Implementing a Proof Assistant using Focusing and Logic Programming

Tomer Libal

The American University of Paris
France

tlibal@aup.edu

Interfacing with theorem provers is normally done by specifying a strategy. Such a strategy can be local, such as using tactics in interactive theorem provers, or global when using automatic ones. Focused proof calculi have a clear separation between the phases in the proof search which can be done fully automatically and those phases which require decision making. The calculi allow for an automated switch between the two phases and thus, make them very suitable for interactive theorem proving. The inference rules of these proof calculi can be defined using logical formulas, such as implications. Logic programming languages allow for proof search over a database of such logical formulas and also support interaction with the user via input/output calls.

In this paper I describe a possible process of writing an interactive proof assistant over a focused sequent calculus using the higher-order programming language lambda-prolog. I attempt to show that one can gain a high level of trust in the correctness of the prover, up to the correctness of an extremely small kernel. This process might allow one to obtain a fully functional proof assistant using a small amount of code and by using a clear process for arbitrary focused calculi. The combination of these factors might allow users from different disciplines to develop personalized, trusted and simple proof assistants.

1 Introduction

Proof assistants are sophisticated systems which have helped users to prove a wide range of mathematical theorems [5, 15, 16, 18] and program properties [4, 6, 20]. Nevertheless, these tools normally require knowledge of computational logic, mathematical skills and experience with the chosen tool. In addition, users must have some familiarity with the domain of the chosen tool, such as intuitionistic type theory for Coq [1] and Lean [27], set theory for Mizar [33] or higher-order logic for Isabelle/HOL [29] and HOL Light [2] despite the existence of different shallow embeddings of other domains.

At the same time, there are many users who lack one of the above requirements but would still like to be able to enjoy the support of a computer in order to proof theorems in a wide array of domains, from university students to researchers of law and linguistics. There is an effort to bring proof assistants closer to different audiences, for example the tool CalcCheck [19], attempts to help students use a proof assistant by abstracting over the mathematical language required. This effort focuses on one domain and one audience.

Many users who could enjoy the help of a proof assistant are left, therefore, without such support. When accumulating the factors which prevent the implementation of a personal proof assistant adapted to the needs of each student and researcher, we see that proof assistants are non-trivial software requiring a high level of programming skills. Therefore, the majority of proof assistants are implemented in functional programming languages which facilitate their creation. Still, programmers of any proof assistant must handle a variety of common but non-trivial tasks such as proof search, variables, binders, unification, substitutions and many others.

Submitted to:
UITP 2018

Taking all these points into account, this paper claims that the main reason for the complexity of proof assistants is their attempt to be as general as possible on the expense of simplicity of the calculus. This is understandable given the complexity of creating a proof assistant. The alternative suggested in this paper is to create a specific and non-general proof assistant. Such a proof assistant will be easier to use for the non-expert. In order to tackle the complexity of designing a new proof assistant, the paper suggests using a specific recipe based on the combination of focused sequent calculus and logic programming.

Advocates of higher-order logic programming languages, such as Felty and Miller [12] have argued that these languages are very suited for the creation of proof assistants [11, 24] and other proof theoretical products [23]. Higher-order logic programming languages give a native support for all of the required tasks just mentioned and offer, therefore, not only a much easier coding experience but also an increased level of trust in correctness. The system Twelf [30], which is based on dependent type theory, is enjoying all the above properties. A main difference between Twelf and λProlog is that Twelf is not a full-featured logic programming languages and lacks properties such as input/output, which are a requirement for an interactive proof assistant.

More recently, Sacerdoti Coen, Tassi and their team have developed an efficient interpreter [10] for the higher-order logic programming language λProlog [25] and applied it to the creation of several proof assistants [9, 17, 32]. They showed that using higher-order logic programming greatly reduces the size of the program.

Another complexity arising in the creation of proof assistants is the need to support complex interfaces between users and machines. The calculi at the core of most proof assistants do not support, out of hand, interactive proof search. Focused sequent calculi [3] partially solve this problem by separating proof search into two different modes. One of the modes, which can be executed fully automatically, can be applied eagerly in order to save the user from tasks not requiring her attention. The assistant then (automatically) switches to the second mode when user interaction is required.

In this paper I want to take one step forward and show that when using higher-order logic programming in combination with a focused calculus, the creation of a new proof assistant becomes rather trivial. I exemplify that by the creation of a proof assistant for first-order classical logic which consists of less than 100 lines of code. This proof assistant can be easily extended with new tactics and features.

