof method `m` and returns zero or more new goals. The second mode is the *state* mode, in which declarative statements such as new claims, assumptions, or definitions can be written. One way to do this is with the `have [stmt]` command, which opens a local goal and switches to the *prove* mode. Once all pending goals are proven, the VM can switch from a node in the *prove* mode to one in the *state* mode, e.g., via the `done` command, which finishes a proof. For simplicity, we omit any discussion of the third mode, called the *chain* mode.

For the purposes of this paper, we define an *apply-style proof* to be one where nodes are in the *prove* mode, which also classifies it as a procedural proof. The only exception to this are steps that close a goal. The name "apply-style" is somewhat of a misnomer since many other commands, such as `using` or `unfolding`, can be used to transition between two nodes in the *prove* mode. However, `apply` is often the most frequent command in these procedural proofs, so Isabelle users often use the term "apply-style" to informally describe this style. Apply-style proofs are an emulation of tactic scripts. Note that users of other ITPs (and even many Isabelle users) will often refer these kinds of procedural proofs as "tactic scripts".

The `proof m` command can be used to transition from the *prove* mode into the *state* mode after applying method `m`. The `proof` command can also be invoked with a dash (`proof(-)`) instead of a method, in which case it directly transitions to the *state* mode. To finish a pending goal established by `proof m`, `show` is used in place of `have`; if multiple goals are established by `proof m`, each one must be proven in a `show` statement. Finally, the `qed` command finishes the proof, resulting in a new theorem. This is the basis for what we will call a *structured Isar proof* in this paper, an instance of the declarative proof style. While

[3] Although, there is a combinator that applies a method to the first n goals.

each new goal declaration will allow the user to switch back to apply-style proving, we restrict structured Isar proofs to mean only those that may use `using` or `unfolding` followed by a single application of the `by`, immediate proof (`.`), or standard proof (`..`) commands. This enforces that for any complex transformation, the goal must be explicitly stated.

In structured Isar proofs, facts can be labeled (e.g. `have myFact: "X"`) and once proven, referenced later, either by name (e.g. `myFact`) or as a fact literal (e.g. `‹X›`). To avoid excessive fact labels, one can chain facts, for instance with the `then` and `moreover`-`ultimately` commands. Additionally, one can fix local variables with `fix` and assert assumptions with `assume`; the Isabelle kernel will refuse to prove a theorem if any fixed variables, asserted assumptions, or `show` statements do not match the goal(s). The command `by m` is similar to `apply m done`, except it is *terminal* in that it runs `m` until it finds a solved state or fails. One can also write `by (m1) m2`, which tries to solve the current proof state by applying `m1` and `m2` in sequence. We explain more commands in Table 1.

The following is an example of a structured Isar proof, where we use `[...]` to elide some of the commands used to discharge proof obligations:

```
lemma "A ∧ C"
proof(-)
  have "A ∧ B" [...]                    (* Some proof of [ A ∧ B ] *)
  then have "A" by (rule conjunct1)     (* Uses the previous fact [ A ∧ B ] *)
  moreover have "C" [...]               (* Some proof of [ C ] *)
  ultimately show "A ∧ C" by (rule conjI) (* Uses the collected facts [ A, C ] *)
qed
```

Throughout the paper, we present many examples of apply-style and structured Isar proofs. As the proof state is not explicit in apply-style proofs, we use comments (denoted by `(* ... *)`) to describe relevant information about the state (goals, facts in the context, etc.) at key steps. Note that when we write "`apply m (* ... *)`", we intend the comment to indicate information about the proof state *after* `m` is applied. We also use the common Isabelle convention of indenting each proof command by $n - 1$ spaces beyond the baseline indentation of 2, where n is the number of goals in the proof state when the command is invoked.

3 Motivating Example and Overview

Consider the following simple example of a lemma with an apply-style proof:

```
consts A :: "bool" B1 :: "bool" B2 :: "bool" C :: "bool"

lemma L1: "B1 ⟹ B2 ⟹ A" [...]
lemma L2: "B1" [...]
lemma L3: "C ⟹ B2" [...]
lemma L4: "C" [...]

lemma apply_prf: "A" (* Our initial goal is [ A ] *)
  apply (rule L1)  (* By backward chaining, L1 reduces this goal to [ B1, B2 ] *)
   apply (rule L2) (* Applying L2 discharges B1, so [ B2 ] is the remaining goal *)
  apply (rule L3)  (* By backward chaining, L3 reduces this goal to [ C ] *)
  by (rule L4)     (* Applying L4 finally discharges [ C ] *)
```

Now, suppose that lemma `L1` is refactored so its hypotheses are swapped:

```
lemma L1: "B2 ⟹ B1 ⟹ A"  [...]
```

This simple change breaks the apply-style proof of lemma `apply_prf` shown above. Specifically, `rule L2` will now be applied to the goal `B2` instead of `B1`, resulting in an error. A user investigating the broken step interactively can see that when `rule L2` is applied, the current goals are `[ B2, B1 ]`. However, it is neither explicit that `rule L2` failed because it was expecting the first goal to be `B1`, nor that the expected result of `rule L2` is the goal `[ B2 ]`.[4]

Now consider a corresponding structured Isar proof for the same lemma:

```
lemma structured_prf: "A"
proof(-)
  have "B1" by (rule L2)
  moreover have "B2"
  proof(-)
    have "C" by (rule L4)
    then show "B2" by (rule L3)
  qed
  ultimately show "A" by (rule L1)
qed
```

If, in this structured proof, `L1` is again changed as described above, the proof will fail at `rule L1` – this more accurately reflects the cause of the failure. Moreover, it is now explicit that `rule L1` was intended to prove `A` from the fact set `[ B1, B2 ]`. This helps the user repair the broken step or – as might be necessary when the method application is very complicated – find an alternative proof. Crucially, the effects of the broken step are isolated – the still-working parts of the proof can remain intact.

`Apply2Isar` can automatically translate existing apply-style proofs to more maintainable structured Isar proofs. At a high level, `Apply2Isar` replays a given apply-style proof and records the remaining goals after each method application, then constructs a structured Isar proof by printing the intermediate goals (and the corresponding method applications) in reverse. The reversal is natural because Isar is more conducive to forward reasoning whereas apply-style proofs generally proceed backward. In the generated Isar, we use the `fact` method as "glue" between steps – the command `by (fact h)` proves a goal that unifies with `h`, and fails otherwise [26].

