LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL *

Yutaka Nagashima[1 2][0000−0001−6693−5325]

[1] Czech Technical University in Prague, Prague, Czech Republic
Yutaka.Nagashima@cvut.cz
[2] University of Innsbruck, Innsbruck, Austria

Abstract. Proof assistants, such as Isabelle/HOL, offer tools to facilitate inductive theorem proving. Isabelle experts know how to use these tools effectively; however they did not have a systematic way to encode their expertise. To address this problem, we present our domain-specific language, `LiFtEr`. `LiFtEr` allows experienced Isabelle users to encode their induction heuristics in a style independent of any problem domain. `LiFtEr`'s interpreter mechanically checks if a given application of induction tool matches the heuristics, thus transferring experienced users' expertise to new Isabelle users.

Keywords: Induction · Isabelle/HOL · Domain-Specific Language.

1 Introduction

Consider the following reverse functions, `rev` and `itrev`, from literature [18]:

```
primrec rev::"'a list =>'a list" where
  "rev  []      = []"
| "rev (x # xs) = rev xs @ [x]"

fun itrev::"'a list =>'a list =>'a list" where
  "itrev  []    ys = ys"
| "itrev (x#xs) ys = itrev xs (x#ys)"
```

where `#` is the list constructor, and `@` appends two lists. How do you prove the following lemma?

```
lemma "itrev xs ys = rev xs @ ys"
```

Since both `rev` and `itrev` are defined recursively, it is natural to imagine that we can handle this problem by applying induction. But how do you apply induction and why? What induction heuristics do you use? In which language do you describe those heuristics?

* This work was supported by the European Regional Development Fund under the project AI & Reasoning (reg. no.CZ.02.1.01/0.0/0.0/15_003/0000466).

Modern proof assistants (PAs), such as Isabelle/HOL [18], are forming the basis of trustworthy software. Klein *et al.*, for example, verified the correctness of the seL4 micro-kernel in Isabelle/HOL [7], whereas Leroy developed a certifying C compiler, CompCert, using Coq [10]. Despite the growing number of such complete formal verification projects, the limited progress in proof automation still keeps the cost of proof development high, thus preventing the widespread adoption of complete formal verification.

A noteworthy approach in proof automation for proof assistants is the so-called hammer tools [1]. Sledgehammer [2], for example, exports proof goals in Isabelle/HOL to various external automated theorem provers (ATPs) to exploit the state-of-the-art proof automation of these backend provers; however, the discrepancies between the polymorphic higher-order logic of Isabelle/HOL and the monomorphic first-order logic of the backend provers severely impair sledgehammer's performance when it comes to inductive theorem proving (ITP).

This is unfortunate for two reasons. First, many Isabelle users chose Isabelle/HOL precisely because its higher-order logic is expressive enough to specify mathematical objects and procedures involving recursion without introducing new axioms. Second, induction lies at the heart of mathematics and computer science. For instance, induction is often necessary for reasoning about natural numbers, recursive data-structures, such as lists and trees, computer programs containing recursion and iteration [3].

This way ITP remains as a long-standing challenge in computer science, and its automation is much needed. Facing the limited automation in ITP, Gramlich surveyed the problems in ITP and presented the following prediction in 2005:

> in the near future, ITP will only be successful for very specialized domains for very restricted classes of conjectures. ITP will continue to be a very challenging engineering process.

We address this conundrum with our domain-specific language, `LiFtEr`. `LiFtEr` allows experienced Isabelle users to encode their induction heuristics in a style independent of problem domains. `LiFtEr`'s interpreter mechanically checks if a given application of induction is compatible with the induction heuristics written by experienced users. Our research hypothesis is that:

> it is possible to encode valuable induction heuristics for Isabelle/HOL in `LiFtEr` and such heuristics can be valid across diverse problem domains, because `LiFtEr` allows for meta-reasoning on applications of induction methods, without relying on concrete proof goals, their underlying proof states, nor concrete applications of induction methods.

In the rest of the paper, we first review how induction works in Isabelle in Section 2. Then, we give the overview of `LiFtEr` and its syntax in Section 3. Section 4 presents six small example assertions written in `LiFtEr` and demonstrates how to write induction heuristics for our ongoing example about `rev` and `itrev`. Section 5 shows that the `LiFtEr` assertions from Section 4 are applicable to an inductive problem in a completely unrelated problem domain. Then, Section 6 reveals `LiFtEr`'s internal pre-processing stage, which allowed for intuitive reasoning about inductive problems. We compare `LiFtEr` with other work for inductive theorem proving in Section 7 before summarizing our contributions and future work in Section 8. Our working prototype is available at GitHub [16].