The use of a focused calculus together with higher-order logic programming very closely relates to the work by Miller and his group towards proof certification [23, 8, 7]. Given that proof certification can be considered as a restriction of interactive proof search where the interaction is done between the proof certificate and the program, this paper attempts to generalize the approach to arbitrary interaction including the interaction between a user and a program. At the same time, using higher-order logic programming towards the creation of proof assistants is one of the purposes of the group behind the ELPI interpreter [9, 17, 32]. Their work is focused on the implementation and extension of fully fledged proof assistants.

My main goal though is the application of this approach to the creation of proof assistants in domains where proof automation is lacking, such as in modal logic and in fields outside of computational logic, such as law and linguistics. I propose that using λProlog and focusing, any user can implement and customize her theorem prover to meet her needs.

Before we go to the next section, I would like to clarify the claims about trust mentioned earlier. Trust in proof assistants is measured according to the complexity of their "kernels" – those parts of the assistants which need to be trusted. In turn, the complexity of kernels is measured by the size and complexity of the code in which they are written. The most popular approach to increase trust in kernels is to program them in a strongly typed functional programming language, such as ML. As discussed above, these kernels are still relatively large (thousands of lines of code) which impairs their trust. Using logic programming allows for much smaller kernels (measured in few hundreds of lines of code). Higher-order logic programming still exhibit strong types and the existence of two compilers, greatly decreases the chance of errors originating in bugs in the implementations of λProlog.

The paper is organized as follows. The next two sections introduce the two technologies I am using, the focused sequent calculus and higher-order logic programming. I then describe the implementation of a proof assistant for first-order classical logic based on these technologies and give examples of ts usage and extensions. I finish with a short conclusion and mention some possible future work.

2 Focused sequent calculus

Theorem provers often employ efficient proof calculi, like, e.g., resolution, possibly with the additional use of heuristics or optimization techniques, whose complexity leads to a lower degree of trust. On the other hand, traditional proof calculi, like the sequent calculus, enjoy a high degree of trust but are quite inefficient for proof search. In order to use the sequent calculus as the basis for proof search, much more structure within proofs needs to be established. Focused sequent calculi, first introduced by Andreoli [3] for linear logic, combine the higher degree of trust of sequent calculi with a more efficient proof search. They take advantage of the fact that some of the rules are "invertible", i.e., can be applied without requiring backtracking, and that some other rules can "focus" on the same formula for a batch of deduction steps. In this paper, I will make use of the classical focused sequent calculus (*LKF*) system defined in [21]. Fig. 1 presents, in black font, the rules of *LKF*.

Formulas in *LKF* are expressed in negation normal form: they can have either positive or negative polarity and are constructed from atomic formulas, whose polarity has to be assigned, and from logical connectives whose polarity is pre-assigned. The choice of polarization does not affect the provability of a formula, but it can have a big impact on proof search and on the structure of proofs: one can observe, e.g., that in *LKF* the rule for $\vee^-$ is invertible while the one for $\vee^+$ is not. The connectives $\wedge^-, \vee^-$ and $\forall$ are of negative polarity, while $\wedge^+, \vee^+$ and $\exists$ are of positive polarity. A compound formula has the same polarity as its main connective. In order to polarize literals, we are allowed to fix the polarity of atomic formulas in any way we see fit. We may ask that all atomic formulas are positive, that they are all negative, or we can mix polarity assignments. In any case, if A is a positive atomic formula, then it is a positive formula and $\neg A$ is a negative formula: conversely, if A is a negative atomic formula, then it is a negative formula and $\neg A$ is a positive formula

Deductions in *LKF* are done during synchronous or asynchronous phases. A synchronous phase, in which sequents have the form $\vdash \Theta \Downarrow B$, corresponds to the application of synchronous rules to a specific positive formula B under focus (and possibly its immediate positive subformulas). An asynchronous phase, in which sequents have the form $\vdash \Theta \Uparrow \Gamma$, consists in the application of invertible rules to negative formulas contained in Γ (and possibly their immediate negative subformulas). Phases can be changed by the application of the *release* rule. Θ denotes the context stored (via the store rule) formulas which are postponed. Since we are always applying negative rules eagerly, the context can contain only positive formulas or negative atoms. The *decide* rule picks a context formula and focuses on it.

In order to simplify the implementation and the representation, I have excluded the `cut` rule from the calculus. given that *LKF* is cut admissible, this simplification does not affect provability but may complicate the process of finding proofs.

