A Simple Semi-automated Proof Assistant for First-order Modal Logics

Tomer Libal

The American University of Paris, France
tlibal@aup.edu

Abstract

Most theorem provers and proof assistants are written in imperative or functional programming languages. Recently, the claim that higher-order logic programming languages might be better suited for this task was revisited and a new interpreter, as well as new proof assistants based on it, were introduced. In this paper I follow these results and describe a concise implementation of a prototype for a semi-automated proof assistant for first-order modal logics. The aim of this paper is to encourage the development of personal proof assistants and semi-automated provers for a variety of modal logics.

1 Introduction

Proof assistants are sophisticated systems which have helped users to prove a wide range of mathematical theorems [8, 20, 21, 23] and program properties [3, 11, 25]. Nevertheless, these tools normally require knowledge of computational logic, mathematical skills and experience with the chosen tool. In addition, these tools are based on specific theories, such as intuitionistic type theory for Coq [1] or higher-order logic for Isabelle/HOL [35] and HOL Light [24], which might not be easily applicable to other domains, such as to first-order modal logics.

Benzmüller and Wolzenlogel Paleo have shown that by embedding higher-order modal logics in Coq [4], one can interactively search for proofs. A general description of their work with respect also to other proof assistants is described in [6]. Such an approach takes advantage of the full power of a leading proof assistant and is also clearly general and applicable to other domains. Possible downsides are the Coq expertise required, the required knowledge in intuistionistic type theory for extensions as well as the fact that despite being shallow, an embedding is still an indirect way of communicating with the target calculus - modal logic in our case.

Except the above mentioned works, very little progress has been made towards using proof assistants for modal logics. One reason for that is 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, unification, substitutions and many others. Therefore, it is not surprising that one of the leading theorem provers for first-order modal logics is MleanCoP, which is written in Prolog [36]. Prolog gives programmers proof search and other operations for free and allow for a more concise and trusted code. Still, the fact that Prolog is based on first-order logic necessarily means that it is not suitable for a shallow embedding of systems whose meta-theory requires higher-order logic. Among such systems are first-order classical and modal logics. Such embeddings would require a programming language which supports higher-order features, such as binding and higher-order unification.

Advocates of higher-order logic programming languages, such as Felty and Miller [18] have argued that these languages are very suited for the creation of proof assistants [17, 30] and proof checkers [29]. Higher-order logic programming languages provide 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 its correctness. More recently, Sacerdoti-Coen, Tassi, Dunchev and Ferruccio have developed an efficient interpreter [16] for the higher-order logic programming language λProlog [31] and used it for the creation of several proof assistants [15, 22, 38]. They showed that using higher-order logic programming greatly reduces the size of the program. While there are many similarities between their work and the current paper, I am interested in utilizing logic programming for the creation of many, simple and personalized proof assistants and not for the implementation of full scale, general and complex ones.

Another complexity arising in the creation of proof assistants is the need to interface between the users and the tool. The calculi at the core of most proof assistants do not support, out of the box, interactive proof search. Focused sequent calculi [2] 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 switches to the second mode when user interaction is required.

The above discussion identifies two issues. First, when one wants to use a proof assistant for modal logics, she needs to have both a good proficiency with the current ones and the ability to embed her logic in their theories. Second, if she chooses to implement her own, the task is far from being simple. In this paper I want to present a third alternative – implement your own proof assistant by following a precise recipe. As we will see, the advantages of this approach are the simplicity of the process – I exemplify that by the implementation of a proof assistant for first-order modal logic which consists of less than 200 lines of code. There are also several disadvantages, the most important of which is the need to have a proof calculus based on focused sequents. Another disadvantage is the required λProlog skill. It should be noted though, that the vast majority of the code requires only proficiency in Prolog.

The technique demonstrated in this paper, of using a focused calculus together with higher-order logic programming, is based on the work by Miller and his group towards proof certification [29, 13, 12]. 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 [15, 22, 38]. Their work is focused on the implementation and extension of fully fledged proof assistants.

My main aim 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 such as law [37]. I propose that by using λProlog and focusing, any user can implement and customize her theorem prover to meet her needs. The proof assistant for first-order modal logics described in this paper is based on the existence of a focused sequent calculus for this logic. I have therefore obtained such a calculus by the combination of two existing ones. The next section focuses on its presentation. The following section then introduces the other technology I use – higher-order logic programming. I then describe the implementation of a proof assistant for first-order modal logic based on these technologies and give examples of usage and extension. I finish with a short conclusion and mention some possible future work.

