Lassie: HOL4 Tactics by Example

Heiko Becker
MPI-SWS,
Saarland Informatics Campus (SIC)
Germany
hbecker@mpi-sws.org

Nathaniel Bos*
McGill University
Canada
nathaniel.bos@mail.mcgill.ca

Ivan Gavran
MPI-SWS
Germany
gavran@mpi-sws.org

Eva Darulova
MPI-SWS
Germany
eva@mpi-sws.org

Rupak Majumdar
MPI-SWS
Germany
rupak@mpi-sws.org

Abstract

Proof engineering efforts using interactive theorem proving have yielded several impressive projects in software systems and mathematics. A key obstacle to such efforts is the requirement that the domain expert is also an expert in the low-level details in constructing the proof in a theorem prover. In particular, the user needs to select a sequence of tactics that lead to a successful proof, a task that in general requires knowledge of the exact names and use of a large set of tactics.

We present Lassie, a tactic framework for the HOL4 theorem prover that allows individual users to define their own tactic language *by example* and give frequently used tactics or tactic combinations easier-to-remember names. The core of Lassie is an extensible semantic parser, which allows the user to interactively extend the tactic language through a process of definitional generalization. Defining tactics in Lassie thus does not require any knowledge in implementing custom tactics, while proofs written in Lassie retain the correctness guarantees provided by the HOL4 system. We show through case studies how Lassie can be used in small and larger proofs by novice and more experienced interactive theorem prover users, and how we envision it to ease the learning curve in a HOL4 tutorial.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; **Programming by example**; **Macro languages**.

Keywords: Interactive Theorem Proving, HOL4, Semantic Parsing, Tactic Programming

ACM Reference Format:
Heiko Becker, Nathaniel Bos, Ivan Gavran, Eva Darulova, and Rupak Majumdar. 2021. Lassie: HOL4 Tactics by Example. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '21), January 18–19, 2021, Virtual, Denmark*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3437992.3439925

1 Introduction

Interactive theorem proving is increasingly replacing "pen-and-paper" correctness proofs in domains such as compilers [22, 24], operating system kernels [23], and formalized mathematics [14, 16]. Interactive theorem provers (ITPs) provide strong guarantees: all proof steps are formalized and machine-checked by a kernel using only a small set of generally accepted proof rules.

These guarantees come at a cost. Writing proofs in an ITP requires both domain expertise in the target research area as well as in the particulars of the interactive theorem prover. Formally proving a theorem requires an expert to manually translate the general high-level proof idea from a pen-and-paper proof into detailed, low-level kernel proof steps, which makes writing formal proofs tedious and time-consuming. Theorem provers thus provide tactic languages that allow to programmatically combine low-level proof steps [10, 15, 26, 37]. While this makes proofs less tedious, users need to build up a vocabulary of appropriate tactics, which constitutes a steep learning curve for novice ITP users.

Controlled natural language interfaces [1, 11] have been explored as an alternative, more intuitive interface to an ITP. However, these systems do not allow a combination with a general tactic language and are thus constrained to a specific subset of proofs.

In this paper, we present the tactic framework *Lassie* that allows HOL4 users to define their own tactic language on top of the existing ones *by example*, effectively providing an individualized interface. Each example consists of the to-be-defined tactic (a natural language expression, called *utterance*) and its definition using existing HOL4 tactics with concrete arguments.

[*] Nathaniel Bos was supported by a DAAD RISE Internship.

CPP '21, January 18–19, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8299-1/21/01.
https://doi.org/10.1145/3437992.3439925

For instance, we can define

```
instantiate 'x' with '⊤'
```

as

```
qpat_x_assum 'x' (qspec_then '⊤' assume_tac)
```

Newly defined Lassie tactics map directly and transparently to the underlying HOL4 tactics, and can be freely combined.

The main novelty to existing tactic languages is that Lassie allows to define tactics by example and thus does not require knowledge in tactic programming. A tactic defined by example is automatically *generalized* into a parametric tactic by Lassie to make the tactic applicable in different contexts, making Lassie go beyond a simple macro system.

Our key technical contribution is that Lassie realizes this definition-by-example using an extensible semantic parser [4, 35]. Lassie tactics are defined as grammar rules that map to HOL4 tactics. Lassie starts with an initial core grammar that is gradually extended through user-provided examples. For each example, the semantic parser finds matchings between the utterance and its definition. These matchings are used to create new rules for the grammar. Effectively, the semantic parser identifies the parameters of the newly given command, and thus generalizes from the given example. In our illustrative example, Lassie will identify `'x'` and ⊤ as arguments and add a rule that will work with arbitrary terms in place of `'x'` and ⊤.

Typically, extending a grammar through examples leads to ambiguity—for a single uterance-definition pair, there may be different possible matchings and thus several new parsing rules introduced. In previous work [35], this ambiguity was resolved through user interaction, e.g. showing the user a visualization of different parses and letting them choose the parse with the intended effect. However, it is nontrivial to visualize intermediate steps in a general-purpose programming language. Our core insight is that ITPs offer an ideal setting to resolve this ambiguity. We show that by carefully designing the core grammar and by making use of type information, the ambiguity can be resolved automatically. Furthermore, ITPs "visualize" individual steps by showing the intermediate proof state, and rule out wrong tactic definitions by forcing proofs to be checked by the ITP systems kernel.

Lassie's target audience are trained ITP users who implement decision procedures and simple tactic descriptions in Lassie. Lassie allows them to define their own individualized language by defining easy-to-remember names for individual tactics, or (frequently used) combinations of tactics. A tactic language implemented in Lassie can then used by non-expert users with prior programming experience but without necessarily in-depth experience with an ITP.

Compared to general tactic languages like ssreflect [15], Ltac [10], and Eisbach [26], Lassie requires less expert knowledge, at the expense of expressiveness. Similar to Lassie, structured tactic languages like Isar [36] have an extended parser. Extending a language like Isar requires editing the source code, while Lassie supports different tactic languages that can be defined simply by example. While Lassie can be used to define a tactic language that is closer to a natural language, by not requiring the interface to be entirely natural, Lassie is more general and flexible than systems like Mizar [1] and Naproche-SAD [11].

We implement Lassie as a library for the HOL4 [32] ITP system, but our technique is applicable to other theorem provers as well. Lassie is fully compatible with standard HOL4 proofs. Since all Lassie tactics map to standard HOL4 tactics, Lassie allows exporting a Lassie proof into standard HOL4 to maintain portability of proofs. On the other hand, the learned grammar can be ported as well and can be used, for example, by a teacher to predefine a domain-specific (tactic) language with Lassie, which is used by learners to ease proofs in a particular area.

We demonstrate Lassie on a number of case studies proving theorems involving logic, and natural and real numbers. In particular, we show the generality of the naturalized tactics by reusing them across different proofs, and we show that Lassie can be incrementally used for proofs inside larger code bases. Finally, by predefining a tactic language with Lassie, we develop a tutorial for the HOL4 theorem prover.

Contributions. In summary, this paper presents:

- an interactive, extensible framework called Lassie for defining tactics in an ITP by example;
- an implementation of this approach inside HOL4 (available at https://github.com/HeikoBecker/Lassie);
- a number of case studies and a HOL4 tutorial (available at https://github.com/HeikoBecker/HOL4-Tutorial) showing the effectiveness of Lassie.

2 Lassie by Example

We start by demonstrating Lassie on a small example, before explaining our approach in detail in Section 3.

For our initial example we choose to prove that the inverse function (x^{-1}) on real numbers is inverse monotonic for ≤. Figure 2 shows the formal statement of this theorem, together with an (informal) proof that one may find in a textbook (the proof uses a previously proven theorem about <).