2.1 Driving the search in the focused sequent calculus

LKF offers a structure for a proof search – we can eagerly follow paths which apply asynchronous inference rules. Full proof search needs also to deal with the synchronous inference rules, for which there is no effective automation. The ProofCert project [23], which offers solutions to proof certification, suggests augmenting the inference rules with additional predicates. On the one hand, these predicates, will serve as points of communication with the implementation of the calculus (the "kernel" from now on) and will allow for the control and tracking of the search. Each inference call in the kernel will call a control predicate with the aim of both obtaining and supplying information. On the other hand, being added as premises to the inference rules, these predicates do not affect the soundness of the kernel and therefore, do not impair the trust we can place in searching over it. Once a control predicate is being called by the kernel, it is using a user supplied and non-trusted code for recording information from the kernel or supplying the kernel with commands from the user. These predicates communicate with each other using a data structure which is passed around during the proof search. This data structure represents the proof evidence in the proof certifier architecture discussed by Miller in [23].

This approach is very suitable for conducting search using both interactive and automatic theorem provers as well. We can generalize the role of the data structure discussed above to represent information between the user and the kernel. I will therefore generalize the "proof evidence" data structure in the proof certification architecture of Miller to a *"proof control"* data structure. In this paper, this data structure can serve as a proof evidence but it will also serve for getting commands from the user as well as for generating a proof certificate once a proof was found. We can now follow other work on proof certification [7, 8] and enrich each rule of *LKF* with a proof control and additional predicates, given in blue font in Fig. 1. I call the resulting calculus LKF^a. LKF^a extends *LKF* in the following way. Each sequent now contains additional information in the form of the proof control Ξ. At the same time, each rule is associated with a predicate (for example $\mathit{initial}(\Xi, l)$) which, according to the proof control, might prevent the rule from being called or guide it by supplying such information as the witness to be used in the application of the $\exists$ inference rule.

One implementation choice is to use indices in order to refer to formulas in the context. I have included these indices also in the presentation of the calculus in Fig. 1 as can be seen in rules `store` and `decide`.

3 Higher-order logic programming

The other technology we require in order to build simple but trusted proof assistants, is a higher-order logic programming language. λProlog [25] is an extension of Prolog which supports binders [24] and restricted higher-order formulas [26]. Being a logic programming language, it gives us proof search, unification, substitution and other operations which are required in any automated or interactive theorem prover. The extensions allow for the encoding of the meta-theory of predicate calculi, which is impossible in the first-order Prolog language. More concretely, the syntax of λProlog has support for λ-abstractions, written `x\ t` for $\lambda x.t$ and for applications, written `(t x)`. Existential variables can occur both in head or argument positions of terms and are denoted by words starting with a capital letter. The variable `w` occurring in a term `F` can be universally quantified by writing `pi w\ F`. I use the symbols `some, all, !-!` and `&-&` to denote the encoded logical connectives "exists", "for all", a negative disjunction and a negative conjunction. The implementation contains only the negative versions of the disjunction and conjunction rules presented in Sec. 2. As discussed there, this choice does not affect provability. β-normalization and α-equality are built-in. The full syntax of the language can be found in Miller and Nadathur's book

ASYNCHRONOUS INTRODUCTION RULES

$$\frac{\Xi' \vdash \Theta \Uparrow A, \Gamma \quad \Xi'' \vdash \Theta \Uparrow B, \Gamma \quad \text{andNeg}(\Xi, \Xi', \Xi'')}{\Xi \vdash \Theta \Uparrow A \wedge^- B, \Gamma}$$

$$\frac{\Xi' \vdash \Theta \Uparrow A, B, \Gamma \quad \text{orNeg}(\Xi, \Xi')}{\Xi \vdash \Theta \Uparrow A \vee^- B, \Gamma} \qquad \frac{(\Xi' y) \vdash \Theta \Uparrow [y/x]B, \Gamma \quad \text{all}(\Xi, \Xi')}{\Xi \vdash \Theta \Uparrow \forall x.B, \Gamma}\ \dagger$$

SYNCHRONOUS INTRODUTION RULES

$$\frac{\Xi' \vdash \Theta \Downarrow B_1 \quad \Xi'' \vdash \Theta \Downarrow B_2 \quad \text{andPos}(\Xi, \Xi', \Xi'')}{\Xi \vdash \Theta \Downarrow B_1 \wedge^+ B_2}$$

$$\frac{\Xi' \vdash \Theta \Downarrow B_i \qquad \text{orPos}(\Xi, \Xi', i)}{\Xi \vdash \Theta \Downarrow B_1 \vee^+ B_2} \qquad \frac{\Xi' \vdash \Theta \Downarrow [t/x]B \quad \text{some}(\Xi, t, \Xi')}{\Xi \vdash \Theta \Downarrow \exists x.B}$$