2 Background

To handle inductive problems, modern proof assistants offer tools to apply induction. For example, Isabelle comes with the `induct` proof method and the `induction` method [3]. For example, Nipkow *et al.* proved our ongoing example as following [17]:

```
lemma model_prf:"itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys) by auto
```

Namely, they applied structural induction on `xs` while generalizing `ys` before applying induction by passing the string `ys` to the `arbitrary` field. The resulting sub-goals are as following:

```
1. !!ys. itrev [] ys = rev [] @ ys
2. !!a xs ys. (!!ys. itrev xs ys = rev xs @ ys) ==>
              itrev (a # xs) ys = rev (a # xs) @ ys
```

where `!!` is the universal quantifier and `==>` is the implication in Isabelle's meta-logic. Due to the generalization, the `ys` in the induction hypothesis is quantified within the hypothesis, and it is differentiated from the `ys` that appears in the conclusion. Had Nipkow *et al.* omitted `arbitrary: ys`, the first sub-goal would be the same, but the second sub-goal would have been as following:

```
 2. !!a xs. itrev xs ys = rev xs @ ys ==>
           itrev (a # xs) ys = rev (a # xs) @ ys
```

Since the same `ys` is shared by the induction hypothesis and the conclusion, the subsequent application of `auto` fails to discharge this sub-goal.

It is worth noting that in general there are multiple equivalently appropriate combinations of arguments to prove a given inductive problem. For instance, the following proof snippet shows an alternative proof script for our example:

```
lemma alt_prf:"itrev xs ys = rev xs @ ys"
  apply(induct xs ys rule:itrev.induct) by auto
```

Here we passed the `itrev.induct` rule to the `rule` field of the `induct` method and proved the lemma by recursion induction [4] over `itrev`. This rule was derived by Isabelle automatically when we defined `itrev`, and it states the following:

[3] Proof methods are the Isar syntactic layer of LCF-style tactics.
[4] Recursion induction is also known as functional induction.

```
(!!ys. P [] ys) ==>
(!!x xs ys. P xs (x # ys) ==> P (x # xs) ys) ==>
P a0 a1
```

Essentially, this rule states that to prove a property `P` of `a0` and `a1` we have to prove it for two cases where `a0` is the empty list and the list with at least two elements. When the `induct` method takes this rule and `xs` and `ys` as induction variables, Isabelle produces the following sub-goals:

```
 1. !!ys. itrev [] ys = rev [] @ ys
 2. !!x xs ys. itrev xs (x # ys) = rev xs @ x # ys ==>
              itrev (x # xs) ys = rev (x # xs) @ ys
```

where the two sub-goals correspond to the two clauses in the definition of `itrev`.

There are other lesser-known techniques to handle difficult inductive problems using the `induct` method, and sometimes users have to develop useful auxiliary lemmas manually; however, for most cases the problem of how to apply induction often boils down to the the following three questions:

- On which terms do we apply induction?
- Which variables do we generalize?
- Which rule do we use for recursion induction?

Isabelle experts resort to induction heuristics to answer such questions and decide what arguments to pass to the `induct` method; however, such reasoning still requires human engineers to carefully investigate the inductive problem at hand. Moreover, Isabelle experts' induction heuristics are sparsely documented across various documents, and there was no way to encode their heuristics in programs. For the wide spread adoption of complete formal verification, we need a program language to encode such heuristics and the system to check if an invocation of the `induct` method written by an Isabelle novice complies with such heuristics.

3 Overview and Syntax

We designed `LiFtEr` to encode induction heuristics as assertions on invocations of the `induct` method in Isabelle/HOL. An assertion written in `LiFtEr` takes the pair of proof goals at hand together with their underlying proof state and arguments passed to the `induct` method. When one applies a `LiFtEr` assertion to an invocation of the `induct` method, `LiFtEr`'s interpreter returns a boolean value as the result of the assertion applied to the proof goals and their underlying proof state.

The goal of a `LiFtEr` programmer is to write assertions that implement reliable heuristics. A heuristic encoded as a `LiFtEr` assertion is reliable when it satisfies the following two properties: first, the `LiFtEr` interpreter is likely to evaluate the assertion to `true` when the arguments of the `induct` method are appropriate for the given proof goal. Second, the interpreter is likely to evaluate the assertion to `false` when the arguments are inappropriate for the goal.

Program 1 The Syntax of LiFtEr.

```
datatype numb    = Numb    of int;
datatype trm     = Trm     of int;
datatype rule    = Rule    of int;
datatype trm_occ = Trm_Occ of int;
datatype pattern = All_Only_Var | All_Constr | Mixed;
datatype assrt   =
(*quantifiers*)
  All_Ind                of trm  * assrt
| All_Arb                of trm  * assrt
| All_Trm                of trm  * assrt
| All_Rule               of rule * assrt
| All_Numb               of numb * assrt
| Some_Ind               of trm  * assrt
| Some_Arb               of trm  * assrt
| Some_Trm               of trm  * assrt
| Some_Rule              of rule * assrt
| Some_Numb              of numb * assrt
| All_Trm_Occ            of trm_occ * assrt
| Some_Trm_Occ           of trm_occ * assrt
| All_Trm_Occ_Of         of trm_occ * trm * assrt
| Some_Trm_Occ_Of        of trm_occ * trm * assrt
(*combinators*)
| And                    of assrt * assrt
| Or                     of assrt * assrt
| Not                    of assrt
| True
| Imply                  of assrt * assrt
(*atomic about proof goal*)
| Is_Rule_Of             of rule    * trm_occ
| Trm_Occ_Is_Of_Trm      of trm_occ * trm
| Are_Same_Trm           of trm     * trm
| Is_In_Trm_Loc          of trm_occ * trm_occ
| Is_Atom                of trm_occ
| Is_Cnst                of trm_occ
| Is_Recursive_Cnst      of trm_occ
| Is_Var                 of trm_occ
| Is_Free                of trm_occ
| Is_Bound               of trm_occ
| Is_Lambda              of trm_occ
| Is_App                 of trm_occ
| Is_An_Arg_Of           of trm_occ * trm_occ
| Is_Nth_Arg_Of          of trm_occ * numb * trm_occ
| Is_Nth_Ind             of trm     * numb
| Is_Nth_Arb             of trm     * numb
| Pattern                of numb * trm_occ * pattern
| Is_At_Deepest          of trm_occ
...
```