The previous translation example demonstrates stylistic choices such as reordered goals and the use of constructs like subproofs and `moreover-ultimately`. Used appropriately, these choices can make the proof more natural to a human reader. However, what makes a proof natural and readable is ultimately subjective. Indeed, readers with Isabelle/HOL expertise may have opinions on how the previous example could have been more natural. With `Apply2Isar`, we avoid making any stylistic choices on behalf of the user; instead, we focus on producing a faithful translation that corresponds closely to the original proof. The faithful translation still benefits from the inherent readability and robustness of structured Isar, and it can also serve as a starting point for further readability improvements made either by the user or by additional tools. Notably, our translation approach explicitly clarifies the logical flow of the original proof, which we believe is useful information for additional refactoring passes. We discuss potential extensions of our work in this direction in Section 6.

Given the apply-style proof of lemma `apply_prf` presented above, `Apply2Isar` produces the following structured proof:

[4] Isabelle includes options such as `unify_trace` that, when enabled, can help debug these apply-style proofs, but we emphasize that this requires further manual interaction and some expertise – our point is that the source of the error is not immediately evident from the error message and proof text.

```
proof(-)
  have h_3_1: "C" by (rule L4)
  have h_2_2: "B2" by (rule L3) (fact h_3_1)
  have h_2_1: "B1" by (rule L2)
  show h_1_1: "A" by (rule L1) (fact h_2_1, fact h_2_2)
qed
```

One benefit of structured Isar is that it can express complex fact dependencies in an organized and explicit way. In the translated proof, each goal has been labeled (in the form "`h_x_y`") so that it can be referenced later in the proof.[5] This is necessary because some methods can create new goals. For instance, `rule L1` transforms the goal `[ A ]` to the goals `[ B1, B2 ]`. So, to prove `h_1_1: "A"` in the Isar proof, we first apply `rule L1`, then discharge the resulting goals `B1` and `B2` with `fact h_2_1` and `fact h_2_2` respectively.

For users who find the fact labels too verbose, we provide the `named_facts` option. If disabled, no labels are generated. Helpfully, the `fact` method can be invoked with no arguments, in which case it automatically proves the current goal by searching the context for a matching fact. Thus, when `named_facts` is disabled, we instead invoke `fact+` to discharge the resulting goals after each method application (the `+` combinator applies a method one or more times). To avoid clutter and unnecessary detail, we present the rest of the examples in this paper without fact labels.

We implement `Apply2Isar` in Isabelle/ML, which allows it to be tightly integrated with Isabelle's interactive environment. `Apply2Isar` first uses functions from the Isabelle/ML ecosystem to tokenize and parse its input (an entire apply-style proof) into a custom AST. Then, `Apply2Isar` crawls this AST to replay each proof command and collect the intermediate goals into a second AST that represents Isar elements. Finally, `Apply2Isar` processes this second AST into a string that gets printed to the output window as a clickable element. When clicked, the translated proof is automatically inserted into the proof document.[6] `Apply2Isar` takes the input proof in a pair of cartouches, which lets the user specify exactly which portion of the proof should be translated:

```
lemma "[...]"
  using [...] unfolding [...] (* The user can decide whether to include these *)
apply2isar‹
  apply m1
  [...]
  by auto›
```

While Isabelle/HOL can optionally produce proof terms, they slow down proof checking and are disabled by default [5]. `Apply2Isar` does not work with proof terms or interact directly with the Isabelle kernel. Instead, it replays each proof command by invoking the corresponding Isabelle/ML function. This level of abstraction is appropriate for `Apply2Isar` because we intend it to produce structured Isar that faithfully reflects the intermediate goals in the original apply-style proof, which is data that is not evident in the final proof term.

[5] Note that if these fact labels shadow existing lemma names, the proof may break. Thus, we provide the `fact_name_prefix` setting to allow users to change the "`h`" prefix to a custom string.

[6] This is the same functionality used by Sledgehammer, which allows the user to insert a proof found by an external solver [7].

Table 1 Proof commands that we handle.

`apply`	Applies a method to the proof state, yielding a new proof state.
`by`	Similar to apply; ends a (sub)proof if the method closes all goals with backtracking.
`done`	Ends a proof if there are no more goals.
`.`	Immediate proof; ends proof if facts in proof state match goals (with unification).
`..`	Equivalent to `by standard`.
`using`	Introduces facts into the proof state.
`unfolding`	Unfolds definitions in all goals and all facts in the proof state.
`subgoal`	Begins a subproof of the first goal in the proof state.
`prefer`	Moves a goal to the first position.
`defer`	Moves a goal to the last position.
`back`	Yields another possible state from the last method application.
`supply`	Introduces existing facts with an optional name.
`proof`	Begins an Isar proof for all current goals.
`sorry`	Proof placeholder; proves all current goals by cheating.

4 Implementation Challenges

Isabelle is intentionally very flexible in both its procedural and declarative styles, but there are significant structural differences between apply-style and structured Isar proofs. `Apply2Isar` supports the most frequently used proof commands; Table 1 provides a full list of supported commands and their definitions. In this section, we explain important design decisions behind `Apply2Isar` and highlight some of the main challenges we encountered.

4.1 Operations on Multiple Goals

Some methods such as `auto`, `safe`, or `simp_all` can simultaneously affect multiple goals in the proof state. For instance, consider:

```
[...]          (* goals [ A1, B1, C1, D1 ] *)
   apply auto  (* modifies A1 and C1; proves B1; does not modify D1 *)
[...]          (* goals [ A2, C2, D1 ] *)
```

A simple way to handle this is to always print all goals in the current proof state at every step. This will even print goals that remain unchanged by a command:

```
have "A2" "C2" "D1" by [...]
have "A1" "B1" "C1" "D1" by (auto) fact+
```

In this translation snippet, the goal `D1` is carried over from the first `have` to the second, getting discharged the second time by `fact+`. This approach essentially treats the apply-style proof as a linear sequence of proof state manipulations. While this generates valid Isar that mimics the structure of the original apply-style proof very closely, it is very verbose and does not clearly indicate the *purpose* of each command. For instance, consider the following example where ten goals are discharged one at a time (one may encounter such a scenario, e.g., when proving locale instantiations):

```
                  (* goals: [ A1, A2, A3, ..., A9, A10 ] *)
        apply m1  (* goals: [ A2, A3, ..., A9, A10 ] *)
       apply m2   (* goals: [ A3, ..., A9, A10 ] *)
      [...]
 apply m9         (* goals: [ A10 ] *)
by m10            (* goals: [ ] *)
```

Following the linear translation approach just described, `Apply2Isar` will generate:

```
have "A10" by m10
have "A9" and "A10" by (m9) fact+
[...]
have "A2" and "A3" and [...] and "A9" and "A10" by (m2) fact+
show "A1" and "A2" and "A3" and [...] and "A9" and "A10" by (m1) fact+
```

There is a lot of unnatural repetition here – intuitively, we might expect `m1` to prove `A1`, `m2` to prove `A2`, and so forth.

