

Design of point-and-click user interfaces for proof assistants

Bohua Zhan^{1(✉)}, Zhenyan Ji^{2(✉)}, Wenfan Zhou², Chaozhu Xiang², Jie Hou², Wenhui Sun²

¹ State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

bzhan@ios.ac.cn

² Beijing Jiaotong University, Beijing, China

{zhyji, zhouwenfan, czxiang, houjie, whsun1}@bjtu.edu.cn

Abstract. In interactive theorem proving, human users interact with proof assistants to construct and verify formal proofs. The most popular proof assistants today all have user interfaces that are largely text-based. This leads to a steep learning curve for new users of these tools. In this paper, we propose a framework for designing user interfaces for proof assistants based on pointing and clicking. While a main goal of the design is ease of learning for new users, we intend for the design to be suitable for real verification tasks. The design is also extensible, allowing custom proof methods and search functionality to be added in a convenient way. We implement our ideas in a web interface, with backend provided by holpy, a new system for interactive theorem proving implemented in Python. The resulting user interface is tested on theorems in logic, sets, functions, Peano arithmetic, and lists, demonstrating its applicability in a wide range of areas.

Keywords: Proof assistants · User interface · Tactics

1 Introduction

Interactive theorem proving aims to construct and verify formal proofs via interaction between the computer and the human user. In recent years, it has seen several major accomplishments, including formal verification of the seL4 microkernel [13], verification of a realistic C compiler [14], and formal proofs of the Feit-Thompson theorem [11] and Kepler’s conjecture [10]. These works show that interactive theorem proving can be applied to very complex mathematical theorems and computer systems. However, verification projects still take considerable human effort. The works on the seL4 project, the Feit-Thompson theorem, and Kepler’s conjecture each have an estimated cost of over 20 person years. In addition, the proof assistants used – HOL Light [12], Coq [4], and Isabelle [15], are generally considered to have a steep learning curve for new users, making it difficult and time consuming to form and train new teams. The combination of these factors can be seen as a major obstacle to more widespread application of interactive theorem proving. Hence, how to design proof assistants to make it more accessible for users is an important problem for this field.

The most popular proof assistants today have user interfaces that are largely text-based. The main form of interaction consists of the user editing a text file containing the

proof, either as a sequence of tactics or (as in Isabelle/Isar [17]) written in a structured proof language. During editing, the user interface displays the state of the proof at the current location of the proof text. To use the proof assistant, the user needs to be familiar with names of the major tactics, as well as some of the commonly used theorems. The Isabelle/Isar language makes the resulting proof text more readable. However, it requires the user to further understand the use of a number of keywords for structuring the proof.

Naturally, we may ask whether it is realistic to have user interfaces for proof assistants that is based on pointing and clicking. In an ideal setting, most of the interaction with the user interface should consist of choosing which facts to consider, and which actions to take through clicks of the mouse. The user interface performs the selected actions, and offers suggestions for future actions. Only occasionally will the user need to enter text using the keyboard, and even then only mathematical expressions rather than names of tactics or theorems.

While there have been attempts to build point-and-click user interfaces in the past, they have not gained widespread adoptance for general-purpose theorem proving. Potential problems with existing designs include limited search functionality – the user still need to find names of theorems to use, and limited extensibility – there is usually a fixed set of proof methods, with no easy way to grow them for new application domains. This limits the use of user interfaces to simple examples, or to the special domains for which they are designed.

In this paper, we propose a new framework for designing user interfaces for proof assistants that is based on pointing and clicking. In this design, the user interacts with the interface mainly in three ways. First, at each step of the proof, the user chooses which goal to consider and which facts in the proof to use. Second, the user chooses an action from the list of actions suggested by the computer. The suggestion process may involve (but is not limited to) matching the chosen facts and goal with existing theorems. Third, the user annotates each proved theorem, to tell the computer which directions for applying the theorem are the most common, and should be considered during the suggestion process in future proofs. We give a general definition of *proof methods*. Any function satisfying this general definition can be added to the user interface. This makes the design extensible: new proof methods reflecting domain-specific knowledge can be added in a convenient way.

We implement our design in a web interface³. The backend for the interface is provided by holpy, a new system for interactive theorem proving implemented in Python [18]. There are several aspects in holpy’s design that are different from systems such as Isabelle and Coq, including a format for explicit representation of proofs and theories based on JSON [8]. The format for theory files is not designed for direct editing by the user. This means any user interface must interpret the theory files for display in a more readable form, and reflect user changes back to the file. While this makes user interfaces more difficult to implement at first, it has the long-term advantage of allowing more flexibility in its design. The current work can be viewed as a first attempt to implement a user interface for holpy, justifying its choice of the theory format.

³ code available at <https://gitee.com/bhzhan/holpy>

We now give an outline for the rest of this paper. In Section 2, we give an overview of the holpy system, focusing on those aspects of design that are different from the major proof assistants, and which are relevant to the current work. In Section 3, we describe the key concept of *proof methods*, which forms the connection between the user interface and the underlying proof automation in holpy. In Section 4, we describe our implementation of these ideas, and present some statistics from tests of the user interface on theorems from various domains. In Section 5, we describe in detail a case study on the Knaster-Tarski fixed point theorem. Finally, we conclude in Section 6 with a discussion of future work.

Related work There have been a few early attempts to build point-and-click user interfaces for proof assistants. The work of Bertot et al. in [5], and extended in [6], introduced the idea of “Proof by Pointing”. In this framework, the user can trigger deduction rules in logic by pointing to specific parts of the goal formula. The latter work also studied how to implement script management (including undoing and redoing steps), and textual explanation of proofs. Another line of work by Abrial et al. [2] developed an user interface for Atelier B to perform formal proofs in set theory. The work by Breitter in [7] constructed a visual theorem proving interface based on connecting blocks, albeit also limited to proofs in logic.