Program 1 shows the essential part of `LiFtEr`'s syntax. `LiFtEr` has five types of variables: `numb`, `rule`, `trm`, `trm_occ`, and `pattern`. A value of type `numb` is a natural number from 0 to the maximum of one of the following two numbers: the number of terms appearing in the proof goals at hand, and the maximum arity of constants appearing in the proof goals. A value of type `rule` corresponds to a name of an auxiliary lemma passed to the `induct` method as an argument in the `arbitrary` field.

The difference between `trm` and `trm_occ` is crucial: a value of `trm` is a term appearing in the proof goals, whereas a value of `trm_occ` is an *occurrence* of such terms. It is important to distinguish terms and term occurrences because the `induct` method in Isabelle/HOL only allows its users to specify induction terms but it does not allow us to specify on which occurrences of such terms we intend to apply induction.

The connectives, `And`, `Or`, `Not`, and `Imply` correspond to conjunction, disjunction, negation, and implication in the classical logic, respectively; And `Imply` admits the principle of explosion.

`LiFtEr` has 12 essential quantifiers and two quantifiers as syntactic sugars. Those starting with the string `All` are universal quantifiers, and those with `Some` are existential quantifiers. Again, it is important to notice the difference between the quantifiers over `trm` and the ones over `trm_occ`: for example, `All_Trm` quantifies all sub-terms appearing in the proof goals, whereas `All_Trm_Occ` quantifies all *occurrences* of such sub-terms. Quantifiers that end with the string `Ind` quantify over all induction terms passed to the `induct` method as induction terms, while quantifiers that end with the string `Arb` quantify over all terms passed to the `induct` method as arguments of the `arbitrary` field.

Some atomic assertions judge properties of term occurrences, and some judge the syntactic structure of proof goals with respects to certain terms, their occurrences, or certain numbers. While most atomic assertions workn on the syntactic structures of proof goals, `Pattern` provides a means to describe a limited amount of semantic information of proof goals since it checks how terms are defined. Section 4 explains the meaning of important atomic assertions through examples.

Attentive readers may have noticed that `LiFtEr`'s syntax does not cover any user-defined types or constants. This absence of specific types and constants is our intentional choice to promote induction heuristics that are valid across various problem domains: it encourages `LiFtEr` users to write heuristics that are not specific to particular data-types or functions. And `LiFtEr`'s interpreter can check if an application of the `induct` method is compatible with a given `LiFtEr` heuristic even if the proof goal involves user-defined data-types and functions even though such types and functions are unknown to the `LiFtEr` developer or the author of the heuristic but come into existence in the future only after developing `LiFtEr` and such heuristic.

4 LiFtEr by Example

This section illustrates how to use those atomic assertions and quantifiers to encode induction heuristics through examples.

4.1 Example 1: Induction terms should not be constants.

Let us revise the first example about the equivalence of two reverse functions, `itrev` and `rev`. One naive induction heuristic would be *"any induction term should not be a constant"* [5] In `LiFtEr`, we can encode this heuristic as the following assertion:

```
All_Ind (Trm 1,
 Some_Trm_Occ (Trm_Occ 1,
   Trm_Occ_Is_Of_Trm (Trm_Occ 1, Trm 1)
  And
   Not (Is_Cnst (Trm_Occ 1)))): assrt;
```

Note the use of `All_Ind` and `Some_Trm_Occ`: when `LiFtEr` handles induction terms, `LiFtEr` treats them as terms, but it is often necessary to analyze the *occurrences* of these terms in the proof goal to decide how to apply induction. In our example lemma, `xs` is a variable, which appears twice: once as the first argument of `itrev`, and once as the first argument of `rev`. With this mind, the above assertion reads as following:

> for all induction terms, named `Trm 1`, there exists a term occurrence, named `Trm_Occ 1`, such that `Trm_Occ 1` is an occurrence of `Trm 1` and `Trm_Occ 1` is not a constant.

Now we compare this heuristic with the model proof by Nipkow *et al.*

The only induction term, `xs`, has two occurrences in the proof goal both as variables. Therefore, if we apply this `LiFtEr` assertion to the model solution, `LiFtEr`'s interpreter acknowledges that the model solution complies with the induction heuristics defined above.

It is a common practice to analyze occurrences of specific terms when describing induction heuristics. Therefore, we introduced two pieces of syntactic sugars to avoid boilerplate code: `Some_Trm_Occ_Of` and `All_Trm_Occ_Of`. Both `Some_Trm_Occ_Of` and `All_Trm_Occ_Of` quantify over term occurrences of a particular term rather than all term occurrences in the proof goal at hand. Using `Some_Trm_Occ_Of`, we can shrink the above assertion from 5 lines to 3 lines as following:

```
All_Ind (Trm 1,
 Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
  Not (Is_Cnst (Trm_Occ 1)))): assrt;
```

[5] This *naive heuristic* is not very reliable: there are cases where the `induct` method takes terms involving constants and apply induction appropriately by automatically introducing induction variables. See Concrete Semantics [17] for more details.