Thus, we implement a second approach, which we call `smart_goals`, that removes some of this redundancy by only printing the goals that change after each command. For instance, `Apply2Isar` can instead translate the previous example to:

```
have "A10" by m10
have "A9" by m9
[...]
have "A2" by m2
have "A1" by m1
show "A1" and "A2" and [...] and "A9" and "A10" by fact+ (* Final combined show *)
```

As the goals are proven piecemeal, `Apply2Isar` combines them in a final `show` line at the end of the Isar proof.[7]

4.2 Operations that Complicate Proof Flow

In addition to manipulating goals, commands may also change the proof state by focusing a single goal or by introducing additional facts.

The current proof state when `Apply2Isar` is invoked may already contain facts introduced by `using`, `from`, etc. As the structured Isar proof must be able to refer to those facts explicitly, `Apply2Isar` collects them and prints them as a named assumption in the translation.

In an apply-style proof, the `supply` command allows one or more existing lemmas to be introduced and given a local name. For example:

```
lemma "True" "P ⟹ P ∨ Q"
   supply myThms = TrueI disjI1 (* myThms = [ True, ?P ⟹ ?P ∨ ?Q ] *)
   apply (rule myThms(1))
  by (rule myThms(2))
```

The corresponding structured command for naming existing facts is `note`, so `Apply2Isar` converts the previous proof to:

```
lemma "True" "P ⟹ P ∨ Q"
proof(-)
  note myThms = TrueI disjI1
  have "P ⟹ P ∨ Q" by (rule myThms(2))
  have "True" by (rule myThms(1))
  show "True" "P ⟹ P ∨ Q" by fact+ (* Final combined show *)
qed
```

[7] In this example, every `have` could be replaced by `show`, which would remove the need for the final combined `show`. However, this does not work in general: `m1` could have simultaneously proven `A1` along with some other non-top-level goal `B`, in which case `show "A1" and "B" by m1` would cause an error.

There is something peculiar here: although the order of the original proof was reversed (so that `rule myThms(2)` got applied first), the `note` command and the corresponding `supply` command are *both* at the beginning of the proof. This is because if the `note` command came after the two `rule` applications, `myThms` would have been referenced before it was defined. Our solution to this, as seen in the prior example, is to always lift generated `note` commands to the beginning of the proof. Note that the order of multiple `note` commands must be preserved, since later `supply` commands can reference previous ones.

Finally, the `subgoal` command focuses the first goal, saving the remaining goals in a proof context. When the subproof is finished, the remaining goals are re-introduced from the proof context. Additionally, just as `supply` introduces a local name for existing lemmas, `subgoal` can give a name to the local fact it proves. For example:

```
(* goals: [ P, P ∨ Q ] *)
subgoal myFact
  (* goals: [ P ] *)
  apply [...]
done
(* goals: [ P ∨ Q ] *)
by (rule disjI1[OF myFact])
```

`Apply2Isar` handles the `subgoal` command by opening a structured subproof:

```
have myFact: "P"
proof(-)
  [...]
qed
```

Note that an unnamed `subgoal` can still be referenced as a fact literal. Just as with `supply`, this presents a fact-ordering problem: naïvely reversing the order of the commands in the original proof results in facts getting referenced before they are proven. `Apply2Isar` handles this by moving all subproofs to the beginning of the Isar. Because subproofs are standalone proofs, we are not in danger of producing a broken proof, so long as we maintain the order of `subgoal` and `supply` commands (since a subproof or `supply` may reference a previous `subgoal` or `supply` fact). Since subproofs can be nested (and `supply` commands can appear inside subproofs), we recursively move subproofs (and `note` commands) in the translation to the top of the structured Isar proof block that contains them.

This solution produces logically valid Isar, but it is not fully faithful to the logical flow of the original proof. For greater readability, one may want the original proof's flow to be captured in the translation. Thus, we implement a `dummy_subproofs` option: when enabled, every moved subproof will be replaced with a "dummy" which proves the same statement by a `fact` invocation. The resulting translation remains more faithful to the original proof, at the cost of being repetitive. For instance, with `dummy_subproofs` enabled, the previous example will yield the following Isar:

```
have myFact: "P"
proof(-)
  [...]
qed
have "P ∨ Q" by (rule disjI1[OF myFact])
have "P" by fact (* Subproof dummy *)
show "P" "P ∨ Q" by fact+  (* Final combined show *)
```

4.3 Interactions Between Commands

In an apply-style proof, the `using` command introduces new facts to the proof state after the goal has already been stated. These facts can then be used in the next proof step, such as the next method invocation via the `apply` or `by` commands. The `unfolding` command unfolds specified definitions in all goals *and* all facts in the proof state (including those introduced by `using`). For example:

```
lemma unfolding_and_using_ex:
  fixes P defines "R ≡ P"
  assumes 1: "P" and 2: "R ⟶ Q" shows "Q" "R"
  using 1 2       (* goals: [ Q, R ] | using: [ P, R ⟶ Q ] *)
  unfolding R_def (* goals: [ Q, P ] | using: [ P, P ⟶ Q ] *)
   apply simp     (* goals: [ P ]    | using: [] *)
  by (rule 1)
```

Because `unfolding` can simultaneously affect all goals *and* all facts in the proof state, it requires some care to handle, particularly in its interaction with `subgoal`. We considered two approaches: one that works generally but introduces repetitiveness, and one that avoids repetitiveness but cannot be used when `unfolding` immediately precedes `subgoal`.