In the area of program and system verification, several tools have user interfaces that allow proofs to be conducted by pointing-and-clicking. These include KeY [3] and KeYmaera/KeYmaera X [16, 9]. These tools allow users to choose subgoals and select which actions to take from a menu. There is some similarity in the mode of interaction between our work and these systems. However, our focus is on general-purpose theorem proving in higher-order logic, rather than for specific program logics.

2 Overview of holpy

In this section, we give an overview of the holpy system, focusing on aspects that are different from systems such as Isabelle and Coq, and which are relevant to the current work. More details on the design of holpy can be found in [18].

holpy is a new system for interactive theorem proving implemented in Python. Its logical foundation is higher-order logic (also known as simply-typed lambda calculus), similar to existing proof assistants such as Isabelle/HOL [15], HOL Light [12], and HOL4 [1]. While holpy does not propose any innovation in logical foundations, it makes significant changes to how proofs and theories are represented. In particular, it proposes a pervasive use of macros in proof representation. Macros can be considered as abbreviations, allowing multiple steps of proof to be represented by a single step. The use of macros means explicit proofs can be stored and checked by third-party tools without running into the usual scalability issues. For representing theories, holpy chooses a JSON-based format. This format is not designed for direct human editing, but is convenient to read and write by computer programs. Finally, holpy provides an API in Python for implementing proof automation (as well as other tools). A major goal of holpy’s design is to show that with export of explicit proofs, the type and memory safety issues of Python does not pose any problem for soundness of proof-checking.

In the remainder of this section, we discuss various aspects of holpy in more detail.

2.1 Logical foundation

The logical foundation for holpy is higher-order logic (simply-typed lambda calculus). Proof are conducted in natural deduction style. Intermediate results in proofs are represented by sequents with a number of antecedents and a single consequent. A sequent with antecedent A_1, \dots, A_n and consequent C is written in the usual notation as $A_1, \dots, A_n \vdash C$.

The logical foundation fixes a set of *primitive deduction rules*, with each rule taking a number of input sequents and possibly additional arguments, and outputs a sequent (or raises an exception). Inserting a previously proved theorem into a proof is also a valid deduction rule. A theorem is *provable* if it can be obtained by applying these rules. Our choice of primitive deduction rules largely follows that of existing proof assistants such as Isabelle/HOL, HOL Light, and HOL4. Examples include introduction and elimination rules for implication and forall quantification, congruence properties of equality, and substitution of type and term variables.

2.2 Proof rules and macros

Proof rules can be considered as a generalization of primitive deduction rules. They are intended to represent a number of more basic steps of proof. In general, a proof rule takes the current theory environment (list of existing constants, theorems, etc), a list of input sequents, and possibly additional arguments, and outputs a single sequent (or raises an exception). Each proof rule defines a precise signature for its additional arguments.

Primitive deduction rules is a special case of proof rules, that cannot be expanded into more basic steps. Another fundamental proof rule is `theorem`, which takes no input sequents and a theorem name as additional argument. If there exists a theorem with that name in the current theory environment, it outputs that theorem as a sequent. Otherwise, it raises an exception.

The other proof rules are called *macros*. They represent several more basic applications of proof rules as a single step. Each macro has an associated expansion function which, given the current theory environment, the input sequents, and the additional arguments, returns the output sequent together with the invocations of proof rules used to obtain it (or raises an exception). The expanded proof (whose format will be described later) can be used during proof checking, so that the implementation of the macro does not need to be trusted. The use of macros means any portion of proof that can be algorithmically generated can be stored as a single step, so that large proofs can be stored for checking by third-party tools, without encountering the usual scalability issues.

We give some examples of macros for illustration.

- Applying a theorem: simple macros include applying a single theorem, which takes the name of the theorem as the additional argument. It first looks up the theorem from the theory environment. Then, it matches the input sequents to the assumptions of the theorem, and outputs the conclusion. The expanded proof consists of inserting the theorem, substitution, and elimination of implication.

- Arithmetic operations on natural numbers: with natural numbers represented in binary form, arithmetic operations, such as $99 \times 101 = 9999$, can be represented as a single step. The expanded proof consists of rewriting using lemmas about binary numbers. Likewise, normalization of expressions using (semi)ring identities can be represented as a single step.
- Automated theorem provers: full automated theorem provers can be implemented as a single (very complex) macro. It takes the goal to be solved as the additional argument, and attempts to solve the goal. The expanded proof may be computed from a trace outputted by the theorem prover. If the trace is not available or cannot be interpreted properly, the macro needs to be *trusted* – used without checking the expanded proof.

2.3 Format for proofs

There are two forms of proof representation used in holpy: linear proofs and proof terms.

A linear proof consists of an ordered list of *proof items*. Each proof item consists of an identifier, the name of a proof rule, additional arguments for the proof rule, and a list of identifiers of earlier proof items, representing the input sequents. A linear proof can be checked (within a theory environment) by reading the proof items in order, computing the sequent for each proof item by invoking the corresponding proof rule. The result of a linear proof is the sequent corresponding to the last proof item.

How to represent identifiers is largely conventional. We choose to represent each identifier as a tuple of natural numbers, written in dot-separated form (e.g. 0.2.1). This allows us to express sub-proofs. For example, steps in the main trunk of the proof have identifiers 0, 1, 2, etc. The conclusion for the proof item with identifier 1 may take several steps outside the main trunk, which are given identifiers 1.0, 1.1, 1.2, etc. In practice, we use sub-proofs like these when introducing variables and assumptions, as will be seen in the examples in the next subsection.