In English, this reads as following:

> for all induction terms, named `Trm 1`, there exists an occurrence of `Trm 1`, named `Trm_Occ 1`, such that `Trm_Occ 1` is not a constant.

4.2 Example 2. Induction terms should appear at the bottom of syntax trees.

Not applying induction on a constant would sound a plausible heuristic, but such heuristic is not very useful.

In this example, we encode an induction heuristic that analyzes not only the properties of the induction terms but also the location of their occurrences within the proof goal at hand. When attacking inductive problems with many variables, it is sometimes a good attempt to apply induction on variables that appear at the bottom of the syntax tree representing the proof goal. We encode such heuristic using `Is_At_Deepest` as the following `LiFtEr` assertion:

```
All_Ind (Trm 1,
 Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
   Is_Atom (Trm_Occ 1)
  Imply
   Is_At_Deepest (Trm_Occ 1)));
```

In English, this assertion reads as following:

> for all induction terms, named `Trm 1`, there exists an occurrence of `Trm 1`, named `Trm_Occ 1`, such that if `Trm_Occ 1` is an atomic term then `Trm_Occ 1` lies at the deepest layer in the syntax tree that represents the proof goal.

We used the infix operator, `Imply`, to add the condition that we consider only the induction terms that are atomic terms. An atomic term is either a constant, free variable, schematic variable, or variable bound by a lambda abstraction. We added this condition because it makes little sense to check if the induction term resides at the bottom of the syntax tree when an induction term is not an atomic term, but a compound term: such compound terms have sub-terms at lower layers.

`LiFtEr`'s interpreter acknowledges that the model solution provided by Nipkow *et al.* complies with this heuristic when applied to this lemma: There is only one induction term, `xs`, and `xs` appears as an argument of `rev` on the right-hand side of the equation in the lemma at the lowest layer of this syntax tree.

4.3 Example 3. All induction terms should be arguments of the same occurrence of a recursively defined function.

Probably, it is more meaningful to analyze where induction terms reside in the proof goal with respects to other terms in the goal. More specifically, one heuristic for promising application of induction would be *"apply induction on terms that appear as arguments of the same occurrence of a recursively defined function"*. We encode this heuristic using LiFtEr's atomic assertions, Is_Atomic_Cnst and Is_An_Arg_Of, as following:

```
Some_Trm (Trm 1,
 Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
  All_Ind (Trm 2,
   Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
     Is_Recursive_Cnst (Trm_Occ 1)
    And
     (Trm_Occ 2 Is_An_Arg_Of Trm_Occ 1)))));
```

where Is_Recursive_Cnst checks if a constant is defined recursively or not, and Is_An_Arg_Of takes two term occurrences and checks if the first one is an argument of the second one.

Note that using Is_Recursive_Cnst this assertion checks not only the syntactic information of the proof goal at hand, but it also extracts an essential part of the semantic information of constants appearing in the goal, by investigating how these constants are defined in the underlying proof context.

As a whole, this assertion reads as following:

> there exists a term, named Trm 1, such that there exists an occurrence of Trm 1, named Trm_Occ 1, such that for all induction terms, named Trm 2, there exists an occurrence of Trm 2, named Trm_Occ 2, such that Trm_Occ 1 is defined recursively and Trm_Occ 2 appears as an argument of Trm_Occ 1.

Attentive readers may have noticed that we quantified over induction terms within the quantification over Trm_Occ 1, so that this induction heuristics checks if all induction terms occur as arguments of the same constant.

The LiFtEr interpreter confirms that the model proof is compatible with this heuristic as well: the constant, itrev, is defined recursively and has an occurrence that takes the only induction variable xs as the first argument.

4.4 Example 4. One should apply induction on the nth argument of a function where the nth parameter in the definition of the function always involves a data constructor.

The previous example checks if all induction terms are arguments of the same occurrence of a recursively defined function. Sometimes we can even estimate on which arguments of such function we should apply induction by inspecting the definitions of the function more carefully.

We introduce three constructs to support such reasoning: Is_Nth_Arg_Of, Is_Nth_Ind, and Pattern. Is_Nth_Arg_Of takes a term occurrence, a number, and another term occurrence, and it checks if the first term occurrence is the nth argument of the second term occurrence where counting starts at 0. Is_Nth_Ind takes a term occurrence and a number and checks if the term is passed to the `induct` method as the nth induction term. `Pattern` takes a term occurrence, a number, one of three *patterns*, `All_Only_Var`, `All_Const`, and `Mixed`. Each of such patterns describes how the term is defined.