Proofs in HOL4. Figure 1a shows the corresponding HOL4 theorem statement and proof. We can be sure that this proof is correct, because it is machine-checked by HOL4. HOL4 [32] is an ITP system from the HOL-family. It is based on higher-order logic and all proofs are justified by inference rules from a small, trusted kernel. Its implementation language is Standard ML (SML), and similar to other HOL provers like HOL-Light [18], and Isabelle/HOL [27], proof steps are described using so-called tactics that manipulate a goal state until the goal has been derived from true.

```
Theorem REAL_INV_LE_AMONO:
  ∀ x y.
    0 < x ∧ 0 < y ⇒
    x$^{-1}$ ≤ y$^{-1}$ ⇔ y ≤ x
Proof
  rpt strip_tac
  \\ `x$^{-1}$ < y$^{-1}$ ⇔ y < x`
    by (MATCH_MP_TAC REAL_INV_LT_ANTIMONO \\ fs [])
  \\ EQ_TAC
  \\ fs [REAL_LE_LT]
  \\ STRIP_TAC
  \\ fs [REAL_INV_INJ]
QED
```

(a) HOL4 proof

```
Theorem REAL_INV_LE_AMONO:
  ∀ x y.
    0 < x ∧ 0 < y ⇒
    x$^{-1}$ ≤ y$^{-1}$ ⇔ y ≤ x
Proof
  nltac `
    introduce assumptions.
    show 'inv x < inv y <=> y < x'
      using (use REAL_INV_LT_ANTIMONO
             THEN follows trivially).
    case split.
    simplify with [REAL_LE_LT].
    introduce assumptions.
    simplify with [REAL_INV_INJ]. trivial.`
QED
```

(b) Lassie proof

Figure 1. HOL4 proof (left) and Lassie proof (right) for theorem REAL_INV_LE_AMONO

Theorem 1. $\forall x\, y, 0 < x \wedge 0 < y \Rightarrow x^{-1} \leq y^{-1} \Leftrightarrow y \leq x$

Proof 1. *We show both sides of the implication separately. To show ($x^{-1} \leq y^{-1} \Rightarrow y \leq x$), we do a case split on whether $x^{-1} < y^{-1}$ or $x^{-1} = y^{-1}$. If $x^{-1} < y^{-1}$, the claim follows because the inverse function is inverse monotonic for $<$. If $x^{-1} = y^{-1}$, the claim follows from injectivity of the inverse. To show the case ($y \leq x \Rightarrow x^{-1} \leq y^{-1}$), we do a case split on whether $y < x$ or $y = x$. If $y < x$ the claim follows because the inverse function is inverse monotonic for $<$. If $y = x$, the claim follows trivially.*

Figure 2. Textbook proof that the inverse function is inverse monotonic for ≤

```
Theorem REAL_INV_LE_AMONO:
  ∀ x y.
    0 < x ∧ 0 < y ⇒
    (inv x ≤ inv y ⇔ y ≤ x)
Proof
  rpt strip_tac

  (* Proof *)

QED
```

```
1 subgoal:
val it =

  0.  0 < x
  1.  0 < y
  ---------------------
  x$^{-1}$ ≤ y$^{-1}$ ⇔ y ≤ x

  : proof
>
```

Figure 3. HOL4 theorem (left) and interactive proof session (right)

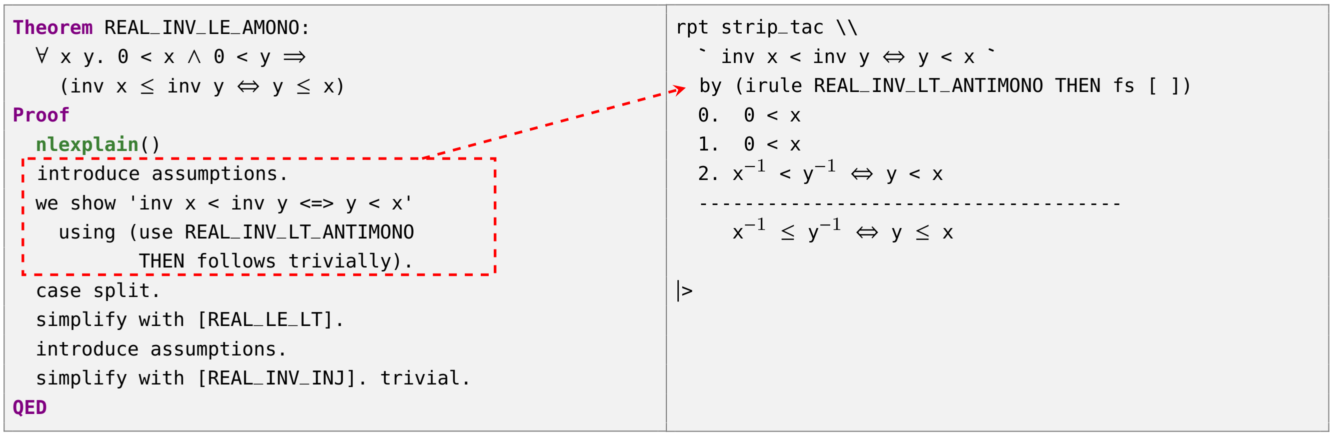
When doing a HOL4 proof, one first states the theorem to be proven and starts an interactive proof. Figure 3 shows the example proof statement from Figure 1a on the left and the interactive session on the right. To show that the theorem holds, the user would write a tactic proof at the place marked with `(* Proof *)`, starting with the initial tactic `rpt strip_tac`, sending each tactic to the interactive session on the right.

A HOL4 tactic implements e.g. a single kernel step, such as `assume_tac thm` which introduces `thm` as a new assumption, but a tactic can also implement more elaborate steps, like `fs`, which implements a stateful simplification algorithm, and `imp_res_tac thm`, resolving `thm` with the current assumptions to derive new facts. In our example, `rpt strip_tac` repeatedly introduces universally quantified variables and introduces left-hand sides of implications as assumptions.

After each tactic application, the HOL4 session prints the goal state that the user still needs to show, keeping track of the state of the proof. Once the HOL4 session prints `Initial goal proved`, the proof is finished. To make sure that the proof can be checked by HOL4 when run non-interactively, the separate tactics used in each step are chained together using the infix-operator `\\`. As this operator returns a tactic after taking some additional inputs, it is called a tactical.

Proofs in Lassie. Figure 1b shows the proof of our theorem using Lassie. This proof follows the same steps as the standard HOL4 proof, but each tactic is called using a name that we have previously defined in Lassie by example. We chose the Lassie tactics to be more descriptive (for us at least), and while they make the proof slightly more verbose, they also make it easier to follow for (non-)experts. Each of our Lassie tactics maps to corresponding formal HOL4 tactics, so that the proof is machine-checked by HOL4 as before, retaining all correctness guarantees.

Unlike existing tactic languages, Lassie allows to define custom tactics *by example* and thus does not require any



```
Theorem REAL_INV_LE_AMONO:
  ∀ x y. 0 < x ∧ 0 < y ⇒
    (inv x ≤ inv y ⇔ y ≤ x)
Proof
  nlexplain()
  introduce assumptions.
  we show 'inv x < inv y <=> y < x'
    using (use REAL_INV_LT_ANTIMONO
           THEN follows trivially).
  case split.
  simplify with [REAL_LE_LT].
  introduce assumptions.
  simplify with [REAL_INV_INJ]. trivial.
QED
```

```
rpt strip_tac \\
  ` inv x < inv y ⇔ y < x `
  by (irule REAL_INV_LT_ANTIMONO THEN fs [ ])
  0.   0 < x
  1.   0 < x
  2. x⁻¹ < y⁻¹ ⇔ y < x
  ------------------------------------
     x⁻¹ ≤ y⁻¹ ⇔ y ≤ x


|>
```

Figure 4. Intermediate proof state using goalTree's and `nlexplain`

knowledge in tactic programming. For instance, for our example proof, we defined a new tactic by

```
def `simplify with [REAL_LE_LT]` `fs [REAL_LE_LT]`;
```

Lassie automatically generalizes from this example so that we can later use this tactic with a different argument:

```
simplify with [REAL_INV_INJ]
```

To achieve this automated generalization, Lassie internally uses an extensible semantic parser [4]. That is, Lassie tactics are defined as grammar rules. Lassie initially comes with a relatively small core grammar, supporting commonly used HOL4 tactics. This grammar is gradually and interactively extended with additional tactic descriptions by giving example mappings. For instance our definition above would add the following rule to the grammar:

```
simplify with [THM1, THM2, ...] → fs [THM1, THM2, ...]
```

Note that this rule allows `simplify with` to be called with a list of theorems, not just a single theorem as in the example given. This generalization happens completely automatically in the semantic parser and does not require any programming by the user.