Linear proofs are more convenient for visualization and storage, and so is the form of proof that the user directly encounters. However, it is clear that linear proofs are difficult to construct for automatic procedures. Automatic procedures prefer to work with proof terms: proofs stored as directed acyclic graphs in memory. Each vertex of the graph is a proof item, and input sequents of the proof item are referred to directly (so identifiers are not needed). We will not discuss proof terms in detail, since proof automation is not the focus of this paper. It suffices to know that there is a standard algorithm for converting proof terms to linear proofs. Hence, the general idea for proof automation in holpy is to first construct proof terms, then convert them to linear proofs for storage and viewing by the user.

2.4 Example proofs

For illustration, we give two simple examples of proofs. Consider the proposition $A \wedge B \longrightarrow B \wedge A$. A linear proof can be written as follows:

0. $A \wedge B \vdash A \wedge B$ **by** assume $A \wedge B$

1. $A \wedge B \vdash A$ **by** `apply_theorem conjD1` **from** 0
2. $A \wedge B \vdash B$ **by** `apply_theorem conjD2` **from** 0
3. $A \wedge B \vdash B \wedge A$ **by** `apply_theorem conjI` **from** 2, 1
4. $\vdash A \wedge B \rightarrow B \wedge A$ **by** `implies_intr` **from** 3

Each line in the above text represents a proof item. The line starts with the identifier of the proof item. The part before **by** is the computed sequent. The part after **by** displays the proof rule, additional arguments, and identifiers of input sequents. The proof rules `assume` and `implies_intr` are primitive deduction rules. The proof rule `apply_theorem` is the macro for applying a single theorem.

The following example illustrates identifiers when sub-proofs are present. The goal to be proved is $n + 0 = n$.

0. $\vdash 0 + 0 = 0$ **by** `rewrite_goal plus_def.1`, $\langle goal \rangle$
- 1.0. $\vdash _VAR\ n$ **by** `variable` $n :: nat$
- 1.1. $n + 0 = n \vdash n + 0 = n$ **by** `assume` $n + 0 = n$
- 1.2. $n + 0 = n \vdash Suc\ (n + 0) = Suc\ n$ **by** `rewrite_goal_with_prev` $\langle goal \rangle$ **from** 1.1
- 1.3. $n + 0 = n \vdash Suc\ n + 0 = Suc\ n$ **by** `rewrite_goal plus_def.2`, $\langle goal \rangle$ **from** 1.2
 1. $\vdash \forall n. n + 0 = n \rightarrow Suc\ n + 0 = Suc\ n$ **by** `intros` **from** 1.0, 1.1, 1.3
 2. $\vdash n + 0 = n$ **by** `apply_theorem_for` `nat_induct`, $\{P: \lambda n. n + 0 = n, x: n\}$ **from** 0, 1

Note the use of a trivial rule `variable` to designate new variables, and macro `intros` to introduce variables and assumptions. Items 1.1 to 1.3 should be read in the backward direction: the goal from induction is $Suc\ n + 0 = Suc\ n$. Rewriting using `plus_def.2` (inductive definition of $+$) changes it to $Suc\ (n + 0) = Suc\ n$, which is resolved by rewriting using the inductive hypothesis. In the arguments for proof rules, $\langle goal \rangle$ is an abbreviation for the goal statement.

This format for displaying linear proofs is still not so easy to read. We choose to use this format in this and the next section in order to show the workings of tactics and methods in more clarity. An improved format will be used in Section 4 and for the user interface.

2.5 Format for theories

In holpy, as in other proof assistants such as Isabelle and Coq, mathematical knowledge is organized as a collection of *theories*. Each theory imports a list of other theories, and may define new types, constants, and theorems. Proof of theorems are also contained in theories. The format for theories in holpy is based on JSON, hence holpy theory files have extension `.json`. They correspond to `.thy` files for Isabelle and `.v` files for Coq. Following the terminology of Isabelle/ML, we also use the word “theory” to mean the object containing the current state of mathematical knowledge (as in “the theory environment”). In the text below, which of the two meanings is used should be clear from context.

The content of a JSON file can be considered as an object made up from dictionaries, ordered lists, numbers, strings, booleans, and null. In our case, a theory is a dictionary with the following keys: `name` for the name of the theory, `imports` for the list of imported theories, `description` for overall comments on the theory, and `contents` which contains the main data. The value for `contents` is a list of items, where each

item can be a type, constant, definition, theorem, and so on. The content of each item depends on its type. In particular, theorem items should contain the proof of the theorem, stored as a linear proof.

Theorem attributes is another important information stored in theories. They are analogous to attributes in Isabelle. Each theorem is assigned a list of strings, indicating how the theorem is usually used. For example, the attribute `backward` means the theorem is usually applied in the backward direction. Attributes in `holpy` is used during the search for suggested actions, in order to limit the number of suggestions (Section 3.3).

Storing theories as a JSON file, rather than as a text file to be edited directly, makes implementation of user interfaces more difficult at first. However, it also creates more flexibility when designing the user interface. In particular, not all information in the JSON file has to be displayed. Information that is (currently) irrelevant to the proof can be hidden. Another advantage is that it is easier to develop other tools to analyze the theories – for example, to profile the performance of proof automation or search functionality. A simple application of this is to automatically produce the test results shown in Table 1.

2.6 Proofs with gaps

For a proof to be valid, all sequents appearing in the proof need to be justified. However, during the construction of a proof, it is often convenient to permit certain sequents to be left unproven, expecting it to be proved later. This is realized by the use of `sorry` in Isabelle and `admitted` in Coq. In `holpy`, we declare a special proof rule called `sorry`. In a linear proof, a proof item with rule `sorry` must contain an expected sequent (similar rule applies for proof terms). During proof checking, if the `no_gaps` flag is turned off, the result of a `sorry` item is its expected sequent, which can be used in subsequent proof items. Proof checking with `no_gaps` flag turned on (the usual setting when verifying a theory file) will raise an exception on encountering a `sorry` proof item.