For example, `Pattern (Numb n, Trm_Occ m, All_Only_Var)` denotes that the nth parameter is always a variable on the left-hand side of the definition of the term that has the term occurrence, `Trm_Occ m`. Likewise, `All_Const` denotes the case where the corresponding parameter of the definition of a particular constant always involves a data constructor, whereas `Mixed` denotes that the corresponding parameter is a variable in some clauses but involves a data constructor in other clauses. With these atomic assertions in mind, we write the following `LiFtEr` assertion:

```
 Not (Some_Rule (Rule 1, True))
Imply
 Some_Trm (Trm 1,
  Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    Is_Recursive_Cnst (Trm_Occ 1)
   And
    All_Ind (Trm 2,
     Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
      Some_Numb (Numb 1,
        Pattern (Numb 1, Trm_Occ 1, All_Constr)
       And
        Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1))))));
```

This roughly translates to the following English sentence:

> if there is no argument in the `rule` field in the `induct` method, then there exists a recursively defined constant, `Trm 1`, with an occurrence, `Trm_Occ 1`, such that for all induction terms `Trm 2`, there exists an occurrence, `Trm_Occ 2`, of `Trm 2`, such that there exists a number, `Numb 1`, such that the `(Numb 1)`th parameter involves a data constructor in all the clauses of the definition of `Trm 1`, and `Trm_Occ 2` appears as the `(Numb 1)`th argument of `Trm_Occ 1` in the proof goal.

Note that we added `Not(Some_Rule(Rule 1,True))` to focus on the case where the `induct` method does not take any auxiliary lemma in the `rule` field, since this heuristic is known to be less reliable when there is an auxiliary lemma passed to the `induct` method. Furthermore, it is important to be aware that `1` in `Numb 1` is merely the identifier of this variable, and the value of `Numb 1` can be a value that is not 1.

`LiFtEr`'s interpreter confirms that Nipkow's model solution to the lemma about `itrev` and `rev` conforms to this heuristic: there exists an occurrence of `itrev`, such that `itrev` is recursively defined and for the only induction term, `xs`, there is an occurrence of `xs` on the left-hand side of the proof goal, such that `itrev`'s first parameter involves data constructor in all clauses of its definition, and this occurrence of `xs` appears as the first argument of the occurrence of `itrev` [6].

4.5 Example 5. Induction terms should appear as arguments of a function that has a related `.induct` rule in the `rule` field.

When the `induct` method takes an auxiliary lemma in the `rule` field that Isabelle automatically derives from the definition of a constant, it is often true that we should apply induction on terms that appear as arguments of an occurrence of such constant.

See, for example, our alternative proof, `alt_prf`, for our ongoing example theorem. When Nipkow *et al.* defined the `itrev` function with the `fun` keyword, Isabelle automatically derived the auxiliary lemma `itrev.induct`, and the occurrence of `itrev` on the left-hand side of the equation takes `xs` and `ys` as its arguments. Furthermore, the alternative proof passes `xs` and `ys` to the `rule` field in the same order they appear as the arguments of the occurrence of `itrev` in the proof goal.

We introduce `Is_Rule_Of` to relate a term occurrence with an auxiliary lemma passed to the `rule` field. `Is_Rule_Of` takes a term occurrence and an auxiliary lemma in the `rule` field of the `induct` method, and it checks if the rule was derived by Isabelle at the time of defining the term. Moreover, we introduce `Is_Nth_Ind`, which let us specify the order of induction terms passed to the `induct` method. Using these constructs, we can encode the aforementioned heuristic as following:

```
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,
  Some_Trm (Trm 1,
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
     (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
     (All_Ind (Trm 2,
      (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
        Some_Numb (Numb 1,
          Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
         And
          (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

As a whole this `LiFtEr` assertion checks if the following holds:

> if there exists a rule, `Rule 1`, in the `rule` field of the `induct` method, then there exists a term `Trm 1` with an occurrence `Trm_Occ 1`, such that `Rule 1` is derived by Isabelle when defining `Trm 1`, and for all induction

[6] Note that in reality the counting starts at 0 internally. Therefore, "the first argument" in this English sentence is processed as the 0th argument within `LiFtEr`.

> terms `Trm 2`, there exists an occurrence `Trm_Occ 2` of `Trm 2` such that, there exists a number `Numb 1`, such that `Trm_Occ 2` is the `(Numb 1)`th argument of `Trm_Occ 1` and that `Trm 2` is the `(Numb 1)`th induction terms passed to the `induct` method.

Our alternative proof is compatible with this heuristic: there is an argument, `itrev.induct`, in the `rule` field, and the occurrence of its related term, `itrev`, in the proof goal takes all the induction terms, `xs` and `ys`, as its arguments in the same order.

4.6 Example 6. Generalize variables in induction terms.

Isabelle's `induct` method offers the `arbitrary` field, so that users can specify which terms to be generalized in induction steps; however, it is known to be a hard problem to decide which terms to generalize.

Of course `LiFtEr` cannot not provide you with a decision procedure to determine which terms to generalize, but it let you describe heuristics to identify variables that are likely to be generalized by experienced Isabelle users. For example, experienced users know that it is usually a bad idea to pass induction terms themselves to the `arbitrary` field. We also know that it is often a good idea to generalize variables appearing within induction terms if induction terms are compound terms.

We can encode the former heuristic using `Are_Same_Trm`, which checks if two terms are the same term or not. For instance, we can write the following assertion:

```
All_Arb (Trm 1,
 Not (Some_Ind (Trm 2,
  Are_Same_Trm (Trm 1, Trm 2))));
```

By now, it should be easy to see that this assertion checks if the following holds:

> for all terms in the `arbitrary` field, there is no induction term of the same term in the `induct` method.

The latter heuristic involves the description of the term structure constituting the proof goal. For this purpose we use `Is_In_Trm_Loc` to check if a term occurrence resides within another term occurrence. With this construct, we can encode the latter heuristic as following:

```
 Some_Ind (Trm 1,
  Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
   (All_Trm (Trm 2,
     Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       ((Trm_Occ 2 Is_In_Trm_Loc Trm_Occ 1)
        And
         Is_Free (Trm_Occ 2))
```

```
Imply
 Some_Arb (Trm 3,
  Are_Same_Trm (Trm 2, Trm 3)))))));
```

Again, we used `Imply` to avoid applying this generalization heuristics to the cases without compound induction terms.

5 Induction Heuristics Across Problem Domains

In Section 4 we wrote six example assertions in `LiFtEr`. When writing these six assertions, we emphasized that none of them is specific to the data structure `list` or the function `itrev` appearing the proof goal. I this section we demonstrate that the `LiFtEr` assertions written in Section 4 are applicable across domains, taking an inductive problem from a completely different domain as an example. The following code is the formalization of a simple stack machine from Concrete Semantics [17]:

```
type_synonym vname = string
type_synonym val   = int
type_synonym state = "vname => val"
datatype instr     = LOADI val | LOAD vname | ADD
type_synonym stack = "val list"

fun exec1 :: "instr => state => stack => stack" where
  "exec1 (LOADI n) _ stk        =  n       # stk"
| "exec1 (LOAD x)  s stk        =  s(x)    # stk"
| "exec1  ADD      _ (j#i#stk)  =  (i + j) # stk"

fun exec :: "instr list => state => stack => stack" where
  "exec []     _ stk = stk"
| "exec (i#is) s stk = exec is s (exec1 i s stk)"
```

`exec1` defines how the stack machine in a certain state transforms a given stack into a new one by executing one instruction, whereas `exec` specifies how the machine executes a series of instructions one by one. Nipkow *et al.* proved the following lemma using structural induction.

```
lemma exec_append_model_prf[simp]:
  "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"
  apply(induct is1 arbitrary: stk) by auto
```

This lemma states that executing a concatenation of two lists of instructions in a state to a stack produces the same stack as executing the first list of the instructions first in the same state to the same stack and executing the second list again in the same state again but to the resulting new stack. As in the case with the equivalence of two reverse functions, there is also an alternative proof based on recursion induction:

```
lemma exec_append_alt_proof:
  "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"
  apply(induct is1 s stk rule:exec.induct) by auto
```

Now we check if the heuristics from Section 4 correctly recommends these proofs.

Example 1. Both `exec_append_model_prf` and `exec_append_alt_prf` are compatible with this heuristic. For example, `is1` is the only induction term in `exec_append_model_prf`, and it has occurrences in the proof goal, where it occurs as a variable.

Example 2. `exec_append_model_prf` complies with the second example: its only induction term, `is1` occurs at the bottom of the syntax tree as a variable, which is an atomic term. `exec_append_alt_prf` also complies with this heuristic: `is1`, `s`, and `stk` as the arguments of the inner `exec` on the right-hand side of the equation are all atomic terms at the deepest layer of the syntax tree.

Example 3. Both proof scripts comply with this heuristic. For example, the inner occurrence of `exec` on the right-hand side of the equation takes all the induction terms of the alternative proof (namely, `is1`, `s`, and `stk`) as its arguments.

Example 4. This heuristic works for both proof scripts, but it explains the model answer particularly well: it has a recursively defined constant, `exec`, and the inner occurrence of `exec` on the right-hand side of the equation has an occurrence that takes the only induction term `is1` as its first argument, and the first parameter of `exec` always involve a data constructor in the definition of `exec`.

Example 5. This heuristic also works for both proof scripts, but it fits particularly well with the alternative answer: the rule `exec.induct` is derived by Isabelle when defining `exec`, while `exec` has an occurrence as part of the third argument of another `exec` on the right-hand side of the equation, and this inner occurrence of `exec` takes all the induction terms (`is1`, `s`, and `stk`) in the same order.

Example 6. None of our proofs involve induction on a compound term, making Example 6-b rather irrelevant, whereas Example 6-a explains the model answer: the only generalized term, `stk`, does not appear as an induction term.

6 `LiFtEr`'s Preprocessor

The previous examples showed that `LiFtEr` let us encode our induction heuristics following our intuitive understanding of our proof scripts; however, such intuitive understanding is often disparate from the default term representation of Isabelle/HOL. For example, new Isabelle users may expect that the term, `itrev xs ys`, has two arguments, `xs` and `ys`, at the same level even though in reality `xs` and `ys` are *not* located at the same level in the syntax tree in Isabelle's default term representation.

Program 2 The Syntax of LiFtEr.

```
datatype term =
  Const of string        * typ
| Free  of string        * typ
| Var   of (string * int) * typ
| Bound of int
| Abs   of string * typ * term
| $     of term   * term
```

Program 2 shows the default term structure of Isabelle. In this data-type declaration, `typ` represents the type of each term. `Const` represents constants, `Abs` stands for lambda abstraction. Variables bound by a lambda abstraction are bound variables denoted by `Bound`, each of which is identified by an integer representing the corresponding de-Bruijn index. Variables that are not bound by a lambda abstraction are called free variables, represented by `Free`. `Var` denotes schematic variable, which corresponds to logical variable in Prolog, and users can instantiate them during the proof process.