IDENTITY RULE

$$\frac{\langle l, \neg P_a \rangle \in \Theta \quad \mathit{initial}(\Xi, l)}{\Xi \vdash \Theta \Downarrow P_a}\ \mathit{init}$$

STRUCTURAL RULES

$$\frac{\Xi' \vdash \Theta \Uparrow N \quad \mathit{release}(\Xi, \Xi')}{\Xi \vdash \Theta \Downarrow N}\ \mathit{release} \qquad \frac{\Xi' \vdash \Theta, \langle l, C \rangle \Uparrow \Gamma \quad \mathit{store}(\Xi, C, l, \Xi')}{\Xi \vdash \Theta \Uparrow C, \Gamma}\ \mathit{store}$$

$$\frac{\Xi' \vdash \Theta \Downarrow P \quad \langle l, P \rangle \in \Theta \quad \mathit{decide}(\Xi, l, \Xi')}{\Xi \vdash \Theta \Uparrow \cdot}\ \mathit{decide}$$

Here, P is a positive formula; N a negative formula; P_a a positive literal; C a positive formula or negative literal; and $\neg B$ is the negation normal form of the negation of B. The proviso marked † requires that y is not free in Ξ, Θ, Γ, B.

Figure 1: The proof system LKF^a, augmented version of *LKF*.

”Programming with Higher-Order Logic” [25].

The implementation of λProlog on which I have tested the prover is ELPI [10] which can be installed following instructions on Github. [1] ELPI offers more than just the implementation of λProlog and includes features such as having input/output modes on predicates and support of constraints [9]. These features are not required in the simple proof assistant I describe and are therefore not used in the implementation. Examples of the way these features are used can be found in the implementations of proof assistants for HOL [9] and Type Theory [17].

4 A proof assistant based on focusing and logic programming

In this section, I will present the architecture and techniques used in order to obtain a minimal, trusted proof assistant for classical first-order logic. I believe that this approach can be applied for creating proof assistants for various other logics, based on the existence of suitable focused calculi.

Some parts of the code are omitted from this paper for brevity. These parts mainly deal with

[1] https://github.com/LPCIC/elpi

bootstrapping the program and with type declarations. The proof assistant implementation can be found on Github [2].

4.1 The kernel

The first immediate advantage of using a higher-order logic programming language is the simple and direct coding of the calculus. Fig. 2, 3 and 4 show the code of the whole implementation. A comparison to Fig. 1 shows that each inference rules directly maps to a λProlog predicate. Each inference rule is implemented using a λProlog implication. The consequent denotes the conclusion of the rule while each antecedent represents one premise of the rule. The consequent and some antecedents represent a focused sequent. These sequents are of the form `check Cert (unfk L)` or `check Cert (foc F)`. The `Cert` variable is used for the transformation of information between the user and the kernel, the symbols `unfk` and `foc` represent an asynchronous and synchronous phases, respectively while L and F denote a set of a single formula.

We can see immediately the way the control predicates work. Before an inference rule is applied, the implementation first consult with the control predicate, which in turn, may change the `Cert` data structure or even falsify the call. I will refer to the implementation of these predicates in the next section. Three predicates of special interest are the `store`, `forall` and `exists`. Each emphasizing the need for a higher-order logic programming language in a different way.

The `store` shows the importance of supporting implications in the bodies of predicates. It allows us to dynamically update the λProlog database with new true predicates. I use this feature in order to denote the context of the sequent, i.e. those formulas on which we may decide on later. One can also deal with this problem in the Prolog programming language. Either by using lists for denoting the context or by using the `assert` and `retract` predicates. Both approaches prevent us from having a direct and concise representation of *LKF*. The first due to the requirement of repeatedly manipulate and check the list (not to mention the overhead for searching in the list) and the second due to the need to apply the system predicates manually in the correct points in the program which can lead to unnecessary complications.