2.7 Tactics

The notion of *tactics* in `holpy` is analogous, but not exactly the same, to tactics in Isabelle and Coq. In `holpy`, a tactic is a function taking a sequent to be proved, a list of input sequents, and possibly additional arguments (with fixed signature for each tactic), and returns a proof whose output is the target sequent (or raises an exception). The resulting proof may refer to input sequents or contain `sorry`s. Intuitively, a tactic converts the current goal (the sequent to be proved) to a list of subgoals (those proof items with rule `sorry`), possibly making use of other known facts (the input sequents).

We now give two representative examples.

Introduction The introduction tactic takes a goal in the `forall-implies` form, and introduces the variables and assumptions in a sub-proof. It takes as additional arguments the names of the new variables (and no input sequents). For example, given the goal

$$\vdash \forall n. n + 0 = n \longrightarrow \text{Suc } n + 0 = \text{Suc } n,$$

and name n for the new variable, the tactic returns the proof

```

0.0.  $\vdash \text{VAR } n$  by variable  $n :: \text{nat}$ 
0.1.  $n + 0 = n \vdash n + 0 = n$  by assume  $n + 0 = n$ 
0.2.  $n + 0 = n \vdash \text{Suc } x + 0 = \text{Suc } x$  by sorry
    0.  $\vdash \forall n. n + 0 = n \longrightarrow \text{Suc } n + 0 = \text{Suc } n$  by intros from 0.0, 0.1, 0.2

```

Applying a theorem Applying a theorem in the backward direction can be implemented as a tactic. For example, given the goal $A \wedge B \vdash B \wedge A$, a theorem name `conjI`, and no input sequents, the tactic produces the following proof:

```

0.  $A \wedge B \vdash B$  by sorry
1.  $A \wedge B \vdash A$  by sorry
2.  $A \wedge B \vdash B \wedge A$  by apply_theorem conjI from 0, 1

```

If $A \wedge B \vdash B$ is given as an input sequent, the resulting proof has only one `sorry`, and the invocation of `apply_theorem` refers to that input sequent.

3 Methods

The concept of *method* forms another level of abstraction over macros and tactics. They provide the direct link between proof automation in holpy and the user interface. Our definition of methods is analogous to that in Isabelle, but there are also some important differences.

In our design of the user interface, the *proof state* at any stage of the proof is simply a linear proof with gaps – proof items with rule `sorry`. These gaps can be considered as the remaining goals. A method defines a transform on the proof state, and the user conducts a proof by applying a series of methods. More precisely, a method is a function taking the following input:

- The current proof state.
- One selected goal in the proof state.
- A number of selected facts in the proof state (which must occur before the goal).
- Some additional arguments, with signature fixed by the method.

Given the inputs, a method either returns a new proof state or raises an exception. Unlike macros and tactics, the additional arguments for methods are always strings indexed by a set of keys (as determined by the method). Each method is responsible for parsing the input strings to the right kinds of objects (e.g. types and terms).

The above definition of methods is quite general. A method can literally make any change to the proof state. In practice, most methods fall into one of two common forms, corresponding to backward and forward reasoning. We now describe the two kinds of methods in more detail.

3.1 Backward reasoning

Methods for backward reasoning take the selected goal, and attempt to replace it by a number of simpler goals. These methods can be easily constructed from tactics. Given a tactic, the corresponding method performs the following actions:

1. Lookup the selected goal and facts in the proof state, to obtain the sequent to be proved and the list of input sequents.
2. Parse the input strings to the right kinds of objects (e.g. types and terms).
3. Apply the tactic on these inputs (and the theory environment of the proof), yielding a proof (possibly with holes) of the goal.
4. *Splice the proof into the proof state.* This involves changing the proof item for the goal so it is no longer a `sorry`, and possibly inserting proof items before the goal.

The last splicing process is easy to understand intuitively, but can be quite tricky to implement. Inserting proof items in the middle of a proof involves changing the identifiers in the output of the tactic, and also in the part of the proof state after the goal (if we wish to keep the identifiers in order). It also needs to link up references to input sequents in the output of the tactic.

We give two examples for illustration.

Introduction Consider the proof of $n + 0 = n$ by induction. After applying induction, we have the following proof state:

0. $\vdash 0 + 0 = 0$ **by** `sorry`
1. $\vdash \forall n. n + 0 = n \longrightarrow \text{Suc } n + 0 = \text{Suc } n$ **by** `sorry`
2. $\vdash n + 0 = n$ **by** `apply_theorem_for nat_induct, {P: $\lambda n. n + 0 = n, x: n$ }` **from** 0, 1

We invoke the method corresponding to the introduction tactic, with proof item 1 as the goal, and the string “n” as the additional argument for names (name of the new variable). The result is:

0. $\vdash 0 + 0 = 0$ **by** `sorry`
- 1.0. $\vdash \text{_VAR } n$ **by** `variable n :: nat`
- 1.1. $n + 0 = n \vdash n + 0 = n$ **by** `assume n + 0 = n`
- 1.2. $n + 0 = n \vdash \text{Suc } n + 0 = \text{Suc } n$ **by** `sorry`
1. $\vdash \forall n. n + 0 = n \longrightarrow \text{Suc } n + 0 = \text{Suc } n$ **by** `intros` **from** 1.0, 1.1, 1.2
2. $\vdash n + 0 = n$ **by** `apply_theorem_for nat_induct, {P: $\lambda n. n + 0 = n, x: n$ }` **from** 0, 1

Note how the output of the tactic (shown in Section 2.7) is converted to starting with identifier 1, and spliced into the proof state.