2 Focused sequent calculus for first-order modal logics

Theorem provers are often implemented using efficient proof calculi, like, e.g., resolution, combined with the additional use of heuristics and optimization techniques. The use of these techniques together with required operations such as unification and search leads to a lower degree of trust. On the other hand, traditional proof calculi, like the sequent calculus, rely on less meta-theory and enjoy a higher degree of trust but are quite inefficient for proof search. In order to use the sequent calculus as the basis of automated deduction, much more structure within proofs needs to be established. Focused sequent calculi, first introduced by Andreoli [2] 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 combine two different focused sequent calculi in order to obtain a sound and complete system for first-order modal logic for K with constant domains and rigid designation. This means that each quantified variable denotes the same element in all worlds and in addition, that the domain for quantification in each world is the same. Please refer to [9] for more information. The existence of focused systems for some other modal logics [27] suggests that the approach described in this paper can be extended. The syntax for first-order modal formulas contains atomic predicates $P(t_1, \ldots, t_n)$, the usual first-order connectives and quantifiers as well as the modal operators $\Box$ and $\Diamond$. The first system I will use is the focused first-order sequent calculus (*LKF*) system defined in [26]. I will combine it with the focused sequent calculus for propositional modal logic for K defined in [33]. This calculus is based on labeled sequents.

The basic idea behind labeled proof systems for modal logic is to internalize elements of the corresponding Kripke semantics into the syntax. The *LMF* system defined in [33] is a sound and complete system for a variety of propositional modal logics. Fig. 1 presents the combined system, LMF^1.

Sequents in LMF^1 have the form $\mathcal{G} \vdash \Theta \Downarrow x : B$ or $\mathcal{G} \vdash \Theta \Uparrow \Gamma$, where the relational set (of the sequent) $\mathcal{G}$ is a set of relational atoms, $x : B$ is a labeled formula (see below) and Θ and Γ are multisets of labeled formulas.

Formulas in LMF^1 which are expressed in negation normal form, 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 $\wedge^-, \vee^-$, $\Box$ and $\forall$ are of negative polarity, while $\wedge^+, \vee^+$, $\Diamond$ and $\exists$ are of positive polarity. A composed formula has the same polarity of 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. The basic entities of the calculus are labelled formulas – $x : F$ – which attach to each formula F a label x which denotes the world F is true in.

Deductions in *LKF* are constructed by synchronous and asynchronous phases. A synchronous phase, in which sequents have the form $\mathcal{G} \vdash \Theta \Downarrow x : B$, 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 $\mathcal{G} \vdash \Theta \Uparrow \Gamma$, 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. In order to simplify the implementation and the representation, I have excluded the `cut` rule from the calculus. The admissibility of this rule in *LKF* means that while proofs might be

Asynchronous introduction rules

$$\frac{\mathcal{G} \vdash \Theta \Uparrow x : A, \Gamma \quad \mathcal{G} \vdash \Theta \Uparrow x : B, \Gamma}{\mathcal{G} \vdash \Theta \Uparrow x : A \wedge^- B, \Gamma} \wedge^-_K \qquad \frac{\mathcal{G} \vdash \Theta \Uparrow x \colon A, x \colon B, \Gamma}{\mathcal{G} \vdash \Theta \Uparrow x \colon A \vee^- B, \Gamma} \vee^-_K$$

$$\frac{\mathcal{G} \cup \{xRy\} \vdash \Theta \Uparrow y \colon B, \Gamma}{\mathcal{G} \vdash \Theta \Uparrow x : \Box B, \Gamma} \Box_K \qquad \frac{\mathcal{G} \vdash \Theta \Uparrow x : [y/z]B, \Gamma}{\mathcal{G} \vdash \Theta \Uparrow x : \forall z.B, \Gamma} \forall_K$$