The `forall` predicate has a condition that the variable `y` is a fresh variable. Dealing with fresh variables is a recurring problem in all implementations of theorem provers. Some approaches favor using a specific naming scheme in order to ensure that variables are fresh while others might use an auxiliary set of used variables. Using λProlog we need just to quantify over this variable. λProlog's variable capture avoidance mechanism will ensure that this variable is fresh. Another feature of λProlog which is exhibited by this rule is higher-order application: variables can have functional types and can be applied to other terms. In this case, the variable `B` is applied to a fresh variable. Such application requires higher-order unification in order to succeed, which is known to be undecidable [14]. Miller has shown [22] that such applications require a simpler form of unification, which is not only decidable but exhibits the same properties as the first-order unification used in Prolog.

The most intriguing predicate though, is `exists`. This rule contains an application of two free variables, `B` and `T`. Such an application is beyond the scope of the efficient unification algorithm just mentioned. Despite that, implementations of λProlog apply techniques of postponing these unification problems [28] which seems to be enough in most cases.

[2] https://github.com/proofcert/PPAssistant (branch: uitp)

```
% decide
check Cert (unfK []) :-
  decide Cert Indx Cert',
  inCtxt Indx P,
  isPos P,
  check Cert' (foc P).
% release
check Cert (foc N) :-
  isNeg N,
  release Cert Cert',
  check Cert' (unfK [N]).
% store
check Cert (unfK [C|Rest]) :-
  (isPos C ; isNegAtm C),
  store Cert C Indx Cert',
  inCtxt Indx C => check Cert' (unfK Rest).
% initial
check Cert (foc (p A)) :-
  initial Cert Indx,
  inCtxt Indx (n A).
```

Figure 2: λProlog implementation of the structural rules

```
% orNeg
check Cert (unfK [A !-! B | Rest]) :-
  orNeg Cert (A !-! B)  Cert',
  check Cert' (unfK [A, B| Rest]).
% conjunction
check Cert (unfK [A &-& B | Rest]) :-
  andNeg Cert (A &-& B) CertA CertB,
  check CertA (unfK [A | Rest]),
  check CertB (unfK [B | Rest]).
% forall
check Cert (unfK [all B | Theta]) :-
  all Cert (all B) Cert',
  pi w\ (check (Cert' w) (unfK [B w | Theta] )).
```

Figure 3: λProlog implementation of the asynchronous rules

```
% conjunction
check Cert (foc (A &+& B)) :-
   andPos Cert (A &+& B) Direction CertA CertB,
   ((Direction = left-first,
   check CertA (foc A),
   check CertB (foc B));
   (Direction = right-first,
    check CertB (foc B),check CertA (foc A))).
% disjunction
check Cert (foc (A !+! B)) :-
  orPos Cert (A !+! B) Choice Cert',
  ((Choice = left,  check Cert' (foc A));
   (Choice = right, check Cert' (foc B))).
% exists
check Cert (foc (some B)) :-
  some Cert T Cert',
  check Cert' (foc (B T)).
```

Figure 4: λProlog implementation of the Synchronous rules

4.2 Interacting with the user

The previous section discussed the implementation of the calculus. For some problems, all we need to do is to apply the kernel on a given formula. λProlog will succeed only if a proof can be found and will automatically handle all issues related to search, substitution, unification, normalization, etc. which are normally implemented as part of each theorem prover or proof assistant. This gives us a very simple implementation of an automated theorem prover for classical first-order logic. The downside is, of course, that first-order theorem proving requires coming up with witnesses, making automated theorem proving over the sequent calculus less practical than other methods, such as resolution [31]. The main novelty of this paper is that we can overcome this downside by using other features of λProlog, namely the input and output functionality.

Using the control predicates, we can notify the user of interesting rule applications, such as the addition of fresh variables or the storing of formulas in the context. We can also use them in order to prompt the user for input about how to proceed in case we need to decide on a formula from the context or pick up a witness. Fig. 5 shows the implementation of the control predicates which support these basic operations. The predicates are divided into two groups. Those which can be applied fully automatically, which include most predicates, and those which are applied interactively, which include the `decide` and `exists` predicates. I have simplified the implementation to include only negative conjunctions and disjunctions. The addition of the positive versions does not fundamentally change the approach presented here. In case we would support these two predicates, we will have to treat them in the interactive group.

The interface for a user interaction with the program is to iteratively add guidance information to the proof control. At the beginning, the control contains no user information and the program stops the moment such information is required. In addition, the program displays to the user information about the current proof state such as about fresh variables which were used or new formulas which were added to the context, together with their indices. When the program stops due to required user information, it prompts a message to the user and asks her to supply this information as can be seen in the implementation of the predicates `decide` (lines 1–3) and `some` (lines 24–26) in Fig 5.