The first approach is to treat `unfolding` similarly to `apply`: for each `unfolding` command, `Apply2Isar` creates a `have` statement to capture that command's effect on the goals:

```
proof(-)
  have "P" by (rule 1)
  have "Q" using 1 2 unfolding R_def by simp (* Repetitive unfolding after using *)
  have "R" unfolding R_def by fact+ (* Captures effects of unfolding on goals *)
  show "Q" and "R" by fact+ (* Combined show *)
qed
```

There is some repetition here: in the original proof, `unfolding R_def` is used after `using 1 2`, so it must appear a second time in the translated proof to capture its effects on facts `1` and `2`. In an attempt to avoid this repetition, we implement a second (perhaps more natural) approach under the option `smart_unfolds`. This approach groups the effects of a sequence of `unfolding`/`using` commands with the next `apply` command.[8] For instance, `Apply2Isar` can translate the above example as follows, where the effects of `unfolding R_def` are combined with those of `apply simp`:

```
proof(-)
  have "P" by (rule 1)
  show "Q" and "R" using 1 2 unfolding R_def by (simp) fact+
qed
```

Unfortunately, this second approach cannot be used when `unfolding` immediately precedes `subgoal`; because goals focused by `subgoal` can be used later in a proof, these goals must be present verbatim in the corresponding Isar. Additionally, if a `using` command precedes the `unfolding` command (i.e., we have a `using`-`unfolding`-`subgoal` sequence), then the facts introduced by `using` will be used in the first method application *inside* the `subgoal` subproof, and we must still ensure those facts are properly unfolded. Thus, in this situation, `Apply2Isar` reverts to the first approach since it correctly duplicates the `unfolding` steps in the translation.

[8] Or the next `apply`-like command (`by`, immediate proof (`.`), and default proof (`..`)).

4.4 Shadowing

The `subgoal` command, introduced in Section 4.2, skolemizes all meta-quantified variables in the goal it focuses. The user can use the `for` keyword to choose names for some or all of the skolemized variables; any variables left unnamed are given an internal representation which cannot be explicitly referenced. In structured Isar, this corresponds to the `fix` keyword.

Importantly, the user is allowed to pick names that shadow existing variables. Shadowing is sometimes natural, for example in induction proofs. However, it can cause difficulties for `Apply2Isar` because Isabelle will use the same name for both the shadowed and the newly-skolemized variable in the output window, differentiated only by metadata (e.g., syntax highlighting). For example:

```
(* goals: [ ⋀x. P x y ]*)
subgoal for y
  (* goals: [ P y y ] (!!! These two y's are different!) *)
  apply [...] done
done
```

If we print `have "P y y"` in the translation, Isabelle interprets both occurrences of `y` as the skolem variable. Thus, `Apply2Isar` can ensure that fresh names are always used:

```
have "⋀x. P x y"
proof(-)
  fix ya (* Renamed local variable *)
  [...]
  show "P ya y" by [...]
qed
```

Unfortunately, this does not work in all cases. To see why, suppose `x` is an existing free variable in the proof context when `subgoal` is invoked in the following snippet:

```
subgoal for x
  by (simp add: some_lemma[where y=x])
```

As the newly-fixed variable `x` is explicitly referenced in the facts given to the `simp` method, renaming the skolemized `x` would also require renaming it in the `simp` invocation. Doing this would require parsing each method in detail, which we do not implement in `Apply2Isar` because we use the existing Isabelle/ML functions for parsing methods directly to the data expected by the Isabelle/ML function for `apply`.

Because this variable renaming behavior may need to be enabled or disabled depending on the situation, we implement the `subgoal_fix_fresh` option. By default, this option is disabled, which means `Apply2Isar`'s default behavior can handle "responsible" shadowing, where there are no goals containing the shadowed variable. Enabling `subgoal_fix_fresh` lets `Apply2Isar` handle "irresponsible" shadowing, at the cost of failing when any renamed variables are explicitly referenced.

4.5 Schematic Variables in Goals

In an Isabelle term, *schematic variables* are variables which can be instantiated by higher-order unification [26].[9] In inferences, terms with meta-quantified variables (e.g., `⋀x. P x`) are automatically converted into their schematic counterparts (e.g., `P ?x`) [25]. In an apply-style

[9] Technically, a term with schematic variables is called a *pattern* [26].

proof, a method application may yield a schematic goal, where the schematic variables represent unknowns that will be instantiated later in the proof. For instance, consider the following proof, where the fact `h1` becomes a schematic fact:

```
lemma
  assumes h1: "⋀x y. P x ⟹ Q y" and h2: "P a" shows "Q b"
  (* h1: P ?x ⟹ Q ?y *)
  apply (rule h1) (* goals: [ P ?x ] *)
  by (rule h2)
```

When `rule h1` is applied, Isabelle unifies the conclusion of `h1`, namely `Q ?y`, with the goal, namely `Q b`, by instantiating `?y` to `b`. However, unification has no need to instantiate `?x` here, so it is left schematic and the new goal is `P ?x`. This goal can be discharged by instantiating `?x` to *any* `x` such that `P x`. In this case, instantiating `?x = a` does the trick, and unification again does this automatically when `rule h2` is applied.

Schematic variables in goals are essentially free variables that will be instantiated by some future method application. However, `have` does not allow schematic goals. Thus, `Apply2Isar` avoids translating any portions of an apply-style proof that contain schematic goals. When a schematic goal appears, `Apply2Isar` collects all proof commands until there are no more schematic goals. In the generated Isar, these collected commands are then printed as an apply-style proof under a single `have` statement. As this leaves some apply-style proofs in the translation, we consider such proofs "partially translated" in our evaluation. We discuss possible alternative solutions in Section 6.

4.6 Types

Until now, we have been presenting examples without any type annotations. While type inference often means type annotations are unnecessary, this is not always the case. In the following example, the first lemma is not provable because `x` is only inferred to have type `'a :: {zero,ord}`, but the second lemma with type annotations is provable:

```
lemma not_provable: "∀x. 0 ≤ x" oops
lemma provable: "∀x :: nat. 0 ≤ x" by simp
```

While type information in an apply-style proof is maintained across proof states, it is lost if terms are printed without annotations (and the types cannot be fully recovered by inference). To address this, we provide the `print_types` option. When set to `none`, `Apply2Isar` will print no types; when set to `all`, `Apply2Isar` will print all type annotations; when set to `necessary`, `Apply2Isar` will only print type annotations when they are required. `Apply2Isar` checks if type annotations are required for a goal by printing it without type annotations, re-parsing the result, then checking if the re-parsed goal equals the original goal (modulo α-equivalence).

4.7 Proof Exploration and Partial Proofs

While writing an apply-style proof, a user might encounter goals which they would rather close with structured Isar. Thus, `Apply2Isar` supports commands for proof exploration, partial apply-style proof, and mixed procedural-declarative proofs.