```
  (Const ("Induction_Demo.itrev", _)
 $
   Free ("xs", "'a list"))
$
 Free ("ys", "'a list")
```

`$` represents a function application in Isabelle, and this causes a gap between how a proof goal is represented in Isabelle and how some Isabelle users and the `induct` method see the goal: `$` takes a pair of a function and exactly one argument of the function even when we handle multi-arity functions. In our running example, `itrev xs ys` may appear as one function application of `itrev` to two arguments, `xs` and `ys`; however, this term is represented by two function applications as shown in the above code-snippet. This means that the two arguments of `itrev` belong to distinct depths in Isabelle's internal representation even though for the `induct` method and many human-engineers they should not be discriminated in terms of the depths in the syntax tree when deciding on which variable one should apply induction.

Another problem with regards to the depth of sub-terms occur when a proof goal contains multiple occurrences of the meta-implication or meta-conjunction. For example, if your proof goal take the form of `"P x ==> Q x ==> R x"`, `Q x` appears at a deeper level than `P x` does because the meta implication, `==>`, associates to the right even though such difference in depth does not make a meaningful difference from the view point of the `induct` method because the `induct` method employs its own preprocessing step.

We circumvented this problem by transforming a given proof goal in Isabelle's default term representation into our custom data-type that is closer to both human intuition and the way the `induct` method perceives the proof goal.

First, we replaced the default function application `$` with a new data constructor for the function application with possibly multiple arguments. Second, we replaced both the meta-implication and meta-conjunction with a new multi-arity meta-implication and a new multi-arity meta-conjuction. Lastly, we tagged each node in the new syntax tree with the path from the root to that node, so that the `LiFtEr` interpreter is able to look up appropriate nodes quickly when processing `LiFtEr` quantifiers that have many corresponding sub-terms.

7 Related Work

The recent development in proof automation for higher-order logic takes the meta-tool approach. Gauthier *et al.*, for example, developed an automated tactic prover, TacTicToe, on top of the HOL4 [4]. TacTicToe leans how human engineers used tactics and applies the knowledge to execute a tactic based Monte Carlo tree search. To automate proofs in Coq [19], Komendantskaya *et al.* developed ML4PG [9]. ML4PG uses recurrent clustering to mine a proof database and attempts to find a tactic-based proof for a given proof goal. Both of them try to identify useful lemmas or hypotheses as arguments of a tactic; however, they do not identify promising terms as arguments of a tactic, which is crucial to apply induction effectively.

The most well-known approach for ITP, called waterfall [11], was invented for a first-order logic, which cannot handle induction without jeopardizing the soundness by introducing axioms. Jiang *et al.* followed this approach and ran multiple waterfalls [6] to automate ITP in HOL light [5]. However, when deciding induction variables, they naively picked the first free variable with recursive type and left the selection of appropriate induction variables as future work.

To determine induction variables automatically, Nagashima *et al.* developed a proof strategy language `PSL` and its default proof strategy, `try_hard` for Isabelle/HOL [14]. `PSL` tries to identify useful arguments for the `induct` method by conducting a depth-first search. Sometimes it is not enough to pass arguments to the `induct` method, but users have to specify necessary auxiliary lemmas before applying induction. To automate such labor-intensive work, `PGT` [15], a new extension to `PSL`, produces many lemmas by transforming the given proof goal while trying to identify a useful one in a goal-oriented manner.

The drawback of `PSL` and `PGT` is that they can not produce recommendations if they fail to complete a proof search: when the search space becomes enormous, neither `PSL` and `PGT` gives any advice to Isabelle users.

`PaMpeR` [13], on the other hand, recommends which proof method is likely to be useful to a given proof goal, using a supervised learning applied to the Archive of Formal Proofs [8]. The key of `PaMpeR` was its feature extractor: `PaMpeR` first applies 108 assertions to each invocation of proof methods and converts each pair of a proof goal with its context and the name of proof method applied to that goal into an array of boolean values of length 108 because this simpler format is amenable for machine learning algorithms to analyze. The limitation of `PaMpeR` is, unlike `PSL`, it cannot recommend which arguments in the `induct` method to tackle a given proof goal.

Taking the same approach as PaMpeR, Nagashima attempted to build a recommendation tool, MeLoId [12], to automatically suggest promising arguments for the induct method without completing a proof: they wrote many assertions in Isabelle/ML. Unfortunately, encoding induction heuristics as assertions directly in Isabelle/ML caused an immense amount of code-clutter, and they could not encode even the notion of depth in syntax tree due to the problem discussed in Section 6. Therefore, we developed LiFtEr, expecting that LiFtEr serves as a language for feature extraction.

8 Conclusion and Future Work