The proof control I use contains 4 elements.

- the proof evidence - this is used in order to display at the end to the user the generated proof
- the list of user commands - this list, initially empty, contains the commands from the user
- the index of the current formula - this index is used to label formulas with indices in a consistent way
- The list of fresh variables generated so far - this list is used in order to allow the user to supply term witnesses which contain fresh variables. The user, of course, has no access to the fresh variables and we use a mechanism discussed below in order to allow her to supply these terms

Each of the interactive predicates contains two versions, one for prompting the user for input and the other for applying the user input. The first is applied when the user command list (the second argument in the controls object) is empty. The input in the case of the `decide` predicate is an index of a formula in context (which should be chosen from the ones displayed earlier by the `store` predicate). In the case of the `some` predicate, the input is the term witness.

In some cases, the implementation of the `some` inference rules needs to substitute fresh variables inside the term supplied by the user. I use λProlog abstraction and β-normalization directly. The user keeps track of the number n and order of fresh variables introduced so far and the term witness is then of the form $x_1\backslash \ldots x_n\backslash t$ where t may contain any of the bound variables. The `apply_vars` predicate is responsible for applying to the terms the fresh variables in the correct order.

```
decide (interact (unary (decideI no_index) leaf) [] _ _) _ _ :- !,
  output std_out "You␣have␣to␣choose␣an␣index␣to␣decide␣on␣from␣the␣context",
  output std_out "\n", fail.
decide (interact (unary (decideI I) L) [I|Com] FI E) I (interact L Com (u FI) E).
store (interact (unary (storeI I) L) Com I E) F I (interact L Com (u I) E) :-
  output std_out "Adding␣to␣context␣formula␣",
  term_to_string F S1,
  output std_out S1,
  output std_out "\n".
release (interact (unary releaseI L) Com FI E) (interact L Com (u FI) E).
initial (interact (axiom (initialI I)) [] _ _) I.
orNeg (interact (unary (orNegI FI) L) Com FI E) F (interact L Com (u FI) E).
andNeg (interact (binary (andNegI FI) L1 L2) [branch LC RC] FI E) F
  (interact L1 LC (l FI) E) (interact L2 RC (r FI) E).
all (interact (unary (allI FI) L) Com FI E) F
    (Eigen\ (interact L) Com (u FI) [eigen FI Eigen| E]) :-
  output std_out "Using␣eigen␣variable␣",
  term_to_string FI S1,
  output std_out S1,
  output std_out "\n".
some (interact (unary (someI no_index) leaf) [] _ _) _ _ :- !,
  output std_out "You␣have␣to␣choose␣the␣term␣to␣use␣for␣instantiation",
  output std_out "\n", fail.
some (interact (unary (someI FI) L) [T|Com] FI E) T' (interact L Com (u FI) E) :-
  apply_vars T E T'.
apply_vars T [] T.
apply_vars T [eigen _ X|L] T' :-
  apply_vars (T X) L T'.
```

Figure 5: λProlog implementation of basic interaction with the user

The indexing mechanism I use is based on trees and assigns each unitary child of a parent `I` the index `(u I)` while binary children are assigned the indices `(l I)` and `(r I)` respectively. The index of the theorem is `e`. For example, if our theorem is `A &-& B`, meaning a negative conjunction, then this formula is assigned the index `e` while `A` is assigned (if stored in the context) the index `(l e)` and `B` is assigned the index `(r e)`.

I note here that the presented implementation is a very primitive one with the most basic user feedback. In particular, when conjunctions and branching are involved, it becomes very difficult to follow the different branches and their respective contexts and fresh variables. I do supply a mechanism for handling conjunctions (via the `branch` command), but a more user friendly implementation would need to display this information in a better way, for example, by the use of graphical trees.

4.3 An example

In this section I demonstrate the execution of the prover on the Drinker's paradox. In order to use the assistant, the user needs to call the prover with the theorem and an empty command list.

```
$>./run.sh 'some x\ (n (drink x)) !-! (all y\ (p (drink y)))' '[]'
```

Since we start in a negative phase and the theorem is positive, the theorem is stored with index `e` and we are prompted to pick up an index of a formula to decide on. We choose `e`. We are now asked to pick up a term for instantiation. A proof of this theorem requires a user to pick an arbitrary witness at this phase and we indeed pick up the witness `a`. We are again asked for an index to decide on but this time we see that the prover has already executed several steps, including the negative (and automatic) disjunction and `all` steps and that in addition to several new formulas in context, we also have a new fresh variable.

```
decide_ke (interact (unary (decideI I) L) [auto|Com] FI E) I
  (interact L Com (u FI) E) :- !.
```

Figure 6: λProlog implementation of the auto tactic

We decide again on the theorem with index `e` in order to instantiate it with the correct term now. We follow this with supplying the term `(x\ x)`. This term is just the fresh variable introduced in the earlier universal step.