The `defer` and `prefer` commands can help with proof exploration by reordering the set of goals [26]. These steps must be replayed as we process the original proof, but they do not correspond to any structured steps because structured facts are explicitly asserted and referenced. Thus, the translated proof will simply state the facts in the new order with the appropriate labels without any explicit command being required.

The `back` command also helps with proof exploration. In an apply-style proof, `apply m` applies the method `m` to obtain a sequence of potential next states (or fails if `m` could not refine the goal) [26]. By default, the first state in the sequence is used, but the user can request other possibilities with the `back` command, which backtracks and returns the next state in the sequence. This is useful for proof exploration, but is very brittle and therefore explicitly banned in AFP entries [1]. An example of `back` from the Isabelle tutorial is the following [21]:

```
lemma " ⟦ x = f x; T (f x) (f x) x ⟧ ⟹ T x x x"
  apply (erule ssubst) (* goals: [ T (f x) (f x) x ⟹ T (f x) (f x) (f x) ] *)
  back                 (* goals: [ T (f x) (f x) x ⟹ T x (f x) (f x) ]     *)
  back                 (* goals: [ T (f x) (f x) x ⟹ T (f x) x (f x) ]     *)
  back                 (* goals: [ T (f x) (f x) x ⟹ T x x (f x) ]         *)
  back                 (* goals: [ T (f x) (f x) x ⟹ T (f x) (f x) x ]     *)
  apply assumption     (* goals: []                                        *)
  done
```

`Apply2Isar` automatically removes `back` commands in its translation of this proof:

```
lemma " ⟦ x = f x; T (f x) (f x) x ⟧ ⟹ T x x x"
proof(-)
  have "T (f x) (f x) x ⟹ T (f x) (f x) x" by (assumption)
  show "x = f x ⟹ T (f x) (f x) x ⟹ T x x x" by (erule ssubst) fact+
qed
```

This works because the `by` command is a terminal method application – it will apply the given method(s) and automatically backtrack until a completely solved state is found (or fail or time out if a solved state cannot be found). Thus, by making intermediate goals explicit, `Apply2Isar` can remove all `back` commands since the necessary backtracking is automatically handled by each `by` command in the translation.

`Apply2Isar` also supports the `sorry` command, which immediately proves all current goals by cheating – that is, it is a proof placeholder. This allows the user to close difficult goals in an apply-style proof with `sorry`, then convert the proof to structured Isar before finishing it. Additionally, if a method application fails during our proof reconstruction, we treat it instead as a `sorry`. This allows broken apply-style proofs to be partially converted to structured Isar.

Additionally, the user can switch directly from an apply-style proofs to structured Isar:

```
lemma "P" "Q" "R" (* goals: [ P, Q, R ] *)
    apply m1 (* goals: [ Q, R ]    *)
proof(-)
  [...]
  show "Q" "R" [...]
qed
```

This pattern is considered brittle and is flagged by the Isabelle linter [19]. Once a structured Isar proof is opened, it must prove all current goals – it is "terminal" in this sense, like `by`. If a structured proof is initiated inside a `subgoal` subproof with the `proof` command, the associated `qed` closes the subproof like `by` and `done`. Structured Isar syntax is far more complex than apply-style syntax, so `Apply2Isar` avoids parsing and replaying such parts of a proof; instead, it simply assumes that it succeeds and copies it verbatim. While `Apply2Isar` maintains some of the user's formatting, comments are unfortunately lost during tokenization.

In `Apply2Isar`'s proof replay, we run the `sorry` command to replicate the effects of switching to Isar – this is necessary to ensure that the proof replay correctly closes subproofs that are concluded in the original apply-style proof with structured Isar. Running `sorry` in

our proof replay is not problematic, since the final generated proof is independently checked by the Isabelle kernel. However, it does mean that `Apply2Isar` will dutifully reproduce even broken structured Isar within an apply-style proof. We view this side-effect of our implementation as a feature that supports proof exploration and partial proof translation.

5 Evaluation

We tested `Apply2Isar` with `Isabelle2025-2`. We constructed our benchmark set by selecting thousands of apply-style proofs from entries in the *Archive of Formal Proofs* (AFP). Because many AFP entries contain relatively few apply-style proofs (in part due to the AFP's general preference for structured proofs), building our benchmark set by randomly sampling AFP entries would not be very effective. Instead, we selected the five AFP entries with the most occurrences of "`apply`" – this is a simple heuristic to identify entries that contain many apply-style proofs.[10] These five entries are: `Group-Ring-Module` [15], `AutoCorres2` [9], `Flyspeck-Tame` [3], `Valuation` [14], and `BNF_Operations` [8], each of which contains multiple theory files. As `AutoCorres2` is an extremely large entry, we focus on the ten theory files that load in the primary session with the most occurrences of "`apply`"; for all other entries, we test every theory file. For each theory file that we tested, we ran `Apply2Isar` on every proof in the file containing at least one `apply` command. We test the entirety of every proof, including any `unfolding` or `using` commands that may precede the first `apply` command. To aid in batch testing, we wrote a wrapper in Isabelle/ML that processes an entire theory file and wraps every apply-style proof in `apply2isar‹[...]›` blocks. Our final benchmark set contained 4461 apply-style proofs. Our test bench had an AMD Ryzen 7 2700X 8-core CPU and 32 GiB of RAM. We do not keep track of precise timing information since our primary concern is the number of apply-style proofs that `Apply2Isar` successfully converts to structured Isar. However, in our evaluation, we limited `Apply2Isar` to a 30-second timeout to ensure that it is reasonably performant.

We present our evaluation results in Table 2, with a visualization in Figure 1. In Table 2, the "**Entry**" column indicates the AFP entry, and the "**Total Proofs**" column indicates the total number of apply-style proofs we tested in the corresponding entry. We tested `Apply2Isar` with our choice of default options: `named_facts`, `smart_unfolds`, and `smart_goals` enabled; `print_types` set to "necessary"; and all other options disabled. The "**Complete Translations**" column indicates the number of proofs in the entry that `Apply2Isar` successfully translated to working Isar.

The number of proofs that were only partially translated to Isar due to schematic goals, as discussed in Section 4.5, is indicated by the "**Partial Translations**" column. We observed that most of `Apply2Isar`'s partial translations are still satisfactory. Many partial translations involve extremely long apply-style proofs with only short stretches of commands where schematic goals are present. With these proofs, the resulting translation is still primarily structured Isar, with only certain steps requiring short apply-style proofs. There were a few test cases where the majority of the proof contained schematic goals, resulting in translations that change very little, but these cases appear to be quite rare in our benchmark set.