The Lassie-defined tactics can be used in a proof using the function `nltac`, that sends tactic descriptions to the semantic parser, which returns the corresponding HOL4 tactic. Because `nltac` has the same return type as all other standard HOL4 tactics, it can be used as a drop-in replacement for standard HOL4 tactics, and can be freely combined with other HOL4 tactics in a proof.

Explaining Proofs with Lassie. Lassie also comes with a function `nlexplain`. Instead of being a drop-in replacement, like `nltac`, `nlexplain` decorates the proof state with the HOL4 tactic that is internally used to perform the current proof step. Figure 4 shows an intermediate state when using `nlexplain` to prove our example theorem. All Lassie tactics inside the red dashed box on the left-hand side have been passed to `nlexplain`. The goal state on the right-hand side shows the current state of the proof as well as the HOL4 tactic script that has the same effect as the Lassie tactics.

We envision `nlexplain` to be used for example in a HOL4 tutorial to ease the learning curve when learning interactive theorem proving. Lassie allows a teacher to first define a custom tactic language that follows the same structure as the HOL4 proof, but that uses descriptive names and may be thus easier to follow for a novice. In a second step, one can use `nlexplain` to teach the actual underlying HOL4 tactics.

Function `nlexplain` can furthermore be used for sharing Lassie proofs without introducing additional dependencies on the semantic parser. While sharing Lassie proof scripts directly is possible, it requires sharing the state of the semantic parser as well. Alternatively, one can send the Lassie proof to `nlexplain` and obtain a HOL4 tactic script that can then be shared without depending on the semantic parser.

More Complex Tactics. While the target user that we had in mind when developing Lassie is not an ITP expert, experts may nonetheless find Lassie useful to, e.g., group commonly used combinations of tactics. For example, to make the proofs of simple subgoals easier, an expert can define a tactic that uses different simplification algorithms and an automated decision procedure to attempt to solve a goal automatically:

```
def `prove with [ADD_ASSOC]`
  `all_tac THEN ( fs [ ADD_ASSOC ] THEN NO_TAC)
    ORELSE (rw [ ADD_ASSOC ] THEN NO_TAC)
    ORELSE metis_tac [ ADD_ASSOC ]`
```

The HOL4 tactic will first attempt to solve the goal using the simplification algorithms implemented in tactics `fs` and `rw`, and if both fail, it will call into the automated decision procedure `metis_tac`, based on first-order resolution. (Tactical `t1 ORELSE t2` applies first tactic `t1`, and if `t1` fails, `t2` is

`$ROOT`	⟶	`$tactic`	$(\lambda x.x)$
`$tactic`	⟶	`$TOKEN`	(λx.`lookup "tactic"` x)
`$tactic`	⟶	`$thm->tactic $thm`	$(\lambda x\, y.x\, y)$
`$thm->tactic`	⟶	`$TOKEN`	(λx.`lookup "thm list->tactic"` x)
`$thm`	⟶	`$TOKEN`	$(\lambda x.x)$

`gen_tac`	:	`tactic`
`all_tac`	:	`tactic`
`strip_tac`	:	`tactic`
`fs`	:	`thm list->tactic`
`simp`	:	`thm list->tactic`

Figure 5. Excerpt from Lassie grammar (left) and the database (right), parsing tactics and thm list tactics

applied. `THEN NO_TAC` makes the simplification fail if it does not solve the goal.)

The resulting tactic description `prove with [THM1, THM2, ...]` is parametric in the used list of theorems making it applicable in different contexts.

Defined tactic descriptions are added to the grammar and are as such part of the generalization algorithm. Thus we can reuse the just defined tactic description to define an even more elaborate version:

```
def `'T' from [ CONJ_COMM ] `
  `'T' by ( prove with [CONJ_COMM] )`;
```

This tactic description, once generalized by the semantic parser, completely hides the fact that we may need to call into three different algorithms to prove a subgoal, while allowing us to enrich our assumptions with arbitrary goals, as long as they are provable by the underlying HOL4 tactics.

3 Defining Tactics in Lassie

Existing approaches to tactic languages, like Eisbach [26] and ssreflect [15] are implemented as domain-specific languages (DSL), usually within the theorem prover's implementation language. In these approaches, defining a new tactic is the same as defining a function in the implemented DSL. If a tactic should be generalized over e.g. a list of theorems, this generalization must be performed manually by the user of the tactic language.

In contrast, Lassie's tactics are defined in a grammar that is extended interactively by example using a semantic parser [4] that performs parameter generalization automatically. We define an initial core grammar (Section 3.1) that users can extend by example (Section 3.2). Each such defined description (Lassie tactic), maps a description to a (sequence of) HOL4 tactics, which is then applied to the proof state and checked by the HOL4 kernel. Note that a Lassie user does not directly modify the underlying (core) grammar—the extension happens by example.

3.1 The Core Grammar

The left-hand side of Figure 5 shows a subset of Lassie's core grammar. `$ROOT` is the symbol for the root node in the grammar and must always be a valid tactic. The core grammar is used to parse theorems, tactics, tacticals (of type `thm list -> tactic`) and to look up functions of these types.

Each rule has the form `$left` ⟶ `$right` ($\lambda x. \ldots$). While `$left` ⟶ `$right` works just as in a standard context free grammar, the λ-abstraction, called logical form, is applied to the result of parsing `$right` using the grammar. The logical form allows us to manipulate parsing results after they have been parsed by the grammar, essentially interpreting them within the parser. In Lassie we use it to implement function applications when combining tactics, and to lookup names in a database.

We have built a core grammar for Lassie that supports the most common tactics and tacticals of HOL4. For instance the core grammar will parse `fs [REAL_INV_INJ]` unambiguously into the equivalent SML code as its logical form. We think of this core grammar as the starting point for users to define Lassie tactics on top of the HOL4 tactics.

Adding every HOL4 tactic and tactical as a separate terminal to the grammar would clutter it unnecessarily and make it hard to maintain. That is why the grammar allows so-called lookup rules that check a dictionary for elements of predefined sets. The right-hand side of Figure 5 shows a subset of the database used for the lookups. In the grammar in Figure 5, a tactic can then either be looked up from the database (second rule), or a tactic can be a combination of a function of type `thm -> tactic` and a theorem (third rule). We refer to functions of type `thm -> tactic` as theorem tactics, as they take a theorem as input, and return a HOL4 tactic. Theorem tactics are again looked up from the database, whereas theorems can be any possible string denoted in the grammar by `$TOKEN`. In addition to HOL4 tactics and theorem tactics, our core grammar also uses a combination of rules (not shown in Figure 5) to support functions that return a tactic of type

- `thm list -> tactic`
- `tactic -> tactic`
- `term quotation -> tactic`
- `(thm -> tactic)-> tactic`
- `tactic -> tactic -> tactic`
- `term quotation -> (thm -> tactic) -> thm -> tactic`
- `term quotation list -> (thm -> tactic) -> thm -> tactic`

These types capture most of the tactics implemented in HOL4, and we add a subset of 53 commonly used tactics into the database.