Applying a theorem For this example, we consider the proof of $A \wedge B \longrightarrow B \wedge A$. Suppose we are at the following intermediate stage of the proof:

0. $A \wedge B \vdash A \wedge B$ **by** `assume A \wedge B`
1. $A \wedge B \vdash B$ **by** `apply_theorem conjD2` **from** 0
2. $A \wedge B \vdash B \wedge A$ **by** `sorry`
3. $\vdash A \wedge B \rightarrow B \wedge A$ **by** `implies_intr` **from** 2

Invoking the method corresponding to backward application of a theorem, with proof item 2 as the selected goal, proof item 1 as (the only) selected fact, and `conjI` as additional argument for `theorem`, the result is:

0. $A \wedge B \vdash A \wedge B$ **by** `assume A \wedge B`
1. $A \wedge B \vdash B$ **by** `apply_theorem conjD2` **from** 0

2. $A \wedge B \vdash A$ **by** sorry
3. $A \wedge B \vdash B \wedge A$ **by** apply_theorem conjI **from** 1, 2
4. $\vdash A \wedge B \rightarrow B \wedge A$ **by** implies_intr **from** 3

Note how proof items 2 and 3 in the original proof state are automatically re-numbered, along with their references.

3.2 Forward methods

Methods for forward reasoning considers only the selected facts. It can be created directly from a macro: the selected facts become the input sequents to the macro, and the input strings are parsed to the arguments for the macro. The output of the macro is added in a new proof item directly in front of the selected goal.

For example, given the following initial proof state:

0. $A \wedge B \vdash A \wedge B$ **by** assume $A \wedge B$
1. $A \wedge B \vdash B \wedge A$ **by** sorry
2. $\vdash A \wedge B \rightarrow B \wedge A$ **by** implies_intr **from** 1

We invoke the method corresponding to the macro `apply_theorem`, with selected goal 1, selected fact 0, and argument `conjD2` for the theorem name. The resulting proof state is as follows.

0. $A \wedge B \vdash A \wedge B$ **by** assume $A \wedge B$
1. $A \wedge B \vdash B$ **by** apply_theorem conjD2 **from** 0
2. $A \wedge B \vdash B \wedge A$ **by** sorry
3. $\vdash A \wedge B \rightarrow B \wedge A$ **by** implies_intr **from** 2

Again, note the re-numbering of proof items 1 and 2 and their references after adding a new proof item before 1.

3.3 Search for suggestions

In addition to the function transforming the proof state, each method also provides a search function. The search function takes as input the current proof state, the selected goal, and a number of selected facts, and outputs a list of suggested invocations of the method. Each suggested invocation provides input strings for some (not necessarily all) of the required arguments.

The search function is an important part of our design. The outputs of search functions for all methods are combined to form the list of suggestions for the user. Hence, the search function should aim to return a suggestion whenever it is plausible, but should not return too many suggestions. For methods requiring no input strings, the search function may simply test whether the method can be applied on the selected goal and facts. For other methods, there may be more choices in the implementation.

For example, for methods applying a single theorem in the forward or backward direction, the search function iterates through theorems having the `forward` (resp. `backward`) attribute. For each theorem, it matches the selected facts and goal with the assumptions and conclusion of the theorem, and returns a suggestion whenever the match succeeds. Likewise, rewriting a fact (resp. goal) using a theorem can be searched for by matching the left side of each theorem having the `rewrite` attribute with parts of the selected fact (resp. goal).

4 Implementation

We implemented the above ideas in a web interface. In addition to the functionality for conducting a proof, the user interface also handles display and editing of theory files in its JSON format. Hence, it provides all of the necessary functionality for interactive theorem proving based on holpy. Figure 1 shows a screenshot of the user interface.

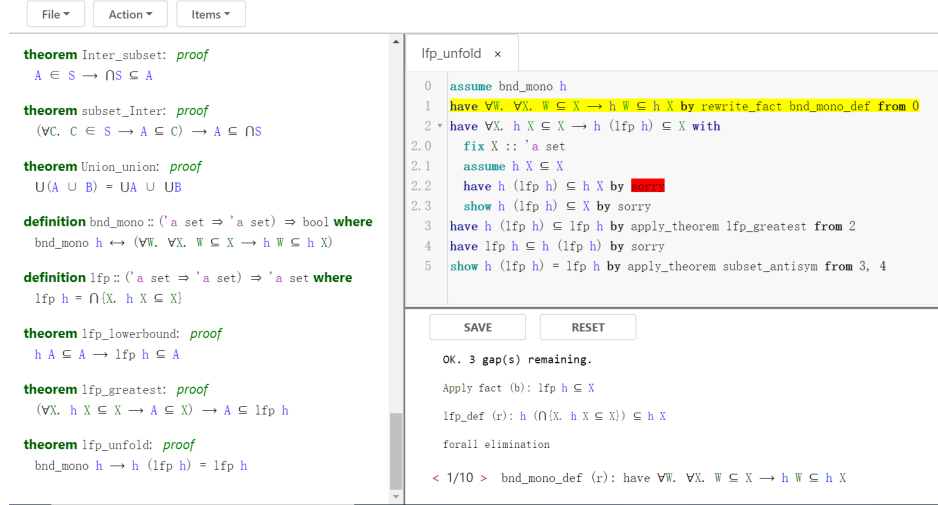


Fig. 1. Screenshot showing an intermediate stage in the proof of lfp_unfold.