Synchronous introduction rules

$$\frac{\mathcal{G} \vdash \Theta \Downarrow x : A \quad \mathcal{G} \vdash \Theta \Downarrow x : B}{\mathcal{G} \vdash \Theta \Downarrow x : A \wedge^+ B} \wedge^+_K \qquad \frac{\mathcal{G} \vdash \Theta \Downarrow x : [t/z]B}{\mathcal{G} \vdash \Theta \Downarrow x : \exists z.B} \exists_K$$

$$\frac{\mathcal{G} \vdash \Theta \Downarrow x : B_i}{\mathcal{G} \vdash \Theta \Downarrow x : B_1 \vee^+ B_2} \vee^+, i \in \{1, 2\} \qquad \frac{\mathcal{G} \cup \{xRy\} \vdash \Theta \Downarrow y : B}{\mathcal{G} \cup \{xRy\} \vdash \Theta \Downarrow x : \Diamond B} \Diamond_K$$

Identity rule

$$\frac{}{\mathcal{G} \vdash x : \neg B, \Theta \Downarrow x : B} \; init_K$$

Structural rules

$$\frac{\mathcal{G} \vdash \Theta, x : B \Uparrow \Gamma}{\mathcal{G} \vdash \Theta \Uparrow x : B, \Gamma} \; store_K \qquad \frac{\mathcal{G} \vdash \Theta \Uparrow x : B}{\mathcal{G} \vdash \Theta \Downarrow x : B} \; release_K \qquad \frac{\mathcal{G} \vdash x : B, \Theta \Downarrow x : B}{\mathcal{G} \vdash x : B, \Theta \Uparrow \cdot} \; decide_K$$

In $decide_K$, B is positive; in $release_K$, B is negative; in $store_K$, B is a positive formula or a negative literal; in $init_K$, B is a positive literal. In $\Box_K$ and $\forall_K$, y is different from x and z and does not occur in Θ, Γ, $\mathcal{G}$.

Figure 1: LMF^1: a focused labeled proof system for the first-order modal logic K.

harder to find, completeness is not impaired.

In order to prove the soundness and completeness of LMF^1, we need to define a translation of first-order modal formulas into first-order logic and prove that the translated formula is provable in *LKF* iff the original formula is provable in LMF^1. The translation $ST_x()$ is similar to the one in [9] and extends the standard translation (see, e.g., [7]) with a treatment for quantifiers. Treatment of polarities is omitted from the definition below since it does not affect provability. This translation provides a bridge between first-order modal logic and first-order classical logic:

$$\begin{array}{rcl@{\qquad}rcl}
ST_x(P(y_1,\ldots,y_n)) & = & P(x,y_1,\ldots,y_n) & ST_x(A \wedge B) & = & ST_x(A) \wedge ST_x(B) \\
ST_x(\neg A) & = & \neg ST_x(A) & ST_x(\Box A) & = & \forall y(R(x,y) \supset ST_y(A)) \\
ST_x(A \vee B) & = & ST_x(A) \vee ST_x(B) & ST_x(\Diamond A) & = & \exists y(R(x,y) \wedge ST_y(A)) \\
ST_x(\forall y P(y)) & = & \forall y ST_x(P(y)) & ST_x(\exists y P(y)) & = & \exists y ST_x(P(y))
\end{array}$$

where x is a free variable denoting the world in which the formula is being evaluated. The first-order language into which modal formulas are translated is usually referred to as *first-order correspondence language* [7]. It consists of a binary predicate symbol R and an $(n+1)$ary predicate symbol P for each $n$ary predicate P in the modal language. When a modal operator is translated, a new fresh variable is introduced. It is easy to show that for any modal formula A, any model $\mathcal{M}$ and any world w, we have that $\mathcal{M}, w \models A$ if and only if $\mathcal{M} \models ST_x(A)[x \leftarrow w]$.

Using the translation, we can state the soundness and completeness proposition. It is intuitively correct given the results for similar calculi but a proof will clearly be added in the future.

$$\frac{\Xi', \mathcal{G} \cup \{xRy\} \vdash \Theta \Downarrow y : B \quad \Diamond(\Xi, \Diamond B, y, \Xi')}{\Xi, \mathcal{G} \cup \{xRy\} \vdash \Theta \Downarrow x : \Diamond B} \Diamond_K$$

Figure 2: Augmenting the $\Diamond_K$ inference rule

Proposition 2.1. *Given a first-order modal formula F, $\vdash \Downarrow x : F$ is provable in LMF^1 for an arbitrary world variable x iff $\vdash \Downarrow ST_x(F)$ is provable in LKF.*

2.1 Driving the search in the focused sequent calculus

The system LMF^1 offers a structure for 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 [29], which offers solutions to proof certification, suggests augmenting the inference rules with additional predicates. These predicates, on the one hand, 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. 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. They can, nevertheless, harm completeness. Consider for example an implementation of the $\exists$ control predicate which always returns the same witness. Clearly, the program will fail to find a proof if it requires any other witness. In this example, a correct implementation will prompt the user for the witness or postpone supplying it (more about that in Sec. 4.4). The control predicates communicate with the user or prover using a data structure which is being transferred and manipulated by the predicates. This data structure represents the proof evidence in the proof certifier architecture discussed by Miller in [29].