Non-Ambiguity. A common issue in semantic parsing is grammar ambiguity. In Lassie, having an ambiguous grammar is not desirable as it would require users to disambiguate each ambiguous Lassie tactic while proving theorems. We thus aim to have an unambiguous grammar and achieve this by a careful design of our core grammar. By encoding the types of the tactics as non-terminals, our core grammar acts as a type-checker for our supported subset of HOL4 tactics. Even after defining custom tactics, the semantic parser will always parse Lassie tactics into the subset it can type check thus keeping the grammar unambiguous. During our experiments we have not found a case where extending the grammar introduced any ambiguity, which reassures this design choice.

3.2 Extending Lassie with New Definitions

With our core grammar, Lassie can parse the HOL4 tactics we have added to the grammar into their (equivalent) SML code. We now explain how this grammar can be interactively extended by example in order to provide custom names for (sequences of) tactics.

Lassie's tactic learning mechanism relies on a semantic parser. A semantic parser converts a natural language utterance into a corresponding (executable) logical form or—due to ambiguity—a ranked list of candidates. Semantic parsers can be implemented in many ways, e.g., they can be rule-based or learned from data [25]. SEMPRE [4], which we use, is a toolkit for developing semantic parsers for different tasks. It provides commonly used natural language processing methods, and different ways of encoding logical forms.

Lassie's semantic parser is implemented on top of the interactive version of SEMPRE [35]. It starts with a core formal grammar, which can be expanded through interactions with the user. Users can add new concepts to the grammar by example using Lassie's library function `def`, which invokes the semantic parser. Each example consists of a (*utterance*, *definition*) pair, where the utterance is the new tactic to be defined and the definition is an expression that is already part of the grammar. For instance, we can give as example:

```
def `simplify with REAL_ADD_ASSOC`   (*utterance*)
  `fs [REAL_ADD_ASSOC]`             (*definition*)
```

Note that the command demonstrates the new tactic (`simplify with`) with a particular argument (`REAL_ADD_ASSOC`), but does not explicitly state what the argument is.

The definition has to already be part of the grammar and thus fully parsable, otherwise the parser will reject the pair, whereas only some parts of the utterance may be parsable. That is, the definition needs to be already understood by the semantic parser, either because it is part of the core grammar or because it was previously already defined by the user.

The function `def` first obtains a logical form for the definition (which exists since the definition is part of the grammar). The semantic parser then induces one or more grammar rules from the utterance-definition pair and attaches the logical form of the definition to those rules.

The induction of new grammar rules relies on finding correspondences between parsable parts of the utterance and its definition. As an example, observe our `simplify with` command. Because `REAL_ADD_ASSOC` can be parsed into a category `$thm`, the two new production rules added to the grammar are:

```
$tactic →
  simplify with REAL_ADD_ASSOC (λ x.fs [REAL_ADD_ASSOC])
$tactic →
  simplify with $thm (λ thm. fs [thm])
```

Based on the second added rule, we can now use the Lassie tactic `simplify with` connected to any other description that is parsed as a `$thm`, because the parser identified `REAL_ADD_ASSOC` as an argument and generalized from our example by learning the λ-abstraction over the variable `thm`.

Next time the user calls, for instance,

```
nltac `simplify with REAL_ADD_COMM`
```

Lassie's semantic parser will parse this command into the tactic `fs [REAL_ADD_COMM]` using the second added rule.

4 Lassie Design

Lassie is implemented as a HOL4 library, which can be loaded into a running HOL4 session with `open LassieLib;`. This will start a SEMPRE process and the library captures its input and output as SML streams. Whenever `nltac` or `nlexplain` are run, the input is send to SEMPRE over the input stream, and if it can be parsed with the currently learned grammar, SEMPRE writes the resulting HOL4 tactic to the output stream as a string. If parsing fails, i.e. SEMPRE does not recognize the description, LassieLib raises an exception, such that an end-user can define the tactic with a call to `def`.

We want `nltac` to act as a drop-in replacement for HOL4 tactics. Therefore, `nltac` must not only be able to parse single tactics, but must also be able to parse full tactic scripts, performing a proof from start to finish. During our case-studies, we noticed that SEMPRE was not built for parsing large strings of text, but rather for smaller examples. To speed up parsing, we have defined a global constant, `LassieSep` which is used to split input strings of `nltac`. For example, calling

```
nltac `case split. simplify with [REAL_LE_LT].`
```

will lead to two separate calls to the semantic parser: one for `case split` and one for `simplify with [REAL_LE_LT]`. The resulting HOL4 tactics are joined together using the `THEN_LT` tactical, which is a more general version of the tactical `\\`, as it has an additional argument for selecting the subgoal to which the given tactic is applied. When proving a goal interactively, some tactics, like induction, and case splitting, can lead to multiple subgoals being generated. We use the `THEN_LT` tactical to implement selecting subgoals in `nltac`.

There are some differences in how `nltac` and `nlexplain` are used. Function `nltac` can be used as a drop-in replacement for HOL4 tactics, and thus supports selection of subgoals.

In contrast, **nlexplain** is meant to be used interactively, and therefore parses Lassie tactics, but does not support selection of subgoals. Instead, subgoals are proven in order of appearance. The main purpose of **nlexplain** is to show how Lassie tactics are translated back into HOL4 tactics. To do so, it modifies HOL4's interactive read-eval-print loop (REPL), and thus can only be used interactively, but not to replace plain HOL4 tactics in proof scripts like **nltac**.

To differentiate between SML expressions and HOL4 expressions, HOL4 requires HOL4 expressions to be wrapped in quotes (`), but quotes are also a way of allowing multiline strings in HOL4 proofscripts. Therefore we choose quotes to denote the start and end of a Lassie proofscript, and use apostrophes (') to denote the start and the end of a HOL4 expression in a Lassie proof script.

Lassie currently does not support debugging tactic applications. While an end-user can easily define new tactics by example using the semantic parser, figuring out the tactics' exact behavior, and fixing bugs still requires the user to manually step through the corresponding HOL4 tactic in an interactive proof and manually inspecting steps. We see extending Lassie with debugging support as future work.

4.1 Extending Lassie with New Tactics

Our initial core grammar supports only a fixed set of the most commonly used HOL4 tactics. However, it is common in ITPs to develop custom tactics on a per-project basis, possibly including fully blown decision procedures [33]. To make sure that users can add their own HOL4 tactics as well as custom decision procedures to Lassie, the library provides the functions `addCustomTactic`, `addCustomThmTactic`, and `addCustomThmlistTactic`.

The difference between **def** and `addCustom[*]Tactic` is in where the elements are added to the semantic parser's grammar. Function **def** uses SEMPRE's generalization algorithm and adds rules to the grammar that may contain non-terminals (e.g. `follows from [ $thms ]`). Function `addCustomTactic` always adds a new terminal to the grammar.

We explain `addCustomTactic` by example. Suppose a user wants to reuse an existing linear decision procedure for real numbers (`REAL_ASM_ARITH_TAC`) to close simple proof goals. Running `addCustomTactic REAL_ASM_ARITH_TAC` adds the new production rule `$tactic` ⟶ `REAL_ASM_ARITH_TAC` to the SEMPRE grammar. Tactic `REAL_ASM_ARITH_TAC` can then be used in subsequent calls to **def** to provide Lassie-based descriptions, or immediately in **nltac** and **nlexplain**.