Running the program again, we are presented with a selection of context formulas to decide on. We can see now that the context contains the positive and negative versions of the atom `drink x0` and we choose the index of the positive version, `u (u (u (u (u (u (u e)))))).`

The prover finishes and displays the whole proof which was generated. The last execution is therefore,

```
$>./run.sh 'some x\ (n (drink x)) !-! (all y\ (p (drink y)))'
  '[e,a,e,(x\x), u (u (u (u (u (u (u e)))))))]'
```

4.4 Creating and using tactics

Supporting interactive proof search still falls short from the needs of most users. Optimally, a proof assistant would require the help of the user only for the most complex problems and will be able to deal with simpler ones by itself. In the previous example, we had to search for the index to decide on. But, there are finitely many options only. Can't we let the prover try all options by itself?

In order to support a tactics language, I extend the program with an additional tactics file. This file will contain additional implementations for the control predicates. The λProlog interpreter will choose the right implementation according to whether the predicate is called with a tactic command or with a command to decide on an index of a formula or a witness term. Fig. 6 presents the additional predicate we need to add in order to support an `auto` tactic which attempts to decide on all possible context formulas.

Using this tactic, the commands required in order to prove the theorem from the example is

```
$>./run.sh 'some x\ (n (drink x)) !-! (all y\ (p (drink y)))' '[auto,a,auto,(x\x),auto]'
```

It should be noted though, that the simple `auto` command introduced in this section might fail to work in cases we encounter an infinite loop on a wrong guess. A more advanced mechanism with a depth bound should be used in this case.

5 Conclusion and further work

The aim of this paper was to describe the process of implementing a minimal proof assistant based on focusing and λProlog. The steps described are the following: given a focused sequent calculus, implement in λProlog its inference rules, a set of control predicates in order to obtain interaction and possibly, a set of tactics. The program can now be executed by running ELPI with a theorem and the required user commands. The implementation described is intended as a "show case" only. I present it in order to show an alternative approach for interfacing with a proof assistant. The interface which is implemented in this paper is too primitive to be useful in most cases.

The main target audience of this approach are users who find the existing professional proof assistants too strong and complex for their needs or users which work with proof calculi which are not easily embedded into any of the existing proof assistants. The next step is to try and apply this approach to formal domains where automated and interactive tools are scarce, such as in modal logic.

There are several other possible extensions to this work. An important extension is the creation of a generic graphical user interface, which can parse and display λProlog proofs and proof information. Another is the creation of a library of basic tactics which can be applied to a variety of logics.

References

[1] *The Coq Proof Assistant*. https://coq.inria.fr/.

[2] *John Harrison. The HOL light theorem prover*. `https://github.com/jrh13/hol-light/`.

[3] Jean-Marc Andreoli (1992): *Logic Programming with Focusing Proofs in Linear Logic*. J. Log. Comput. 2(3), pp. 297–347, doi:10.1093/logcom/2.3.297. Available at `http://dx.doi.org/10.1093/logcom/2.3.297`.

[4] Gilles Barthe, Benjamin Grégoire & Santiago Zanella Béguelin (2009): *Formal certification of code-based cryptographic proofs*. ACM SIGPLAN Notices 44(1), pp. 90–101.

[5] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond & Pierre Weis (2010): *Formal proof of a wave equation resolution scheme: the method error*. In: International Conference on Interactive Theorem Proving, Springer, pp. 147–162.

[6] Boutheina Chetali & Quang-Huy Nguyen (2008): *About the world-first smart card certificate with EAL7 formal assurances*. Slides 9th ICCC, Jeju, Korea (September 2008), www. commoncriteriaportal. org/iccc/9iccc/pdf B 2404.

[7] Zakaria Chihani, Tomer Libal & Giselle Reis (2015): *The proof certifier checkers*. In: International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, Springer, pp. 201–210.

[8] Zakaria Chihani, Dale Miller & Fabien Renaud (2017): *A semantic framework for proof evidence*. Journal of Automated Reasoning 59(3), pp. 287–330.