This approach is very suitable for conducting search using interactive or automated 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. I can now follow other works on proof certification [12, 13] and enrich each rule of LMF^1 with proof controls and additional predicates. Figure 2 gives an example of adding the control and additional predicate (in blue) to the $\Diamond_K$ inference rule. Figure 3 lists all the predicates separately from the calculus (due to lack of space). 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, such as $\Diamond(\Xi, F, w, \Xi')$. This predicate 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 $\exists_K$ or $\Diamond_K$ inference rules. The arguments in the example are the input proof control Ξ, the formula F, the world w to which we should move next and a proof control Ξ' which is given to the upper sequent. I call the resulting calculus LMF^a.

One implementation choice is to use indices in order to refer to formulas in the context. In order to achieve that, the implementations of $store_K$ and $decide_K$ rules contain additional information which is omitted from the definition of the LMF^1 calculus given in Fig. 1.

Asynchronous control predicates

$$\wedge^-(\Xi, F, \Xi', \Xi'') \qquad \vee^-(\Xi, F, \Xi') \qquad \forall(\Xi, F, \Xi' y) \qquad \Box(\Xi, F, \Xi' w)$$

Synchronous control predicates

$$\wedge^+(\Xi, F, \Xi', \Xi'') \qquad \vee^+(\Xi, F, \Xi', i) \qquad \exists(\Xi, F, t, \Xi') \qquad \Diamond(\Xi, F, w, \Xi')$$

Identity and structural control predicates

$$init(\Xi, l) \qquad release(\Xi, \Xi') \qquad store(\Xi, C, l, \Xi') \qquad decide(\Xi, l, \Xi')$$

Figure 3: The additional predicates added to the inference rules of LMF^1 in order to obtain LMF^a

3 Higher-order logic programming

The other technology I use in this paper in an attempt to build a simple but trusted proof assistant, is a higher-order logic programming language. λProlog [31] is an extension of Prolog which supports binders [30] and restricted higher-order formulas [32]. 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 anywhere inside a term 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, box, dia, !-!` and `&-&` to denote the encoded logical connectives "exists", "for all", the modalities "box" and "diamond", 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 "Programming with Higher-Order Logic" [31].

The implementation of λProlog on which I have tested the prover is ELPI [16] 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 [15]. 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 [15] and type theory [22].

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 first-order modal logic. I believe that this approach can be applied for creating proof assistants for various other logics, based on the existence of suitable focused

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

calculi. Some parts of the code are omitted from this paper for brevity. These parts mainly deal with bootstrapping the program and are written using shell scripts. The proof assistant implementation can be found on Github[2] and Zenodo[3].

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. 4, 5 and 6 show the code of the whole implementation. A comparison to Fig. 1 shows that each inference rule directly maps to a λProlog clause. The conclusion of each rule is denoted by the head of the formula while each premise is denoted by a single conjunct in the body. The components of each head are the `Cert` variable, which is used for the transformation of information between the user and the kernel as well the formula (or formulas, in the case of a negative phase) to prove. The two phases are denoted by the function symbols `unfk` and `foc`.

We can see immediately the way the control predicates work. Before we can apply a rule, we need first to 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, but we can already demonstrate how they work. Consider, for example, the `diamond` rule (Fig. 2 and Fig. 6). When λProlog tries to satisfy this clause, it attempts to satisfy each of the antecedents. The first of which is a call to the implementation of the `dia_ke` control predicate. The implementation is discussed in the next section but one can see that in case the implementation of this clause fails, λProlog will fail to apply the `diamond` rule and it will backtrack. One can also see that the implementation can substitute for the variable `T` a term. This term will then be used by the rule as the new label for the formula. This simple mechanism allows us to both control the proof search and to supply additional information (based on user input, for example).