Now that SEMPRE accepts the decision procedure as a valid tactic, we extend our expert automation tactic from before to try to solve a goal with this decision procedure too:

```
def `prove with [ADD_ASSOC]`
  `all_tac THEN ( fs [ ADD_ASSOC ] THEN NO_TAC)
    ORELSE (rw [ ADD_ASSOC ] THEN NO_TAC)
    ORELSE REAL_ASM_ARITH_TAC
    ORELSE metis_tac [ ADD_ASSOC ]`
```

Functions `addCustomThmTactic`, and `addCustomThmlistTactic` work similarly, adding grammar rules for `$thm->tactic` and `$thm list->tactic`.

4.2 Defining and Loading Libraries

Users can define libraries with their own defined Lassie tactics using the function `registerLibrary` which takes as first input a string, giving the libraries a unique name, and as second input a function of type `:unit -> unit`, where the function should call **def** on the definitions to be added, following Section 3.2. The defined libraries can then be shared and loaded simply by calling the function `loadLibraries`.

We defined libraries for proofs using logic, natural numbers, and real numbers from our case studies and used these in our HOL4 tutorial (Section 5)

5 Case Studies

We evaluate Lassie on three case studies and show how it can be used for developing a HOL4 tutorial. In the paper, we show only the main theorems for the case studies, but the full developments can be found in the Lassie repository.

5.1 Case Study: Proving Euclid's Theorem

First, we prove Euclid's theorem from the HOL4 tutorial [32] that is distributed with the HOL4 theorem prover documentation. Euclid's theorem states that the prime numbers form an infinite sequence. Its HOL equivalent states that for any natural number *n*, there exists a natural number *p* which is greater than *n* and a prime number.

To prove the final theorem, shown in Figure 6, we have proven 19 theorems in total. To prove these theorems, we defined a total of 22 new tactics using `LassieLib.`**`def`**. Some tactics have been used only once, but for example the tactic `[...] solves the goal`, was reused 16 times.

Another example is the tactic `thus PRIME_FACTOR for 'FACT n + 1'` which introduces a specialized version of the theorem `PRIME_FACTOR`, proving the existence of a prime factor for every natural number. Note how the tactic description can freely mix text descriptions with the parameters for the underlying tactic. Similarly, the first step of the HOL4 proof reads `CCONTR_TAC`, which initiates a proof by contradiction. For an untrained user, figuring out and remembering this name can be cumbersome, even though the user might know the high-level proof step. Instead, in Lassie we have used the—for us—more intuitive name `suppose not`.

Finally, each sub-step of the HOL4 proof is closed using the tactic `metis_tac`. For an expert user, it is obvious that `metis_tac` can be used, because the expert knows that it performs first order resolution to prove the goal. In the Lassie proof, we hide `metis_tac []` in combination with the simplification tactics `fs []` and `rw[]` under the description `[] solves the goal`.

```
Theorem EUCLID:
  ∀ n . ∃ p . n < p ∧ prime p
Proof
  CCONTR_TAC \\ fs[]
  \\ `FACT n + 1 ≠ 1`
      by rw[FACT_LESS, neq_zero]
  \\ qspec_then `FACT n + 1` assume_tac PRIME_FACTOR
  \\ `∃ q. prime q ∧ q divides (FACT n + 1)` by fs[]
  \\ `q ≤ n` by metis_tac[NOT_LESS_EQUAL]
  \\ `0 < q` by metis_tac[PRIME_POS]
  \\ `q divides FACT n`
      by metis_tac [DIVIDES_FACT]
  \\ `q = 1` by metis_tac[DIVIDES_ADDL, DIVIDES_ONE]
  \\ `prime 1` by fs[]
  \\ fs[NOT_PRIME_1]
QED
```

```
Theorem EUCLID: (* Lassie *)
  ∀ n . ∃ p . n < p ∧ prime p
Proof
  nltac`
    suppose not. simplify.
    we can derive 'FACT n + 1 <> 1'
      from [FACT_LESS, neq_zero].
    thus PRIME_FACTOR for 'FACT n + 1'.
    we further know
      '∃ q. prime q and q divides (FACT n + 1)'.
    show 'q <= n' using [NOT_LESS_EQUAL].
    show '0 < q' using [PRIME_POS] .
    show 'q divides FACT n' using [DIVIDES_FACT].
    show 'q=1' using [DIVIDES_ADDL, DIVIDES_ONE].
    show 'prime 1' using (simplify).
    [NOT_PRIME_1] solves the goal.`
QED
```

Figure 6. HOL4 proof (left) and Lassie proof (right) of euclids theorem

To further automate proving simple subgoals, we combine the tactic `[] solves the goal` with our Lassie tactic for proving subgoals (`show 'T' using (gen_tac)`) by defining `show 'T' using [...]` as
`show 'T' using ([...] solves the goal)`.

5.2 Case Study: Real and Natural Number Theorems

Next, we will show how Lassie can be used in more involved proofs about both real and natural numbers. As an example, we prove that for any natural number n, the sum of the cubes of the first n natural numbers is the same as the square of the sum. The Lassie proof of the final theorem is in Figure 7.

We have proven a total of 5 theorems: two (real-numbered) binomial laws, the closed form for summing the first n natural numbers, a side lemma on exponentiation, and the main result about cubing the first n numbers. All our proofs in this case study have been performed using the HOL4 theory of real numbers simply for convenience, as we found real number arithmetic easier for proving theorems that involve subtractions, powers, and divisions. We defined a total of 42 tactics by example using `LassieLib.def` and added 3 custom tactics using `LassieLib.addCustomTactic` and `LassieLib.addCustomThmTactic`. Again, some of the tactics were used only once or twice but our Lassie tactics for rewriting with a theorem (two calls to `LassieLib.def` to support rewriting from left to right, and right to left) are reused 13 times within the proofs.

This Lassie proof shows how it can be extended with custom tactics. Our restricted core grammar of Lassie does not include HOL4's decision procedure for reals. Nevertheless, a user may want to provide this tactic as part of some automation. Because Lassie supports on-the-fly grammar extensions we add the decision procedure for reals (`REAL_ASM_ARITH_TAC`) to the grammar: `addCustomTactic REAL_ASM_ARITH_TAC`. Having added this tactic, it can be used just like the HOL4 tactics we support in the base grammar. Thus we define a Lassie tactic using the decision procedure:

```
def `we know 'T'`
  `'T' by (REAL_ASM_ARITH_TAC ORELSE DECIDE_TAC)`
```

The semantic parser now automatically generalizes the grammar rule for this tactic, learning the rule

```
$tactic →
  we know '$term'(λ t.
    't' by (REAL_ASM_ARITH_TAC ORELSE DECIDE_TAC))
```

With this, we can use more complicated tactics like `we know '2 * &n * (1 + &n)* inv 2 = 2 * inv 2 * &n * (1 = &n)'`.

In general, combining the extensibility of Lassie and the generalization of SEMPRE allows us to support arbitrary settings where trained experts can implement domain-specific decision procedures and provide simple tactic descriptions to novice users that want to use them in a HOL4 proof, essentially decoupling the automation from its implementation. Equally, any user can define personalized and more intuitive names for often-used tactics.