ITP has been a considered as a very challenging task. To address this issue, we presented LiFtEr. LiFtEr is a domain-specific language in the sense that we developed LiFtEr to encode induction heuristics; however, heuristics written in LiFtEr are often not specific to any problem domains. To the best of our knowledge, LiFtEr is the first programming language developed to capture induction heuristics across problem domains, and its interpreter is the first system that executes meta-reasoning on interactive inductive theorem proving.

The novelty of LiFtEr and its interpreter make its syntax and behaviour rather esoteric. Therefore, we explained how to write induction heuristics in LiFtEr and how its interpreter behaves for a given heuristic and invocation of the induct method using six small self-contained examples.

We hope that when combined into the supervised learning framework of MeLoId, assertions written in LiFtEr extract the essence of induction in Isabelle/HOL in a cross-domain style and produce a useful database for machine learning algorithms, so that new Isabelle users can have the recommendation of promising arguments for the induct method in a fully automatic way.

References

1. Blanchette, J., Kaliszyk, C., Paulson, L., Urban, J.: Hammering towards qed. Journal of Formalized Reasoning **9**(1), 101–148 (2016). https://doi.org/10.6092/issn.1972-5787/4593, https://jfr.unibo.it/article/view/4593
2. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending sledgehammer with SMT solvers. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6803, pp. 116–130. Springer (2011). https://doi.org/10.1007/978-3-642-22438-6, http://dx.doi.org/10.1007/978-3-642-22438-6
3. Bundy, A.: The automation of proof by mathematical induction. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning (in 2 volumes), pp. 845–911. Elsevier and MIT Press (2001)
4. Gauthier, T., Kaliszyk, C., Urban, J.: Tactictoe: Learning to reason with HOL4 tactics. In: Eiter, T., Sands, D. (eds.) LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana,

May 7-12, 2017. EPiC Series in Computing, vol. 46, pp. 125–143. EasyChair (2017), http://www.easychair.org/publications/paper/340355

5. Harrison, J.: HOL light: A tutorial introduction. In: Srivas, M.K., Camilleri, A.J. (eds.) Formal Methods in Computer-Aided Design, First International Conference, FMCAD '96, Palo Alto, California, USA, November 6-8, 1996, Proceedings. Lecture Notes in Computer Science, vol. 1166, pp. 265–269. Springer (1996). https://doi.org/10.1007/BFb0031814, https://doi.org/10.1007/BFb0031814
6. Jiang, Y., Papapanagiotou, P., Fleuriot, J.D.: Machine learning for inductive theorem proving. In: Fleuriot, J.D., Wang, D., Calmet, J. (eds.) Artificial Intelligence and Symbolic Computation - 13th International Conference, AISC 2018, Suzhou, China, September 16-19, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11110, pp. 87–103. Springer (2018). https://doi.org/10.1007/978-3-319-99957-9_6, https://doi.org/10.1007/978-3-319-99957-9_6
7. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: sel4: formal verification of an operating-system kernel. Commun. ACM **53**(6), 107–115 (2010). https://doi.org/10.1145/1743546.1743574, http://doi.acm.org/10.1145/1743546.1743574
8. Klein, G., Nipkow, T., Paulson, L., Thiemann, R.: (2004), https://www.isa-afp.org/
9. Komendantskaya, E., Heras, J.: Proof mining with dependent types. In: Geuvers, H., England, M., Hasan, O., Rabe, F., Teschke, O. (eds.) Intelligent Computer Mathematics - 10th International Conference, CICM 2017, Edinburgh, UK, July 17-21, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10383, pp. 303–318. Springer (2017). https://doi.org/10.1007/978-3-319-62075-6_21, https://doi.org/10.1007/978-3-319-62075-6_21
10. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009). https://doi.org/10.1145/1538788.1538814, http://doi.acm.org/10.1145/1538788.1538814
11. Moore, J.S.: Computational logic : structure sharing and proof of program properties. Ph.D. thesis, University of Edinburgh, UK (1973), http://hdl.handle.net/1842/2245
12. Nagashima, Y.: Towards machine learning mathematical induction. CoRR **abs/1812.04088** (2018), http://arxiv.org/abs/1812.04088
13. Nagashima, Y., He, Y.: PaMpeR: proof method recommendation system for isabelle/hol. In: Huchard, M., Kästner, C., Fraser, G. (eds.) Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018. pp. 362–372. ACM (2018). https://doi.org/10.1145/3238147.3238210, https://doi.org/10.1145/3238147.3238210
14. Nagashima, Y., Kumar, R.: A proof strategy language and proof script generation for isabelle/hol. In: de Moura, L. (ed.) Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10395, pp. 528–545. Springer (2017). https://doi.org/10.1007/978-3-319-63046-5_32, https://doi.org/10.1007/978-3-319-63046-5_32
15. Nagashima, Y., Parsert, J.: Goal-oriented conjecturing for isabelle/hol. In: Rabe, F., Farmer, W.M., Passmore, G.O., Youssef, A. (eds.) Intelligent Computer Mathematics - 11th International Conference, CICM 2018, Hagenberg, Austria, August 13-17, 2018, Proceedings. Lecture Notes in Computer Science, vol.

11006, pp. 225–231. Springer (2018). https://doi.org/10.1007/978-3-319-96812-4_19, https://doi.org/10.1007/978-3-319-96812-4_19
16. Nagashima, Y., et al.: data61/psl, https://github.com/data61/PSL/releases/tag/v0.1.3-alpha
17. Nipkow, T., Klein, G.: Concrete Semantics - With Isabelle/HOL. Springer (2014). https://doi.org/10.1007/978-3-319-10542-0, https://doi.org/10.1007/978-3-319-10542-0
18. 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)
19. The Coq development team: The Coq proof assistant, https://coq.inria.fr