The left side has three tabs displaying the list of theories, the content of the selected theory, and the list of variables in the current proof. On the right side, the top half displays the current state of the proof. The user can select goals and facts in the proof by clicking on the corresponding lines. The selected facts and goal are colored in yellow and red, respectively. After each change of selection, the user interface queries the holpy backend for the list of suggestions of method applications, which is displayed in the bottom half of the right side, one line for each suggestion. The user may perform one of the suggested actions by clicking on the corresponding line. If the suggestion does not provide all required arguments, the user is prompted to enter the missing arguments.

Occasionally, the user will want to invoke a method not among the suggestions. Two common methods that are not in the search are `cases` and `cut`. Both take a string which is parsed into a boolean term A . The `cases` method reduces the selected goal C into two goals $A \rightarrow C$ and $\neg A \rightarrow C$. The `cut` method inserts A as a new goal right before the current goal. When A is proved, it can be used in the proof of the original goal. The user can select invocation of these (and other) methods from the menu, and then enter the required arguments.

When displaying the proof, the user interface converts the proof to a more readable form compared to that used in Section 2 and 3. The basic transforms applied include:

- Use **fix** and **assume** for variable and assume rules.
- Hide antecedents of sequents (which can be inferred from previous **assumes**).
- Change invocations of **intros** to **with** blocks.
- Add **show** for the last sequent of a block, and **have** for other intermediate sequents.
- Indentation according to **with** blocks.

We tested the user interface on a selection of theorems from logic, sets, functions, Peano arithmetic, and lists. The results are given in Table 1. In the table, #S is the total number of steps to prove the theorem, #Y is the number of steps that are among the suggestions, and #N is the number of steps that must be selected from the menu. The results show that the current user interface is already applicable to a wide range of areas, allowing proofs of some basic results to be conducted largely by choosing from the suggestions.

5 Case study: Knaster-Tarski theorem

In this section, we show a case study on the Knaster-Tarski fixed point theorem, with the aim to show how user interaction works in practice for a nontrivial result. Roughly speaking, the theorem states that any bounded monotone function has a (smallest) fix-point. We state and prove a basic version of the theorem using our user interface.

The definition of bounded monotone functions is given as follows (here h is of type $'a \text{ set} \Rightarrow 'a \text{ set}$, and we assume the bound on h is given by the type $'a$).

$$\text{bnd_mono } h \longleftrightarrow (\forall W. \forall X. W \subseteq X \longrightarrow h W \subseteq h X)$$

Given a bounded monotone function, its least fixed point is constructed using the following definition:

$$\text{lfp } h = \bigcap \{X. h X \subseteq X\}$$

Two properties of $\text{lfp } h$ follow immediately from the definition. The first says that $\text{lfp } h$ is contained in any set A satisfying $h A \subseteq A$. The second says in order to show any set A is a subset of $\text{lfp } h$, it suffices to show A is a subset of any X satisfying $h X \subseteq X$. These properties are stated in higher-order logic as follows.

$$\text{lfp_lowerbound} : h A \subseteq A \longrightarrow \text{lfp } h \subseteq A$$

$$\text{lfp_greatest} : (\forall X. h X \subseteq X \longrightarrow A \subseteq X) \longrightarrow A \subseteq \text{lfp } h$$

The main theorem states that $\text{lfp } h$ is in fact a fixed point of h :

$$\text{lfp_unfold} : \text{bnd_mono } h \longrightarrow h (\text{lfp } h) = \text{lfp } h$$

We now show how to prove this theorem using our user interface. The initial state of the proof is:

```
0 assume bnd_mono h
1 show h (lfp h) = lfp h by sorry
```

First, select 0 as a fact, and apply the suggestion to rewrite the fact using `bnd_mono_def`. Next, select 1 as the goal (without selecting any facts), and use the suggestion to apply `subset_antisym`, to reduce the goal to two subset relations. The resulting state after these two operations is:

Name	Proposition	#S	#Y	#N
double_neg	$\neg\neg A \longleftrightarrow A$	9	8	1
disj_conv_imp	$\neg A \vee B \longleftrightarrow A \longrightarrow B$	12	11	1
ex_conj_distrib	$(\exists x. A x \wedge B x) \longrightarrow (\exists x. A x) \wedge (\exists x. B x)$	6	6	0
all_conj_distrib	$(\forall x. A x \wedge B x) \longrightarrow (\forall x. A x) \wedge (\forall x. B x)$	7	7	0
conj_disj_distribL1	$A \wedge (B \vee C) \longleftrightarrow A \wedge B \vee A \wedge C$	23	23	0
pierce	$((A \longrightarrow B) \longrightarrow A) \longrightarrow A$	5	4	1
drinker	$\exists x. P x \longrightarrow (\forall x. P x)$	11	8	3
subset_antisym	$A \subseteq B \longrightarrow B \subseteq A \longrightarrow A = B$	7	7	0
subset_trans	$A \subseteq B \longrightarrow B \subseteq C \longrightarrow A \subseteq C$	4	4	0
cantor	$\exists S. \forall x. \neg f x = S$	13	12	1
Inter_subset	$A \in S \longrightarrow \bigcap S \subseteq A$	4	4	0
subset_Inter	$(\forall C. C \in S \longrightarrow A \subseteq C) \longrightarrow A \subseteq \bigcap S$	6	6	0
Union_union	$\bigcup(A \cup B) = \bigcup A \cup \bigcup B$	43	43	0
lfp_lowerbound	$h A \subseteq A \longrightarrow \text{lfp } h \subseteq A$	3	3	0
lfp_greatest	$(\forall X. h X \subseteq X \longrightarrow A \subseteq X) \longrightarrow A \subseteq \text{lfp } h$	5	5	0
lfp_unfold	$\text{bnd_mono } h \longrightarrow h (\text{lfp } h) = \text{lfp } h$	10	9	1
fun_upd_triv	$(f)(a := f a) = f$	8	7	1
fun_upd_upd	$(f)(a := b, a := c) = (f)(a := c)$	9	8	1
fun_upd_twist	$\neg c = a \longrightarrow (f)(a := b, c := d) = (f)(c := d, a := b)$	19	17	2
comp_fun_assoc	$(f \circ g) \circ h = f \circ g \circ h$	4	4	0
injective_comp_fun	$\text{injective } f \longrightarrow \text{injective } g \longrightarrow \text{injective}(g \circ f)$	5	5	0
surjective_comp_fun	$\text{surjective } f \longrightarrow \text{surjective } g \longrightarrow \text{surjective}(g \circ f)$	11	9	2
add_comm	$x + y = y + x$	7	6	1
add_assoc	$x + y + z = x + (y + z)$	6	6	0
distrib_l	$x * (y + z) = x * y + x * z$	7	7	0
mult_assoc	$x * y * z = x * (y * z)$	7	6	1
mult_comm	$x * y = y * x$	7	6	1
less_eq_trans	$k \leq m \longrightarrow m \leq n \longrightarrow k \leq n$	9	9	0
append_right_neutral	$xs @ [] = xs$	5	5	0
append_assoc	$(xs @ ys) @ zs = xs @ ys @ zs$	6	6	0
length_append	$\text{length}(xs @ ys) = \text{length } xs + \text{length } ys$	9	9	0
rev_append	$\text{rev}(xs @ ys) = \text{rev } ys @ \text{rev } xs$	9	8	1
rev_rev	$\text{rev}(\text{rev } xs) = xs$	12	12	0
rev_length	$\text{length}(\text{rev } xs) = \text{length } xs$	10	10	0
Total: 34 theorems		318	300	18