The way we store polarized formulas in the implementation of the labeled sequent calculus is by using a term of the form `lform w f` where `w` is the label (world) and `f` is the formula. Atoms are polarized using the constructor `p` for positive atoms and `n` for negative ones. The example in Sec. 4.3 demonstrates the use of these constructors. Five predicates of special interest are the `store`, `forall`, `exists`, `box` and `diamond`. Each emphasizes 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. We use this feature in order to both denote the context of the sequent, i.e those formulas on which we may decide on later, and the relational set. 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 LMF^1. The first due to the requirement to repeatedly manipulate and check the list (not to mention the overhead for searching in the list). The second due to the need to apply the system predicates manually in the correct points in the program. For example, one should manually retract an asserted predicate once we leave the scope of the implication. These manual manipulations 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

[2] https://github.com/proofcert/PPAssistant
[3] https://zenodo.org/record/1252457

```
% 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 (lform L (p A))) :-
  initial_ke Cert Indx,
  inCtxt Indx (lform L (n A)).
```

Figure 4: λProlog implementation of the structural rules

might use an auxiliary set of used variables. Using λProlog we need just to quantify over this variable. λProlog 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. The quantified formula variable `B` is applied to the fresh variable. In general, the application of a variable to a term requires higher-order unification in the proof search, which is known to be undecidable [19]. Miller has shown [28] that the application of a variable to a bound variable require a simpler form of unification, which is not only decidable but exhibits the same properties as the first-order unification used in Prolog.

A more intriguing predicate though, is `exists`. Here we see 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 sound techniques of postponing these unification problems [34] which seem to suffice in most cases.

Regarding the modalities, we see a close similarity between `box` and `forall`. The only difference being the addition of the new accessible world to the λProlog database, in a similar way to `store`. The `diamond` rule, which is very similar to the `exists` one, then also requires the existence of the specific relation in the λProlog database in order to proceed.

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 first-order modal logic. The downside is, of course, that first-order modal theorem proving is undecidable and requires coming up with witnesses for worlds and terms, making automated theorem proving over the sequent calculus less practical than other methods, such as resolution [14] and free-variable tableaux [10]. 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, new worlds or the storing of formulas in the context. We can also

```
% orNeg
check Cert (unfK [lform L (A !-! B) | Rest]) :-
  orNeg_kc Cert (lform L (A !-! B))  Cert',
  check Cert' (unfK [lform L A, lform L B| Rest]).
% conjunction
check Cert (unfK [lform L (A &-& B) | Rest]) :-
  andNeg_kc Cert (lform L (A &-& B)) CertA CertB,
  check CertA (unfK [lform L A | Rest]),
  check CertB (unfK [lform L B | Rest]).
% box modality
check Cert (unfK [lform L (box B) | Theta]) :-
  box_kc Cert (lform L (box B)) Cert',
  pi w\ rel L w => check (Cert' w) (unfK [lform w B | Theta] ).
% forall quantifier
check Cert (unfK [lform L (all B) | Theta]) :-
  all_kc Cert (all B) Cert',
  pi w\ (check (Cert' w) (unfK [lform L (B w) | Theta] )).
```

Figure 5: λProlog implementation of the asynchronous rules

```
% diamond modality
check Cert (foc (lform L (dia B))) :-
  dia_ke Cert (lform L (dia B)) T Cert',
  rel L T,
  check Cert' (foc (lform T B)).
% exists quantifier
check Cert (foc (lform L (some B))) :-
  some_ke Cert (lform L (some B)) T Cert',
  check Cert' (foc (lform L (B T))).
```

Figure 6: λProlog implementation of the Synchronous rules

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 or a world. Fig. 7 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`, `diamond` and `exists` predicates. I have simplified the implementation to include only negative conjunctions and disjunctions (see Sec. 3). The addition of the positive versions does not fundamentally change the approach presented here. In case we would like to support these two inference rules, 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 asking the user to supply this information as can be seen in the implementation of the predicates `decide` (lines 1-3), `diamond` (lines 25-32) and `some` (lines 42-49).

The proof control I use contains 5 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

- an index marking the current inference rule – this index is used to store labeled formulas in a consistent way
- the list of fresh worlds generated so far – this list is used in order to allow the user to pick up a world The user, of course, has no access to the fresh worlds (or to any other part in the trusted kernel) and I use a mechanism discussed below in order to allow her to supply them
- the list of fresh variables generated so far – Similarly to the list of fresh worlds, this list is used in order to allow the user to supply term witnesses which contain fresh variables

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 commands 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 `diamond` or `some` predicates, the input is the term witness.