We observed that many of `Apply2Isar`'s failures are caused by inconsistencies in Isabelle's term-printing mechanism. In general, printing goals and re-parsing them is not guaranteed to succeed in Isabelle; the re-parsing may result in a different term or simply cause a syntax error.

[10] We simply `grep`'d for "`apply`" in each entry, which counts spurious occurrences of "`apply`" such as those in definitions and lemma names. However, since we only need a rough heuristic, this is acceptable.

Table 2 Data from our evaluation of `Apply2Isar`'s complete translations, partial translations, and failures.

Entry	Total Proofs	Complete Translations	Partial Translations	Fails		
				Print Fails	Timeout Fails	Misc. Fails
`Group-Ring-Module`	2387	2364	16	2	3	2
`AutoCorres2`	952	856	52	33	0	11
`Flyspeck-Tame`	561	377	182	1	1	0
`Valuation`	384	368	2	1	10	3
`BNF_Operations`	177	49	124	1	3	0

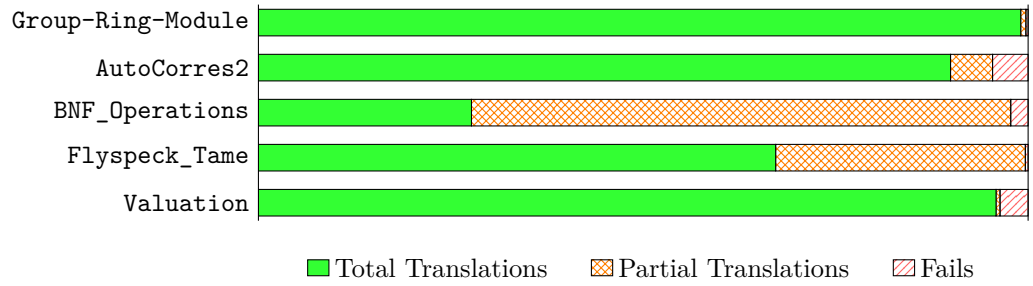

Figure 1 A visualization of our evaluation data. The solid green (leftmost) segment of each bar indicates the portion of complete translations; the crosshatched orange (middle) segment indicates the portion of partial translations; and the single-hatched red (rightmost) segment indicates the portion of failures.

In fact, this can occur even when manually copying goals from the output window, so this is not a problem localized to our implementation. While this is a rare occurrence, we primarily observed it in our benchmark set with certain goals that use custom syntax. We hope to see further Isabelle development toward improving this round-trip reliability,[11] since we do not see any general solution for this failure case beyond addressing this underlying cause. In Table 2, the "**Fails**" column indicates the number of proofs in the entry for which `Apply2Isar` did not produce working Isar – the "Print Fails" subcolumn indicates the number of failures due to inconsistencies in Isabelle's term-printing mechanism; the "Timeout Fails" subcolumn indicates the number of timeouts; and the "Misc. Fails" subcolumn indicates the number of all other failures. We observed that many of the timeouts are caused by extremely slow re-parsing of certain goals (as described in Section 4.6, we re-parse goals when `print_types` is set to "necessary"). In these cases, we observed that manually copying the offending goals into a `have` statement resulted in extremely slow parsing by the `have` command itself, so we do not see a general solution to these timeouts. The sporadic miscellaneous failures do not appear to fit into a single category, though we observed that they typically involve edge-case behavior of uncommon methods that succeed in the original apply-style proof but fail in the translation. Notably, proof length did not seem to cause failures. Indeed, the longest apply-style proof successfully handled by `Apply2Isar` contained over 350 commands.

[11] Cross-reference the following 2024 discussion on the Isabelle Users Mailing List: `https://lists.cam.ac.uk/sympa/arc/cl-isabelle-users/2024-11/msg00006.html`.

6 Future and Related Work

As discussed in Section 4, translating apply-style proofs to structured Isar is far from straightforward. Here, we discuss some high-hanging fruit which could improve robustness or provide additional convenience to the user. We also examine related work and discuss other features that might be useful to integrate into `Apply2Isar`.

Our current handling of shadowed variables could be improved. As discussed in Section 4.4, renaming variables in a `subgoal` command to avoid shadowing will break commands in the subproof that explicitly refer to the old name. While the `subgoal_fix_fresh` option allows the user to disable this renaming, it requires manual intervention and does not succeed in all cases. Handling all potential edge cases would require renaming user-chosen names as well as all explicit references to them; but we expect implementing this robustly would require more intricate parsing of individual methods. As it stands, this problem only appears a handful of times in our benchmark set.

It would also be possible to improve our handling of schematic goals, as discussed in Section 4.5. One approach would be to obtain the instantiations made during the proof, which could be done by examining proof terms. Alternatively, Isabelle provides the `schematic_goal` theory command, which is similar to `lemma` except that it allows goals to be schematic. The final theorem produced by `schematic_goal` is thus determined by the instantiations made by the proof. It is possible that this mechanism could be adapted to allow `Apply2Isar` to handle schematic goals more nicely.

Additionally, we do not currently support custom commands. While custom commands did not appear in our benchmark set, some specialized libraries use them (e.g. [16, 17]). While it would be difficult to support custom commands – which could do almost anything – in a principled way, we could potentially support bypassing them (similar to our handling of schematic goals; see Section 4.5).

In 2012, Wiedijk developed a proof language called `miz3` for HOL Light that combines the procedural and declarative styles [29] based on the language for the Mizar system [20]. Wiedijk also implemented `miz3_of_hol`, an automatic translation to `miz3` from standard procedural HOL Light proofs.[12] We intend for `Apply2Isar` to convert procedural proofs to declarative ones entirely within the existing Isabelle/Isar framework; as structured Isar has been very widely adopted by Isabelle users, we do not strive to develop a new declarative (or declarative-procedural hybrid) proof language. While the high-level approach of `miz3_of_hol` is similar to our approach with `Apply2Isar`, HOL Light is designed to be a clean and simple system [11], and this makes the implementation of `miz3_of_hol` simpler than that of `Apply2Isar`; as discussed in Section 4, much of our implementation effort focused on correctly handling the various Isabelle/Isar proof commands and their interactions. For instance, the `miz3_of_hol` implementation represents a procedural proof step using a datatype with two constructors (to represent a possibly-nested tactic), whereas our corresponding datatype uses fifteen constructors (to represent the various procedural proof commands in Isabelle/Isar). Our proof replay must therefore handle these fifteen cases and account for the various interactions between these cases.[13]

[12] The code for `miz3` and `miz3_of_hol` is included in the HOL Light distribution: `https://github.com/jrh13/hol-light/tree/master/miz3`.