Table 1. Statistics on the test suite.

```

0 assume bnd_mono h
1 have  $\forall W. \forall X. W \subseteq X \longrightarrow h W \subseteq h X$  by rewrite_fact bnd_mono_def from 0
2 have  $h(\text{lf}p\ h) \subseteq \text{lf}p\ h$  by sorry
3 have  $\text{lf}p\ h \subseteq h(\text{lf}p\ h)$  by sorry
4 show  $h(\text{lf}p\ h) = \text{lf}p\ h$  by apply_theorem subset_antisym from 2, 3

```

Next, select item 2 and follow the suggestion to apply `lf_p_greatest`. This results in a `forall` goal. Select the goal and the introduction method, and enter X for the name of the new variable, we get the following proof state:

```

0 assume bnd_mono h
1 have  $\forall W. \forall X. W \subseteq X \longrightarrow h W \subseteq h X$  by rewrite_fact bnd_mono_def from 0
2 have  $\forall X. h X \subseteq X \longrightarrow h(\text{lf}p\ h) \subseteq X$  with
2.0 fix  $X :: 'a\ \text{set}$ 
2.1 assume  $h X \subseteq X$ 
2.2 show  $h(\text{lf}p\ h) \subseteq X$  by sorry
3 have  $h(\text{lf}p\ h) \subseteq \text{lf}p\ h$  by apply_theorem lf_p_greatest from 2
4 have  $\text{lf}p\ h \subseteq h(\text{lf}p\ h)$  by sorry
5 show  $h(\text{lf}p\ h) = \text{lf}p\ h$  by apply_theorem subset_antisym from 3, 4

```

Next, we perform the only manual step in this proof, inserting an intermediate goal $h(\text{lf}p\ h) \subseteq h X$ before $h(\text{lf}p\ h) \subseteq X$. The resulting proof state is (now showing only the block for proof item 2):

```

2 have  $\forall X. h X \subseteq X \longrightarrow h(\text{lf}p\ h) \subseteq X$  with
2.0 fix  $X :: 'a\ \text{set}$ 
2.1 assume  $h X \subseteq X$ 
2.2 have  $h(\text{lf}p\ h) \subseteq h X$  by sorry
2.3 show  $h(\text{lf}p\ h) \subseteq X$  by sorry

```

Next, select goal 2.2 and fact 1, and follow the suggestion to apply fact 1 to goal 2.2, resulting in new goal $\text{lf}p\ h \subseteq X$. We get:

```

2 have  $\forall X. h X \subseteq X \longrightarrow h(\text{lf}p\ h) \subseteq X$  with
2.0 fix  $X :: 'a\ \text{set}$ 
2.1 assume  $h X \subseteq X$ 
2.2 have  $\text{lf}p\ h \subseteq X$  by sorry
2.3 have  $h(\text{lf}p\ h) \subseteq h X$  by apply_fact_for lf_p_h, X from 1, 2.2
2.4 show  $h(\text{lf}p\ h) \subseteq X$  by sorry

```

Select item 2.2, the user interface suggests use of the theorem `lf_p_lowerbound`, reducing to the goal $h X \subseteq X$, which is already available as a fact. This proves 2.2. Next, select goal 2.4 and fact 2.2, the user interface suggests use of the theorem `subset_trans`, again reducing to the goal $h X \subseteq X$. Performing these two steps finishes the proof of 2. The resulting proof state is:

```

2 have  $\forall X. h X \subseteq X \longrightarrow h(\text{lf}p\ h) \subseteq X$  with
2.0 fix  $X :: 'a\ \text{set}$ 
2.1 assume  $h X \subseteq X$ 
2.2 have  $\text{lf}p\ h \subseteq X$  by apply_theorem_for lf_p_lowerbound, ... from 2.1
2.3 have  $h(\text{lf}p\ h) \subseteq h X$  by apply_fact_for lf_p_h, X from 1, 2.2
2.4 show  $h(\text{lf}p\ h) \subseteq X$  by apply_theorem subset_trans from 2.3, 2.1