In the case of the `diamond` and on some cases, also for the `some` inference rule, the implementation needs to substitute fresh variables inside the term supplied by the user. I use λProlog abstraction and β-normalization directly. The user keeps track on the number n and order of both fresh worlds and fresh variables introduced so far and the chosen world or the term witness is then of the form $x_1\backslash \dots x_n\backslash t$ where t may contain any of the bound variables, in the case of `some`, or the actual chosen world, in the case of `diamond`. The `apply_vars` predicate is responsible for applying to the terms the fresh variables in the correct order. It should be noted that this cumbersome mechanism can be easily replaced by a naming mechanism given a more sophisticated user interface.

The indexing mechanism I use is based on trees and assign 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`. Note that the indexing system is based on inferences and that indices are assigned to formulas only upon storing them in the context. For example, if our theorem is `A &-& B`, meaning a negative conjunction, then this formula is assigned the index `e`. In the focused sequent calculus, the only inference rule which can be applied right now is the negative conjunction and the left and right derivations keep track of the indices `(l e)` and `(r e)`. Only in case we store the formulas `A` or `B` in the following step will we assign them the indices `(l e)` or `(r e)`

I note here that the implementation is using a relatively 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 Barcan formula, which is a theorem of modal logic K with constant domains and rigid terms. In order to use the assistant, the user needs to call the prover with the theorem and an empty commands list.

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

Since we start in a negative phase and the theorem is negative, the assistant eagerly does the following ordered steps, the first five of which are asynchronous.

```
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_ke (interact (unary (decideI I) L) [I|Com] FI E1 E2) I
    (interact L Com (u FI) E1 E2).
store_kc (interact (unary (storeI I) L) Com I E1 E2) F I (interact L Com (u I) E1 E2) :-
  output std_out "Adding␣to␣context␣formula␣",
  term_to_string F S1,
  output std_out S1,
  output std_out "␣with␣index␣",
  term_to_string I S2,
  output std_out S2,
  output std_out "\n".
release_ke (interact (unary releaseI L) Com FI E1 E2) (interact L Com (u FI) E1 E2).
initial_ke (interact (axiom (initialI I)) []  _ _ _) I.
orNeg_kc (interact (unary (orNegI FI) L) Com FI E1 E2) F (interact L Com (u FI) E1 E2).
andNeg_kc (interact (binary (andNegI FI) L1 L2) [branch LC RC] FI E1 E2) F
    (interact L1 LC (l FI) E1 E2) (interact L2 RC (r FI) E1 E2).
box_kc (interact (unary (boxI FI) L) Com FI E1 E2) F (Eigen\ (interact L) Com (u FI)
    [eigen FI Eigen| E1] E2) :-
  output std_out "Using␣world␣variable␣",
  term_to_string FI S1,
  output std_out S1,
  output std_out "\n".
dia_ke (interact (unary (diaI no_index) leaf) [] FI _ _) F _ _ :- !,
  output std_out "You␣have␣to␣choose␣the␣world␣to␣use␣for␣instatiation␣for␣the␣formula:␣",
  term_to_string F S1,
  output std_out S1,
  output std_out "\nAt␣index:␣",
  term_to_string FI S2,
  output std_out S2,
  output std_out "\n", fail.
dia_ke (interact (unary (someI FI) L) [W|Com] FI E1 E2) _ W'
    (interact L Com (u FI) E1 E2) :-
  apply_vars W E1 W'.
all_kc (interact (unary (allI FI) L) Com FI E1 E2) F
    (Eigen\ (interact L) Com (u FI) E1 [eigen FI Eigen| E2]) :-
  output std_out "Using␣eigen␣variable␣",
  term_to_string FI S1,
  output std_out S1,
  output std_out "\n".
some_ke (interact (unary (someI no_index) leaf) [] FI _ _) F _ _ :- !,
  output std_out "You␣have␣to␣choose␣the␣term␣to␣use␣for␣instantiation:␣",
  term_to_string F S1,
  output std_out S1,
  output std_out "\nAt␣index:␣",
  term_to_string FI S2,
  output std_out S2,
  output std_out "\n", fail.
some_ke (interact (unary (someI FI) L) [T|Com] FI E1 E2) _ T'
    (interact L Com (u FI) E1 E2) :-
  apply_vars T E2 T'.
apply_vars T [] T.
apply_vars T [eigen _ X|L] T' :-
  apply_vars (T X) L T'.
```