[9] Cvetan Dunchev, Claudio Sacerdoti Coen & Enrico Tassi (2016): *Implementing HOL in an higher order logic programming language*. In: Proceedings of the Eleventh Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, ACM, p. 4.

[10] Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen & Enrico Tassi (2015): *ELPI: Fast, Embeddable, λProlog Interpreter*. In: Logic for Programming, Artificial Intelligence, and Reasoning, Springer, pp. 460–468.

[11] Amy Felty (1993): *Implementing tactics and tacticals in a higher-order logic programming language*. Journal of Automated Reasoning 11(1), pp. 43–81.

[12] Amy Felty & Dale Miller (1988): *Specifying theorem provers in a higher-order logic programming language*. In: International Conference on Automated Deduction, Springer, pp. 61–80.

[13] Melvin Fitting (2012): *First-order logic and automated theorem proving*. Springer Science & Business Media.

[14] Warren D Goldfarb (1981): *The undecidability of the second-order unification problem*. Theoretical Computer Science 13(2), pp. 225–230.

[15] Georges Gonthier (2008): *Formal proof–the four-color theorem*. Notices of the AMS 55(11), pp. 1382–1393.

[16] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell OConnor, Sidi Ould Biha et al. (2013): *A machine-checked proof of the odd order theorem*. In: International Conference on Interactive Theorem Proving, Springer, pp. 163–179.

[17] Ferruccio Guidi, Claudio Sacerdoti Coen & Enrico Tassi (2017): *Implementing Type Theory in Higher Order Constraint Logic Programming*.

[18] Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Hoang Le Truong, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen et al. (2017): *A formal proof of the Kepler conjecture*. In: Forum of Mathematics, Pi, 5, Cambridge University Press.

[19] Wolfram Kahl (2011): *The teaching tool CalcCheck a proof-checker for Gries and Schneiders logical approach to discrete math*. In: International Conference on Certified Programs and Proofs, Springer, pp. 216–230.

[20] Xavier Leroy (2009): *Formal verification of a realistic compiler*. Communications of the ACM 52(7), pp. 107–115.

[21] Chuck Liang & Dale Miller (2009): *Focusing and polarization in linear, intuitionistic, and classical logics*. Theor. Comput. Sci. 410(46), pp. 4747–4768, doi:10.1016/j.tcs.2009.07.041. Available at `http://dx.doi.org/10.1016/j.tcs.2009.07.041`.

[22] Dale Miller (1991): *A logic programming language with lambda-abstraction, function variables, and simple unification*. Journal of logic and computation 1(4), pp. 497–536.

[23] Dale Miller (2011): *A proposal for broad spectrum proof certificates*. In: International Conference on Certified Programs and Proofs, Springer, pp. 54–69.

[24] Dale Miller (2017): *Mechanized Metatheory Revisited: An Extended Abstract* . In: Post-proceedings of TYPES 2016 , Novi Sad, Serbia, doi:10.4230/LIPIcs. Available at `https://hal.inria.fr/hal-01615681`.

[25] Dale Miller & Gopalan Nadathur (2012): *Programming with higher-order logic*. Cambridge University Press.

[26] Dale Miller, Gopalan Nadathur, Frank Pfenning & Andre Scedrov (1991): *Uniform proofs as a foundation for logic programming*. Annals of Pure and Applied logic 51(1-2), pp. 125–157.

[27] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn & Jakob von Raumer (2015): *The Lean theorem prover*.

[28] Gopalan Nadathur (2005): *A treatment of higher-order features in logic programming*. Theory and Practice of Logic Programming 5(3), pp. 305–354.

[29] Tobias Nipkow, Lawrence C Paulson & Markus Wenzel (2002): *Isabelle/HOL: a proof assistant for higher-order logic*. 2283, Springer Science & Business Media.

[30] Frank Pfenning & Carsten Schürmann (1999): *System description: Twelfa meta-logical framework for deductive systems*. In: International Conference on Automated Deduction, Springer, pp. 202–206.

[31] John Alan Robinson (1965): *A machine-oriented logic based on the resolution principle*. Journal of the ACM (JACM) 12(1), pp. 23–41.

[32] Enrico Tassi (2017): *Elpi: an extension language for Coq Metaprogramming Coq in the Elpi λProlog dialect*.

[33] Andrzej Trybulec & Howard A Blair (1985): *Computer assisted reasoning with MIZAR.* In: IJCAI, 85, pp. 26–28.