5.3 Case Study: Naturalizing a Library Proof

In our final example, we show how Lassie can be integrated into larger developments, by proving a soundness theorem from a library of FloVer [3]. FloVer is a verified checker for finite-precision roundoff error bounds implemented in HOL4. Its HOL4 definitions and proofs span approximately 10000 lines of code and the interval library is one of the critical components which is used in most of the soundness proofs. As the FloVer proofs are performed over real numbers, we

```
Theorem sum_of_cubes_is_squared_sum:
   ∀ n. sum_of_cubes n = (sum n) pow 2
Proof
  nltac `
    induction on 'n'.
    simplify conclusion with [sum_of_cubes_def, sum_def].
    rewrite with [POW_2, REAL_LDISTRIB, REAL_RDISTRIB,
      REAL_ADD_ASSOC].
    showing
      '&SUC n pow 3 =
       &SUC n * &SUC n + &SUC n * sum n + sum n * &SUC n'
      closes the proof
      because (simplify conclusion with [REAL_EQ_LADD]).
    we know '& SUC n * sum n + sum n * &SUC n =
      2 * (sum n * & SUC n)'.
    rewrite once [<- REAL_ADD_ASSOC].
    rewrite last assumption.
    rewrite with [pow_3, closed_form_sum, real_div,
      REAL_MUL_ASSOC].
    we know '2 * &n * (1 + &n) * inv 2 =
      2 * inv 2 * & n * (1 + &n)'.
    rewrite last assumption.
    simplify conclusion with [REAL_MUL_RINV].
    we show 'n + 1 = SUC n' using (simplify conclusion).
    rewrite last assumption. simplify conclusion.
    we show '2 = (SUC (SUC 0))'
      using (simplify conclusion).
    rewrite last assumption. rewrite last assumption.
    rewrite with [EXP].
    we show 'SUC n = n + 1' using (simplify conclusion).
    rewrite last assumption.
    rewrite with [GSYM REAL_OF_NUM_ADD, pow_3].
    rewrite with [REAL_OF_NUM_ADD, REAL_OF_NUM_MUL,
                  MULT_RIGHT_1, RIGHT_ADD_DISTRIB,
                  LEFT_ADD_DISTRIB, MULT_LEFT_1].
    simplify.`
QED
```

Figure 7. Lassie proof that the sum of the natural numbers from 1 to *n* cubed is the same as the square of their sum

reuse the tactic descriptions from our previous example and do not need to add additional definitions. In Figure 8 we show that if we have an interval *iv*, and a real number $a \in iv$, then the inverse of *a* is contained in the inverse of *iv*.

This example shows that Lassie's tactic definitions are expressive enough to build libraries of common tactic descriptions that can be shared between projects.

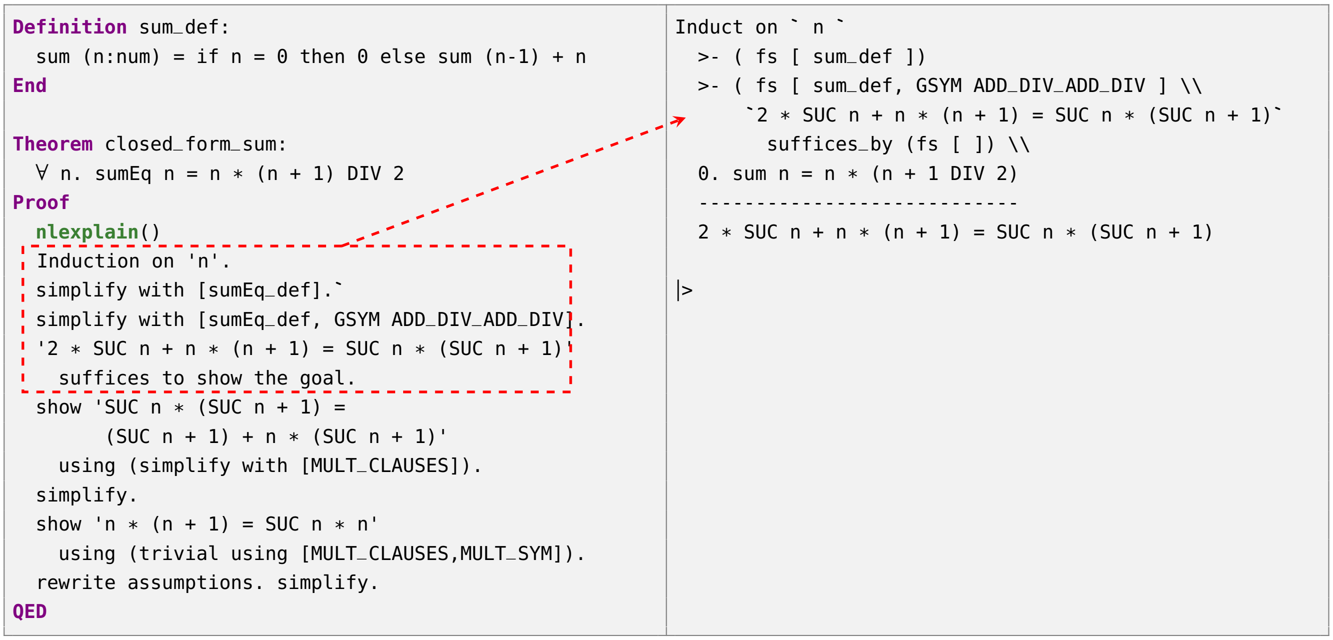
5.4 HOL4 Tutorial

We have used Lassie to write a new tutorial for HOL4 with the goal of decoupling the learning of the basic structure of formal proofs from the particular syntax and tactic names of HOL4, and by this easing the learning curve. Our tutorial is based on the existing HOL4 tutorial [32] and the HOL4 emacs interaction guide.

```
Theorem interval_inversion_valid:
  ∀ iv a.
    (SND iv < 0 \/ 0 < FST iv) /\ contained a iv ==>
    contained (inv a) (invertInterval iv)
Proof
  nltac `
  introduce variables.
  case split for 'iv'.
  simplify with [contained_def, invertInterval_def].
  introduce assumptions.
  rewrite once [<- REAL_INV_1OVER].
  Next Goal.
    rewrite once [ <- REAL_LE_NEG].
    we know 'a < 0'. thus 'a <> 0'.
    we know 'r < 0'. thus 'r <> 0'.
    'inv(-a) <= inv (-r) <=> (- r) <= -a' using
      (use REAL_INV_LE_AMONO THEN simplify).
    resolve with REAL_NEG_INV.
    rewrite assumptions.
    follows trivially.
  Next Goal.
    rewrite once [<- REAL_LE_NEG].
    we know 'a < 0'. thus 'a <> 0'. we know 'q <> 0'.
    resolve with REAL_NEG_INV.
    'inv (-q) <= inv (-a) <=> (-a) <= (-q)' using
      (use REAL_INV_LE_AMONO THEN simplify
       THEN trivial).
    rewrite assumptions. follows trivially.
  Next Goal.
    rewrite with [<- REAL_INV_1OVER].
    'inv r <= inv a <=> a <= r' using
      (use REAL_INV_LE_AMONO THEN trivial).
    follows trivially.
  Next Goal.
    rewrite with [<- REAL_INV_1OVER].
    'inv a <= inv q <=> q <= a' using
      (use REAL_INV_LE_AMONO THEN trivial).
    follows trivially.`
QED
```

Figure 8. Soundness of FloVer's interval inversion in Lassie

First, the new HOL4 user uses **nltac** and the Lassie tactics that we defined for our three case studies (i.e. loads them as libraries) to do the proofs. He or she can thus learn the syntax of theorems and definitions, as well as structure of proofs without having to also learn the often unintuitive tactic names of the proofs. For example, we show the proof of the closed form for summing the first *n* natural numbers from our tutorial in Figure 10. The example proof shows Lassie



```
Definition sum_def:
  sum (n:num) = if n = 0 then 0 else sum (n-1) + n
End

Theorem closed_form_sum:
  ∀ n. sumEq n = n * (n + 1) DIV 2
Proof
  nlexplain()
  Induction on 'n'.
  simplify with [sumEq_def].`
  simplify with [sumEq_def, GSYM ADD_DIV_ADD_DIV].
  '2 * SUC n + n * (n + 1) = SUC n * (SUC n + 1)'
    suffices to show the goal.
  show 'SUC n * (SUC n + 1) =
        (SUC n + 1) + n * (SUC n + 1)'
    using (simplify with [MULT_CLAUSES]).
  simplify.
  show 'n * (n + 1) = SUC n * n'
    using (trivial using [MULT_CLAUSES,MULT_SYM]).
  rewrite assumptions. simplify.
QED
```

```
Induct on ` n `
  >- ( fs [ sum_def ])
  >- ( fs [ sum_def, GSYM ADD_DIV_ADD_DIV ] \\
      `2 * SUC n + n * (n + 1) = SUC n * (SUC n + 1)`
        suffices_by (fs [ ]) \\
  0. sum n = n * (n + 1 DIV 2)
  ------------------------------
  2 * SUC n + n * (n + 1) = SUC n * (SUC n + 1)

|>
```

Figure 9. Intermediate state of nlexplain in our tutorial

tactics that abstract from the tactic, but not the theorem names. Lassie has limited support for defining descriptions of theorems similar to how Lassie tactics are defined which could be used when developing individual languages.