Figure 7: λProlog implementation of basic interaction with the user

```
Adding to context formula lform z (some x0 \ dia (n (q x0))) with index u e
Using world variable u (u e)
Using eigen variable u (u (u e))
Adding to context formula lform x0 (p (q x1)) with index u (u (u (u e)))
Adding to context formula lform x0 (n (q x1)) with index u (u (u (u (u (u (u (u (u e))))))))
You have to choose an index of a formula to decide on from the context
At index: u (u (u (u (u (u (u (u (u (u e)))))))))
```

Figure 8: A screenshot of an intermediary step of proving the Barcan formula

1. Applies the $\vee^-_K$ inference rule
2. Adds to the context the positive formula `lform z some x\ dia (n (q x))` with index `u e` (`z` denotes the starting world)
3. Applies the $\Box_K$ in order to produce a fresh world
4. Applies the $\forall_K$ in order to produce a fresh variable
5. Adds to the context the positive formula `lform x0 (p (q x1))` with index `u (u (u (u e)))`
6. Prompts the user to input an index to decide on

Using the provided simple user interface, the program now terminates and the user is expected to run it again, this time setting the list provided in the second argument to contain an interactive command. The first interactive command is therefore, to choose the index `u e`. We are now entering the synchronous phase and are asked also to supply the witness for the $\exists_K$ rule. In this case, the witness is just the (only) fresh variable introduced earlier and we command the assistant to choose `(x\x)`. Still being in a synchronous phase, we are now asked to supply the world to satisfy the $\Diamond_K$ rule. We choose the first (and only) previously introduced world using `(x\x)`. The assistant now observes that we have the negative atom `lform x0 (n (q x1))` with index `u (u (u (u (u (u (u (u (u e))))))))`. We are now asked again to pick up an index of a formula to decide on. We observe that the context contains the positive and negative versions of the same atom (in the same world) and we decide on the positive version with index `u (u (u (u e)))`. The $init_K$ rule is automatically applied and the assistant responds with the formal proof we have obtained.

The last execution is therefore,

```
$>./run.sh '((some x\ dia (n (q x))) !-! (box (all x\ (p (q x)))))'
  '[(u e),(x\x),(x\x),u (u (u (u e)))]'
```

Fig. 8 shows a screenshot of the interaction after supplying the input `'[(u e),(x\x),(x\x)]'`

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

```
decide_ke (interact (unary (decideI I) L) [auto|Com] FI E1 E2) I
    (interact L Com (u FI) E1 E2).
dia_ke (interact (unary (diaI FI) L) [world|Com] FI E1 E2) _ T
    (interact L Com (u FI) E1 E2) :-
  apply_vars T E1 T'.
some_ke (interact (unary (someI FI) L) [var|Com] FI E1 E2) _ _
    (interact L Com (u FI) E1 E2) :- !.
```

Figure 9: λProlog implementation of some tactics

term. Fig. 9 presents the additional predicates we need to add in order to support several basic tactics.

The `auto` tactic, which is supplied when asked to decide on a formula, attempts to choose one according to the order they are stored in the λProlog database. Similarly, the `world` tactic attempts to choose a world according to the order we have stored them in the proof control. Coming up with a witness is more complex. Unlike with deciding on a formula or selecting a world, we are now facing a possibly infinite number of options. Luckily, λProlog can again help us with the task. We can use the language metavariables and λProlog will postpone the choice until it can unify this variable with an appropriate term. The `var` tactic therefore replaces the chosen term with such a metavariable.

Using these tactics, the commands required in order to prove the theorem from the example is

```
$>./run.sh '((some x\ dia (n (q x))) !-! (box (all x\ (p (q x)))))'
  '[auto,var,world,auto]'
```

It should be noted though, that in the general case the simple tactics presented may not be as easy to use. For example, when deciding using the `auto` command, the system will decide on the last formula stored in the context and only if it fails later to find a proof, will it backtrack and pick another. This scheme will therefore get very confusing if we are also prompt to input information on the wrong branch. One can think of more advanced tactics which present to the user all possible paths and not just one as in the current implementation.

5 Conclusion and further work

The aim of this paper was to investigate the applicability of a minimal proof assistant based on focusing and λProlog for interactive proof search in first-order modal logic. We have considered also the amount of work required in order to design proof assistants for arbitrary focused systems.