```

Two more steps are needed to finish the overall proof: reducing goal 4 to showing $h(h(\text{lfp } h)) \subseteq h(\text{lfp } h)$ using `lfp_lowerbound`, then using item 1 and 3 to solve the goal. The user interaction is similar to before. The final state of the proof is:

```

0 assume bnd_mono h
1 have  $\forall W. \forall X. W \subseteq X \longrightarrow h W \subseteq h X$  by rewrite_fact bnd_mono_def from 0
2 have  $\forall X. h X \subseteq X \longrightarrow h(\text{lfp } h) \subseteq X$  with
2.0 fix  $X :: 'a \text{ set}$ 
2.1 assume  $h X \subseteq X$ 
2.2 have  $\text{lfp } h \subseteq X$  by apply_theorem_for lfp_lowerbound, ... from 2.1
2.3 have  $h(\text{lfp } h) \subseteq h X$  by apply_fact_for lfp h, X from 1, 2.2
2.4 show  $h(\text{lfp } h) \subseteq X$  by apply_theorem subset_trans from 2.3, 2.1
3 have  $h(\text{lfp } h) \subseteq \text{lfp } h$  by apply_theorem lfp_greatest from 2
4 have  $h(h(\text{lfp } h)) \subseteq h(\text{lfp } h)$  by apply_fact_for ... from 1, 3
5 have  $\text{lfp } h \subseteq h(\text{lfp } h)$  by apply_theorem_for lfp_lowerbound, ... from 4
6 show  $h(\text{lfp } h) = \text{lfp } h$  by apply_theorem subset_antisym from 3, 5

```

As we can see, the resulting proof is quite readable, similar to a proof using Isabelle/Isar. All intermediate conclusions are shown, as well as each theorem and proof rule used. However, the entire proof is constructed just with a few clicks, occasionally entering names of variables, instantiations (when it cannot be derived by matching), and intermediate goals.

6 Conclusion

In this paper, we presented a framework for designing point-and-click user interfaces in interactive theorem proving. While a major goal of the design is ease of learning for newcomers to this field, we also intend to produce a fully functional system, able to be used for general purpose theorem proving. We implemented a prototype user interface based on this framework, and tested it on theorems about logic, sets, functions, Peano arithmetic, and lists, showing that these theorems can be proved largely by clicking on suggestions, and occasionally entering additional information.

We intend the current work to be the beginning of a long-term project to build a proof assistant that is both easy-to-use and scalable to large formalizations. On the user interface side, we envision two major next steps. First, we currently lack strong proof automation in the system. Proof assistants such as Isabelle benefit from powerful tactics (such as `auto` and `blast`), as well as calls to external provers via Sledgehammer. In the future, we intend to incorporate both powerful internal automation, as well as connections to external provers. They can be easily fit into the current framework: the user selects the goal and a number of facts to use, and the system invokes proof automation in the background to check whether the goal can be solved using the selected facts.

Second, we currently make no attempt to order the suggestions. This does not pose a problem so far, since the test examples are still in the beginning stages of mathematical development, so there are few options at each step. As we move to formalizing deeper mathematical theories, it is expected that the number of options at each step will increase, even as we try to control it by having the user annotate the theorems and select which facts to use. One potentially promising approach is to use machine learning models for ordering the suggestions.

References

1. The HOL 4 system. <http://hol.sourceforge.net/>
2. Abrial, J., Cansell, D.: Click'n prove: Interactive proofs within set theory. In: Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003, Rom, Italy, September 8-12, 2003, Proceedings. pp. 1–24 (2003)
3. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer (2016)
4. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004)
5. Bertot, Y., Kahn, G., Théry, L.: Proof by pointing. In: Theoretical Aspects of Computer Software, International Conference TACS '94, Sendai, Japan, April 19-22, 1994, Proceedings. pp. 141–160 (1994)
6. Bertot, Y., Théry, L.: A generic approach to building user interfaces for theorem provers. *J. Symb. Comput.* **25**(2), 161–194 (1998)
7. Breitner, J.: Visual theorem proving with the incredible proof machine. In: Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings. pp. 123–139 (2016)
8. The JSON data interchange syntax. <http://ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf> (12 2017)
9. Fulton, N., Mitsch, S., Quesel, J., Völpl, M., Platzer, A.: Keymaera X: an axiomatic tactical theorem prover for hybrid systems. In: Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings. pp. 527–538 (2015)
10. Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Roux, S.L., Mahboubi, A., O'Connor, R., Biha, S.O., Pasca, I., Rideau, L., Solovyev, A., Tassi, E., Théry, L.: A machine-checked proof of the odd order theorem. In: Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings. pp. 163–179 (2013)
11. Hales, T., Adams, M., Bauer, G., Dang, T.T., Harrison, J., Hoang, L.T., Kaliszyk, C., Magron, V., McLaughlin, S., Nguyen, T.T., et al.: A formal proof of the kepler conjecture. *Forum of Mathematics, Pi* **5**, e2 (2017)
12. Harrison, J.: HOL light: An overview. In: Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings. pp. 60–66 (2009)
13. Klein, G., Andronick, J., Elphinstone, K., Murray, T.C., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.* **32**(1), 2:1–2:70 (2014)
14. Leroy, X.: Formal verification of a realistic compiler. *Commun. ACM* **52**(7), 107–115 (2009)
15. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer (2002)
16. Platzer, A., Quesel, J.: Keymaera: A hybrid theorem prover for hybrid systems (system description). In: Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings. pp. 171–178 (2008)
17. Wenzel, M.: Isar - A generic interpretative approach to readable formal proof documents. In: Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings. pp. 167–184 (1999)
18. Zhan, B.: holpy: Interactive Theorem Proving in Python. arXiv e-prints arXiv:1905.05970 (May 2019)