In the second step, the new HOL4 user is introduced to the HOL4 tactics using nlexplain. For instance, they can step through the proof and see the HOL4 tactics underlying each Lassie tactic. We show an example in Figure 9. The left-hand side shows the HOL4 proof state obtained by applying Lassie tactics with nlexplain, and the right-hand side the modified HOL4 REPL with the current proof goal and a partial HOL4 tactic script. The red dashed box on the left-hand side marks all Lassie tactics that have been passed to nlexplain.

Our tutorial is split into six separate parts. We start by explaining how HOL4 (and Lassie) are installed and configured on a computer such that the tutorial can be followed interactively. Next, we explain how one interacts with HOL4 in an interactive session. The first technical section uses the proof from Figure 10 as a first example of an interactive HOL4 proof, using only nltac to perform proofs. Having introduced the reader to the basics of interactive proofs in HOL4, we show how a simple library of proofs can be developed. The library is a re-implementation of our first case study, and hence follows the structure of the original HOL4 tutorial. It spans a total of two definitions, and 13 theorems. For each of the theorems we show a proof using nltac. Only after these introductory sections, where a user will have already gained an intuition both about how one interacts with the HOL4 REPL, and how proofs are stored in reusable theories, the next section introduces nlexplain and explains how HOL4 proofs are performed with plain HOL4 tactics. Finally, the tutorial concludes with some helpful tips and tricks that we have collected.

We defined the tutorial using definitions that we personally found intuitive. However, Lassie’s ability to define tactics by example allows each teacher to define their own individual language in a straightforward way.

6 Related Work

In this section, we review approaches designed to ease the user burden when writing proofs in an ITP.

Hammers. So-called “hammers” use automated theorem provers (ATP) to discharge proof obligations by translating a proof goal into the logic of an ATP and a proof back into the logic of the interactive prover. Examples are Sledgehammer [28] for Isabelle, HolyHammer [21] for HOL4, and a hammer for Coq [8]. A general overview is given in the survey paper by Blanchette et al. [5]. Some of these use learning to predict which premises are needed to be sent to the ATP, in order not to overwhelm the prover. In contrast to Lassie, the main focus of such hammers is not to make the proofs more accessible but to solve simple proof obligations using a push-button method. As Lassie is open to adding custom decision procedures we think that integrating a hammer with Lassie could provide for even richer and easier to define tactic languages by automating simple proofs.

Learning-based. While hammers try to automate the proof with the help automated theorem provers, other systems use statistical methods to recommend tactics to the end

```
Theorem closed_form_sum:
  ∀ n. sum n = (n * (n + 1)) DIV 2
Proof
  nltac`
   Induction on 'n'.
   Goal 'sum 0 = 0 * (0 + 1) DIV 2'.
     simplify.
   End.
   Goal 'sum (SUC n) = SUC n * (SUC n + 1) DIV 2'.
     use [sum_def, GSYM ADD_DIV_ADD_DIV] to simplify.
     '2 * SUC n + n * (n + 1) = SUC n * (SUC n + 1)'
       suffices to show the goal.
     show 'SUC n * (SUC n + 1) =
          (SUC n + 1) + n * (SUC n + 1)'
       using (simplify with [MULT_CLAUSES]).
     simplify.
     show 'n * (n + 1) = SUC n * n'
       using (trivial using [MULT_CLAUSES, MULT_SYM]).
     '2 * SUC n = SUC n + SUC n' follows trivially.
     'n * (SUC n + 1) = SUC n * n + n' follows trivially.
     rewrite assumptions. simplify.
   End.`
QED
```

Figure 10. Example proof of the closed form for summing n numbers using Lassie in our HOL4 tutorial

user to finish a proof. DeepHOL [2] learns a neural network that, given a proof goal, predicts a potential next tactic in HOL Light. GamePad [19] and the work by Yang et al. [38] similarly use machine learning to predict tactics for Coq. TacticToe [13] uses A* search, guided by previous tactic-level proofs, to predict tactics in HOL4.

Programming Language-based. Languages like Eisbach [26], Ltac [10], Ltac2 [29] and Mtac2 [20] use rigorous programming language foundations to give more control to expert users when writing tactics. Eisbach and Ltac are tactic languages similar to the one of HOL4. Mtac2 formalizes "Coq in Coq" allowing to define tactics as Coq programs, whereas Ltac2 is a strongly typed language for writing Coq tactics. The tactic language of the Lean theorem prover [9] additionally implements equational reasoning on top of its tactics, which allows for more textbook-like proofs. Recently, the Lean theorem prover has also been extended with a hygienic macro system [34]. A core contribution of their work is excluding unintentional capturing in tactic programming, thus making tactic programming more robust. In Lassie we did not experience any hygiene issues as the definition by example relies on the semantic parser to do the generalization and as such keeps variable levels separate. Using any of the languages above requires all the desired generality to be stated explicit in the tactic definition, usually in the form of function definitions. In contrast, Lassie's definition by example makes it easier to define new tactics and generalizes automatically.

Natural Language Interfaces. Several systems provide an interface to a theorem prover that is as close as possible to natural language. Languages like Isar [36], Mizar [1], and the work by Corbineau [6] follow a similar approach as Lassie by having an extended parser. Their supported naturalized proof descriptions are fixed to the authors style of declarative proofs and extending or changing these would required editing the tool code. In contrast, Lassie is extensible enough to support different tactic languages that can coexist without interferring if not loaded simultaneously.

The Naproche system [11] provides a controlled natural language, which maps natural language utterances into first-order logic proof obligations, to be checked by an (automated) theorem prover (e.g. E Prover [31]). The extensions to Alfa by Hallgren et al. [17] also use natural language processing technology to extend the Alfa proof editor with a more natural language. The book by Ganesalingam [12] gives a comprehensive explanation of the relation between natural language and mathematics. Similarly, Ranta et al. [30] provide more sophisticated linguistic techniques to translate between natural language and predicate logic. An orthogonal approach to the above is presented in the work by Coscoy et al. [7]. Instead of translating from natural language to tactics, they provide a translation from Coq proof terms to natural language. The main goal of these systems is to provide an interface that supports as much natural language as possible. A major limitation, however, is that their grammars are fixed, i.e. only the naturalized tactics implemented by the authors is available. Our work does not strive to be a full natural language interface, and in turn provides an extensible grammar, which adapts to different users and proofs.

7 Conclusion

We have presented the Lassie tactic language framework for the HOL4 theorem prover. Using a semantic parser with an extensible grammar, Lassie learns individualized tactics from user-provided examples. Our example case studies show that these learned tactics can be easily reused across different proofs and can ease both the writing and reading of HOL4 proofs by providing a more intuitive, personalized interface to HOL4's tactics.