The main target audience of this approach are users who are in need of a proof assistant for logics which do not enjoy an abundant number of tools. The next step is to try to apply this approach to concrete domains where interactive tools are scarce, such as in deontic logic ([5] contains some interesting recent developments). 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] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. Log. Comput.*, 2(3):297–347, 1992.

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

[4] Christoph Benzmüller and Bruno Woltzenlogel Paleo. Interacting with modal logics in the coq proof assistant. In *International Computer Science Symposium in Russia*, pages 398–411. Springer, 2015.

[5] Christoph Benzmüller, Xavier Parent, and Leendert van der Torre. A deontic logic reasoning infrastructure. In Russel Miller, Dirk Nowotka, and Florin Manea, editors, *14th Conference on Computability in Europe, CiE 2018, Kiel, Germany, July 30-August, 2018, Proceedings*. Springer, 2018. To appear.

[6] Christoph Benzmüller and Bruno Woltzenlogel Paleo. Higher-order modal logics: Automation and applications. In Adrian Paschke and Wolfgang Faber, editors, *Reasoning Web 2015*, number 9203 in LNCS, pages 32–74, Berlin, Germany, 2015. Springer. (Invited paper).

[7] Patrick Blackburn and Johan Van Benthem. Modal logic: a semantic perspective. In *Studies in Logic and Practical Reasoning*, volume 3, pages 1–84. Elsevier, 2007.

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

[9] Torben Braüner and Silvio Ghilardi. 9 first-order modal logic. In *Studies in Logic and Practical Reasoning*, volume 3, pages 549–620. Elsevier, 2007.

[10] Serenella Cerrito and Marta Cialdea Mayer. Free-variable tableaux for constant-domain quantified modal logics with rigid and non-rigid designation. In *International Joint Conference on Automated Reasoning*, pages 137–151. Springer, 2001.

[11] Boutheina Chetali and Quang-Huy Nguyen. 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, 2008.

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

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

[14] Marta Cialdea. Resolution for some first-order modal systems. *Theoretical Computer Science*, 85(2):213–229, 1991.

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

[16] Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. Elpi: Fast, embeddable, \lambda prolog interpreter. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 460–468. Springer, 2015.

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

[18] Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In *International Conference on Automated Deduction*, pages 61–80. Springer, 1988.

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

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

[21] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot,

Stéphane Le Roux, Assia Mahboubi, Russell OConnor, Sidi Ould Biha, et al. A machine-checked proof of the odd order theorem. In *International Conference on Interactive Theorem Proving*, pages 163–179. Springer, 2013.

[22] Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. Implementing type theory in higher order constraint logic programming. 2017.

[23] 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. A formal proof of the kepler conjecture. In *Forum of Mathematics, Pi*, volume 5. Cambridge University Press, 2017.

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

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

[26] Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theor. Comput. Sci.*, 410(46):4747–4768, 2009.

[27] Sonia Marin, Dale Miller, and Marco Volpe. A focused framework for emulating modal proof systems. In *Advances in Modal Logic 11, proceedings of the 11th conference on "Advances in Modal Logic," held in Budapest, Hungary, August 30 - September 2, 2016*, pages 469–488, 2016.

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

[29] Dale Miller. A proposal for broad spectrum proof certificates. In *International Conference on Certified Programs and Proofs*, pages 54–69. Springer, 2011.

[30] Dale Miller. Mechanized Metatheory Revisited: An Extended Abstract . In *Post-proceedings of TYPES 2016* , Novi Sad, Serbia, 2017.

[31] Dale Miller and Gopalan Nadathur. *Programming with higher-order logic*. Cambridge University Press, 2012.

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

[33] Dale Miller and Marco Volpe. Focused labeled proof systems for modal logic. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 266–280. Springer, 2015.

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

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

[36] Jens Otten. Mleancop: A connection prover for first-order modal logic. In *International Joint Conference on Automated Reasoning*, pages 269–276. Springer, 2014.

[37] Livio Robaldo and Xin Sun. Reified input/output logic: Combining input/output logic and reification to represent norms coming from existing legislation. *Journal of Logic and Computation*, 27(8):2471–2503, 2017.

[38] Enrico Tassi. Elpi: an extension language for coq metaprogramming coq in the elpi λprolog dialect. 2017.