[13] Table 1, which lists the commands we handle, only contains fourteen entries; the additional constructor is to account for the possibility of facts being introduced into the proof context (e.g., via `using` or `from`) prior to the invocation of `Apply2Isar`, as discussed in Section 4.2.

In 2015, Adams developed the Tactician tool for Rocq to pack (resp. unpack) chained sequences of tactics into (resp. out of) a single proof step [2]. Similar ideas were explored by Magaud to convert entire Rocq proof scripts to a single step [18]. In 2025, Shi et al. explored "deautomation" in Rocq [23] – that is, automatically refactoring a highly complex proof involving automated steps into a simpler proof involving more "primitive" steps. Notably, they conducted a user study in which participants expressed their desire for deautomation due to the difficulties of working with heavily automated proofs. As Isabelle also allows users to construct complex methods via combinators, integrating deautomation into `Apply2Isar` could prove useful. However, deautomation relies on inspecting the behavior of individual methods. Since Isabelle/HOL's automation is quite "black-box", implementing such a feature could prove difficult. Even splitting up a method sequence like `apply (m1, m2)` into two separate `apply` commands is not necessarily straightforward in light of possible interactions with other commands like `using`. However, Kappelmann's recent Zippy tool, which implements white-box automation for Isabelle [12, 13], may be helpful.

In 2022, Megdiche et al. developed an Isabelle linter which flags bad style including (but not limited to) complex steps (involving many combined methods), complex initial and terminal Isar methods, outdated methods and brittle commands (like `back`), and switching to structured Isar in an apply-style proof [19]. Note that the AFP automatically rejects submissions that do not pass certain linter checks [1]. The linter works syntactically on the proof document itself – unlike `Apply2Isar`, it does not involve replaying proofs or examining the proof state. While it formerly had some automatic refactoring capability, this feature only worked on syntactic anti-patterns and was ultimately removed. One notable aspect of the Isabelle Linter is its integration with the Isabelle/PIDE document model [27]. Currently, `Apply2Isar` is implemented entirely within the Isabelle/ML ecosystem, but better integration with the Isabelle/PIDE framework could improve the user experience.

Haftmann's Sketch and Explore tools can print an Isar proof skeleton after a single method application, optionally running a specified method or Sledgehammer on each of the new goals [10]. Building on this, Tan et al. developed Super Sketch, which aims to offer more fine-grained automation options [24]. Both tools are intended for proof exploration, and not for refactoring existing proofs. In addition to Super Sketch, Tan et al. also developed Super Fix, a tool for making small proof fixes (e.g., one-line changes) in bulk. Toward producing more natural translations, it may be worthwhile to integrate a tool like Sketch into `Apply2Isar`; for instance, specific methods in the apply-style proof, such as `induct`, could translate to a structured subproof whose skeleton is generated by Sketch. However, we believe this will require careful handling of multiple subgoals, making the integration nontrivial.

As part of their AutoCorrode project, AWS Labs has very recently launched Isabelle Assistant, which integrates LLMs into the Isabelle interactive environment to perform many tasks including proof refactoring [4]. As one of the Isabelle Assistant features, the user can ask it to convert an apply-style proof to structured Isar. Though we have not thoroughly evaluated Isabelle Assistant's effectiveness at this task, our initial impression is that it can be slow and unreliable in certain situations. For instance, we found that Isabelle Assistant with Anthropic's Claude Opus 4.6 LLM failed to convert the following proof (taken from the Isabelle/HOL library) to structured Isar:

```
lemma (in monoid) division_weak_partial_order: (* From HOL-Algebra.Divisibility *)
  "weak_partial_order (division_rel G)"
  apply unfold_locales
        apply (simp_all add: associated_sym divides_antisym)
    apply (metis associated_trans)
   apply (metis divides_trans)
  by (meson associated_def divides_trans)
```

In this example, the `unfold_locales` invocation yields seven subgoals, and the `simp_all` invocation proves four of those subgoals and modifies the others. The effects of the `simp_all` invocation appear to have confused the LLM, leading it to produce a broken proof. On the other hand, `Apply2Isar` can handle this proof without issue. It is worth noting, however, that when Isabelle Assistant does succeed, its translations are arguably more "natural" than those produced by `Apply2Isar`. Despite this, we view the algorithmic approach of `Apply2Isar` as a strong benefit – the output of `Apply2Isar` is predictable and does not impose stylistic choices onto the user (and as discussed in Section 3, "naturalness" is a subjective metric). `Apply2Isar` also has the advantage that it can be run locally on moderate hardware, rather than requiring a subscription to a cloud provider. While we believe there are many virtues in taking an algorithmic approach, it may be interesting to explore how LLM-based tools could make the output of `Apply2Isar` more natural and human-readable. Notably, the fact labels printed by `Apply2Isar` make the fact dependency tree of the structured proof explicit and follow a uniform naming scheme, which we believe could be valuable information for an LLM-based tool.

7 Conclusion

Our tool `Apply2Isar`, which translates apply-style proofs to structured Isar, works well in practice and achieves our goal of translating a wide range of apply-style proofs. `Apply2Isar` aims to improve the readability, maintainability, and robustness of Isabelle/HOL developments. We envision this being useful both in maintaining existing developments and in supporting active proof developments. We believe `Apply2Isar` is a valuable addition to the Isabelle/HOL toolbox, and we look forward to future improvements and new features.

References

1 Archive of formal proofs. `https://www.isa-afp.org/`.

2 Mark Adams. Refactoring proofs with Tactician. In Domenico Bianculli, Radu Calinescu, and Bernhard Rumpe, editors, *Software Engineering and Formal Methods - SEFM 2015 Collocated Workshops: ATSE, HOFM, MoKMaSD, and VERY*SCART, York, UK, September 7-8, 2015, Revised Selected Papers*, volume 9509 of *Lecture Notes in Computer Science*, pages 53–67. Springer, 2015. `doi:10.1007/978-3-662-49224-6_6`.

3 Gertrud Bauer and Tobias Nipkow. Flyspeck I: Tame graphs. *Archive of Formal Proofs*, May 2006. Formal proof development. URL: `https://isa-afp.org/entries/Flyspeck-Tame.html`.