Acknowledgments

The authors would like to thank Magnus Myreen, Zachary Tatlock, and the anonymous reviewers of ITP 2020 and CPP 2021 for providing feedback on Lassie and (initial) drafts of the paper. Gavran and Majumdar were supported in part by the DFG project 389792660 TRR 248–CPEC and by the European Research Council under the Grant Agreement 610150 (ERC Synergy Grant ImPACT).

References

[1] Grzegorz Bancerek, Czeslaw Bylinski, Adam Grabowski, Artur Kornilowicz, Roman Matuszewski, Adam Naumowicz, Karol Pak, and Josef Urban. 2015. Mizar: State-of-the-art and Beyond. In *International Conference on Intelligent Computer Mathematics (CICM)*. https://doi.org/10.1007/978-3-319-20615-8_17

[2] Kshitij Bansal, Sarah Loos, Markus Rabe, Christian Szegedy, and Stewart Wilcox. 2019. HOList: An Environment for Machine Learning of Higher Order Logic Theorem Proving. In *International Conference on Machine Learning (ICML)*.

[3] Heiko Becker, Nikita Zyuzin, Raphaël Monat, Eva Darulova, Magnus O Myreen, and Anthony Fox. 2018. A Verified Certificate Checker for Finite-Precision Error Bounds in Coq and HOL4. In *FMCAD (Formal Methods in Computer Aided Design)*. https://doi.org/10.23919/FMCAD.2018.8603019

[4] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic Parsing on Freebase from Question-Answer Pairs. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[5] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. 2016. Hammering towards QED. *Journal of Formalized Reasoning* 9, 1 (2016). https://doi.org/10.6092/issn.1972-5787/4593

[6] Pierre Corbineau. 2007. A Declarative Language for the Coq Proof Assistant. In *International Workshop on Types for Proofs and Programs (TYPES)*. https://doi.org/10.1007/978-3-540-68103-8_5

[7] Yann Coscoy, Gilles Kahn, and Laurent Théry. 1995. Extracting Text from Proofs. In *International Conference on Typed Lambda Calculi and Applications (TLCA)*. https://doi.org/10.1007/BFb0014048

[8] Łukasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for dependent type theory. *Journal of Automated Reasoning* 61, 1-4 (2018). https://doi.org/10.1007/s10817-018-9458-4

[9] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *International Conference on Automated Deduction (CADE)*. https://doi.org/10.1007/978-3-319-21401-6_26

[10] David Delahaye. 2000. A Tactic Language for the System Coq. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*. https://doi.org/10.1007/3-540-44404-1_7

[11] Steffen Frerix and Peter Koepke. 2019. Making Set Theory Great Again: The Naproche-SAD Project. *Conference on Artificial Intelligence and Theorem Proving (AITP)* (2019).

[12] Mohan Ganesalingam. 2013. *The Language of Mathematics - A Linguistic and Philosophical Investigation.* Lecture Notes in Computer Science, Vol. 7805. Springer. https://doi.org/10.1007/978-3-642-37012-0

[13] Thibault Gauthier, Cezary Kaliszyk, Josef Urban, Ramana Kumar, and Michael Norrish. 2020. TacticToe: Learning to Prove with Tactics. *Journal of Automated Reasoning* (2020).

[14] Georges Gonthier. 2008. Formal proof–the four-color theorem. *Notices of the AMS* 55, 11 (2008).

[15] Georges Gonthier and Assia Mahboubi. 2010. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning* 3, 2 (2010). https://doi.org/10.6092/issn.1972-5787/1979

[16] Thomas C. Hales. 2006. Introduction to the Flyspeck Project. In *Mathematics, Algorithms, Proofs.*

[17] Thomas Hallgren and Aarne Ranta. 2000. An Extensible Proof Text Editor. In *International Conference on Logic for Programming and Automated Reasoning (LPAR)*. https://doi.org/10.1007/3-540-44404-1_6

[18] John Harrison. 2009. HOL light: An overview. In *International Conference on Theorem Proving in Higher Order Logics (TPHOL)*.

[19] Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. 2019. GamePad: A Learning Environment for Theorem Proving. In *International Conference on Learning Representations (ICLR)*.

[20] Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas, and Derek Dreyer. 2018. Mtac2: typed tactics for backward reasoning in Coq. *Proc. ACM Program. Lang.* 2, ICFP (2018), 78:1–78:31. https://doi.org/10.1145/3236773

[21] Cezary Kaliszyk and Josef Urban. 2014. Learning-Assisted Automated Reasoning with Flyspeck. *Journal of Automated Reasoning* 53, 2 (2014). https://doi.org/10.1007/s10817-014-9303-3

[22] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2019. The verified CakeML compiler backend. *Journal of Functional Programming* 29 (2019). https://doi.org/10.1017/S0956796818000229

[23] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles (SOSP)*. https://doi.org/10.1145/1629575.1629596

[24] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (2009). https://doi.org/10.1145/1538788.1538814

[25] Percy Liang. 2016. Learning executable semantic parsers for natural language understanding. *Commun. ACM* 59, 9 (2016). https://doi.org/10.1145/2866568

[26] Daniel Matichuk, Toby C. Murray, and Makarius Wenzel. 2016. Eisbach: A Proof Method Language for Isabelle. *Journal of Automated Reasoning* 56, 3 (2016). https://doi.org/10.1007/s10817-015-9360-2

[27] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic.* Lecture Notes in Computer Science, Vol. 2283. Springer. https://doi.org/10.1007/3-540-45949-9

[28] Lawrence C. Paulson and Kong Woei Susanto. 2007. Source-Level Proof Reconstruction for Interactive Theorem Proving. In *International Conference on Theorem Proving in Higher Order Logics (TPHOL)*. https://doi.org/10.1007/978-3-540-74591-4_18

[29] Pierre-Marie Pédrot. 2019. Ltac2: Tactical Warfare. *CoqPL 2019* (2019).

[30] Aarne Ranta. 2011. Translating between Language and Logic: What Is Easy and What Is Difficult. In *International Conference on Automated Deduction (CADE)*. https://doi.org/10.1007/978-3-642-22438-6_3

[31] Stephan Schulz. 2013. System Description: E 1.8. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*. https://doi.org/10.1007/978-3-642-45221-5_49

[32] Konrad Slind and Michael Norrish. 2008. A Brief Overview of HOL4. In *International Conference on Theorem Proving in Higher Order Logics (TPHOL)*. https://doi.org/10.1007/978-3-540-71067-7_6

[33] Alexey Solovyev and Thomas C. Hales. 2013. Formal Verification of Nonlinear Inequalities with Taylor Interval Approximations. In *NASA Formal Methods Symposium (NFM)*. https://doi.org/10.1007/978-3-642-38088-4_26

[34] Sebastian Ullrich and Leonardo de Moura. 2020. Beyond Notations: Hygienic Macro Expansion for Theorem Proving Languages. In *International Joint Conference on Automated Reasoning (IJCAR)*. https://doi.org/10.1007/978-3-030-51054-1_10

[35] Sida I. Wang, Samuel Ginn, Percy Liang, and Christopher D. Manning. 2017. Naturalizing a Programming Language via Interactive Learning. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL*. https://doi.org/10.18653/v1/P17-1086

[36] Markus Wenzel. 1999. Isar - A Generic Interpretative Approach to Readable Formal Proof Documents. In *International Conference on Theorem Proving in Higher Order Logics (TPHOL)*. https://doi.org/10.1007/3-540-48256-3_12

[37] Markus Wenzel and Lawrence C. Paulson. 2006. Isabelle/Isar. In *The Seventeen Provers of the World.* Lecture Notes in Computer Science, Vol. 3600. Springer, 41–49. https://doi.org/10.1007/11542384_8

[38] Kaiyu Yang and Jia Deng. 2019. Learning to Prove Theorems via Interacting with Proof Assistants. In *International Conference on Machine Learning (ICML)*.