4 Hanno Becker, Nathan Chong, Robert Dockins, Jim Grundy, Jason Z. S. Hu, Ike Mulder, Dominic P. Mulligan, Paul Mure, Lawrence C. Paulson, and Konrad Slind. AutoCorrode software verification framework for Isabelle/HOL. `https://github.com/awslabs/autocorrode`, 2025.

5 Stefan Berghofer and Tobias Nipkow. Proof terms for simply typed higher order logic. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics*, pages 38–52, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. `doi:10.1007/3-540-44659-1_3`.

6 Sage Binder, Hanna Lachnitt, and Katherine Kosaian. Apply2Isar. Software, swhId: `swh:1:dir:8753f0c693cdc7159375110e6dd9d977e62984a8` (visited on 2026-07-13). URL: `https://github.com/sagebinder/apply2isar`, `doi:10.4230/artifacts.27121`.

7 Jasmin Blanchette, Martin Desharnais, Lawrence C. Paulson, and Lukas Bartl. *Hammering Away: A User's Guide to Sledgehammer for Isabelle/HOL*, January 2026. User's Guide. URL: `https://isabelle.in.tum.de/doc/sledgehammer.pdf`.

8 Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. Operations on bounded natural functors. *Archive of Formal Proofs*, December 2017. Formal proof development. URL: https://isa-afp.org/entries/BNF_Operations.html.

9 Matthew Brecknell, David Greenaway, Johannes Hölzl, Fabian Immler, Gerwin Klein, Rafal Kolanski, Japheth Lim, Michael Norrish, Norbert Schirmer, Salomon Sickert, Thomas Sewell, Harvey Tuch, and Simon Wimmer. AutoCorres2. *Archive of Formal Proofs*, April 2024. Formal proof development. URL: https://isa-afp.org/entries/AutoCorres2.html.

10 Florian Haftmann. Sketch_and_Explore. Isabelle Distribution. URL: https://isabelle.in.tum.de/library/HOL/HOL-ex/Sketch_and_Explore.html.

11 John Harrison. HOL Light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 60–66, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. doi:10.1007/978-3-642-03359-9_4.

12 Kevin Kappelmann. Zippy – generic white-box proof search with zippers, 2025. doi:10.48550/arXiv.2503.20413.

13 Kevin Kappelmann. Zippy - generic white-box proof search with zippers. *Archive of Formal Proofs*, November 2025. Formal proof development. URL: https://isa-afp.org/entries/Zippy.html.

14 Hidetsune Kobayashi. Fundamental properties of valuation theory and Hensel's lemma. *Archive of Formal Proofs*, August 2007. Formal proof development. URL: https://isa-afp.org/entries/Valuation.html.

15 Hidetsune Kobayashi, L. Chen, and H. Murao. Groups, rings and modules. *Archive of Formal Proofs*, May 2004. Formal proof development. URL: https://isa-afp.org/entries/Group-Ring-Module.html.

16 Peter Lammich. The imperative refinement framework. *Archive of Formal Proofs*, August 2016. Formal proof development. URL: https://isa-afp.org/entries/Refine_Imperative_HOL.html.

17 Peter Lammich. Refinement to imperative HOL. *Journal of Automated Reasoning*, 62(4):481–503, April 2019. doi:10.1007/s10817-017-9437-1.

18 Nicolas Magaud. Towards automatic transformations of Coq proof scripts. *Electronic Proceedings in Theoretical Computer Science*, 398:4–10, January 2024. doi:10.4204/EPTCS.398.4.

19 Yecine Megdiche, Fabian Huch, and Lukas Stevens. A linter for Isabelle: Implementation and evaluation. *CoRR*, abs/2207.10424, 2022. doi:10.48550/arXiv.2207.10424.

20 Adam Naumowicz and Artur Korniłowicz. A brief overview of mizar. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 67–72, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

21 Tobias Nipkow, Lawrence C. Paulson, and Makarius Wenzel. *A Proof Assistant for Higher Order Logic*. Isabelle/HOL Tutorial. URL: https://isabelle.in.tum.de/doc/tutorial.pdf.

22 Lawrence C. Paulson. Isabelle: The next seven hundred theorem provers. In Ewing Lusk and Ross Overbeek, editors, *9th International Conference on Automated Deduction*, pages 772–773, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg. doi:10.1007/BFB0012891.

23 Jessica Shi, Cassia Torczon, Harrison Goldstein, Andrew Head, and Benjamin C. Pierce. Designing Proof Deautomation for Rocq. In *15th PLATEAU Workshop on Programming Languages and Human-Computer Interaction*, July 2025. doi:10.1184/R1/29087003.v1.

24 Chengsong Tan, Alastair F. Donaldson, Jonathan Julián Huerta y Munive, and John Wickerson. The burden of proof: Automated tooling for rapid iteration on large mechanised proofs. In *13th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE@ICSE 2025, Ottawa, ON, Canada, April 27-28, 2025*, pages 34–45. IEEE, 2025. doi:10.1109/FORMALISE66629.2025.00010.

25 Makarius Wenzel. *The Isabelle/Isar Implementation*. Isabelle/Isar Implementation Manual. URL: https://isabelle.in.tum.de/dist/Isabelle2025-2/doc/implementation.pdf.

26 Makarius Wenzel. *The Isabelle/Isar Reference Manual*. Isabelle/Isar Reference Manual. URL: https://isabelle.in.tum.de/doc/isar-ref.pdf.

27 Makarius Wenzel. Interaction with formal mathematical documents in isabelle/pide. In Cezary Kaliszyk, Edwin C. Brady, Andrea Kohlhase, and Claudio Sacerdoti Coen, editors, *Intelligent Computer Mathematics - 12th International Conference, CICM 2019, Prague, Czech Republic, July 8-12, 2019, Proceedings*, Lecture Notes in Computer Science, pages 1–15. Springer, 2019. doi:10.1007/978-3-030-23250-4_1.

28 Markus Wenzel. Isar – A generic interpretative approach to readable formal proof documents. In Yves Bertot, Gilles Dowek, Laurent Théry, André Hirschowitz, and Christine Paulin, editors, *Theorem Proving in Higher Order Logics*, pages 167–183, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. doi:10.1007/3-540-48256-3_12.

29 Freek Wiedijk. A synthesis of the procedural and declarative styles of interactive theorem proving. *Log. Methods Comput. Sci.*, 8(1), 2012. doi:10.2168/LMCS-8(1:30)2012.