A Proof-Planning Framework with explicit Abstractions based on Indexed Formulas

Serge Autexier

FR 6.2 Informatik, Saarland University, P.O. Box 15 11 50, 66041 Saarbrücken, Germany, e-mail:autexier@ags.uni-sb.de

Abstract. A major motivation of proof-planning is to bridge the gap between high-level, cognitively adequate reasoning for specific domains, and calculus-level reasoning to ensure soundness. For high reasoning levels the cognitive adequacy of representation and reasoning techniques is a major issue, while for lower reasoning levels the adequacy wrt. the modelled domain is important. Furthermore, proof construction is an engineering task and there is a need to support the design and application of proof-search engineering methods. To this end we present a framework to explicitly support different reasoning levels. To structure reasoning levels the framework allows for an explicit representation of abstractions and proof-search refinement techniques. In order to ensure soundness within a reasoning level, we use techniques developed in the context of matrix characterisation relying on the notion of indexed formulas. Furthermore, we introduce a uniform concept for contextual reasoning, and sketch basic tacticals for the definition of tactics to organise the overall proof-search inside and across different reasoning levels.

Keywords: Proof-planning, Methodology, Abstraction, Matrix Characterisations

1 Introduction

A major motivation for proof-planning is to allow on the one hand for a high-level reasoning which is (cognitively) adequate for specific problems and domains. On the other hand the final product of the reasoning process must be a calculus level proof wrt. some logic, in order to have a verifiably sound proof. Besides the cognitive motivation, another reason for the different reasoning levels is that the construction of a proof for some theorem is an *engineering task*. Analogously to the development of software, where one starts with a high-level specification of a program that is subsequently refined to an executable program in some target programming language, we view proof-planning as a methodology to design (and subsequently apply) different reasoning levels as well as methods to stepwise refine *proof specifications* (i.e. proof plans). Hence, proof-planning has been introduced as a methodology to refine high-level proof-plans of a theorem to a calculus level.

The objectives of the different reasoning levels are different: for high reasoning levels, the adequacy of the representation and the reasoning techniques is a key issue, in order to allow for an intuitive reasoning, which is as close as possible to say the style of a human. Although the reasoning inside one reasoning level should be sound, there is another notion of *relative soundness* due to the presence of different reasoning levels. A high reasoning level is *relatively sound* wrt. some lower reasoning level, if any high level proof can be refined to some low level proof. Usually, this relative soundness does not hold, which is a well-known effect in the presence of abstractions [5]. We do not want to enforce the *relative soundness* of high reasoning levels, as this usually may hamper the adequacy of representations, whereas the primary purpose of those levels is to allow for an efficient and adequate search for a proof plan. The relative soundness of a *proof* on a high reasoning level is established by stepwise refinement of this proof to a calculus level proof. On the lower reasoning levels we are less concerned with the cognitive adequacy of representations, but rather with the adequacy wrt. the modelled domain, as for example some program behaviour or a mathematical domain.

In this paper we present a proof-planning framework that supports the design and application of proof search engineering methods. First, we define in Sect. 2 the notion of a *reasoning level*, which consists of *reasoning objects* (e.g formulas, sequents, etc.) and *reasoning procedures* (e.g. tactics, planning algorithms, general theorem proving algorithms, etc.). For example, in a proof-planner like [10, 2], the reasoning objects are sequents as well as the pre- and postconditions of some methods, which are used as inference rules. The reasoning procedure of a proof-planner is the depth-first search algorithm applying the methods. To ensure the soundness wrt. some logic within a reasoning level, a variety of logics are predefined in the framework, and a uniform mechanism to encode those logics is introduced. For sake of representational adequacy, the logics are not embedded in some meta-logic. Instead we present in Sect. 4 techniques developed in the context of matrix characterisation [12, 9] that rely on the notion of *indexed formulas*. Those allow for a uniform treatment of a large class of classical, modal and intuitionistic logics, without having to change the syntax of the formulas.

Secondly, we define in Sect. 3 vertical structuring mechanisms to relate different reasoning levels. This includes on the one hand the explicit definition of abstractions, which define how reasoning objects of a higher reasoning level are related to those of a lower level. On the other hand, we introduce the notion of refinements, which determine how higher-level reasoning procedure can be implemented by some lower-level reasoning procedures. This subsumes e.g. the notion of refinement encoded in a method of a proof-planner like [10, 2], where the tactic wrapped inside a method determines the refinement of the abstract rule described by the pre- and postcondition of the method. Based on the general notion of refinement, the refinement of an abstract proof can be defined in a uniform manner.

Thirdly, reasoning procedures should be definable in an intuitive manner, again because of the overall motivation of cognitive adequacy. To this end we present in Sect. 5 window inference reasoning, which is a formalisation of an intuitive reasoning style, that allows to focus the reasoning on sub-formulas and sub-terms, and aims at contextual reasoning. We integrate window inference reasoning with the indexed formulas from Sect. 4 and present as a by-product how this allows for contextual reasoning in a canonical way.

Finally, we present in Sect. 6 the basic programming language to define reasoning and refinement procedures and how to combine those in order to organise the overall proof search. These build upon the intuitive reasoning rules provided by window inference reasoning and have an explicit failure and success semantics.

2 Reasoning Levels

In this section we introduce the notion of *reasoning levels*. Those are the *horizontal structures* of the framework in opposition to the abstraction and refinement techniques (cf. Sect. 3) that are the *vertical structures*.

A reasoning level consists on the one hand of the *objects* we want to reason about as well as the *basic reasoning rules*, and on the other hand of the reasoning procedures organising the proof search within one reasoning level. The former are defined by a signature and a set of formulas wrt. some logic. Thus, the reasoning objects and rules of a reasoning level are given by a pair (Σ, Φ), where Σ is the signature and Φ is a set of "axioms", and we denote such a pair as a *representation*. This is similar to the notion of a specification in formal specification languages [8], which are e.g. used to define abstract datatypes and functions about abstract datatypes. For a reasoning level, the signature Σ determines the syntax of the *reasoning objects* and the available *basic reasoning rules* are determined by the "axioms" from Φ.

Example 1. As an example we present a representation called `TLA` for TLA formulas [7] and a representation `TLA-States&Transitions` for the abstraction of TLA formulas. The signature for `TLA` contains the usual logical connectives, as well as $\Box$, $\Diamond$, and also WF to express fairness requirements. As an example we use a simple program, which has a counter x of initial value 15 and that decreases this counter in every step if it is greater than 0. This is encoded in TLA by the following formula

$$x = 15 \wedge \Box(x \geq 0 \wedge x' = x - 1) \wedge WF(x \geq 0 \wedge x' = x - 1)_x$$

In order to prove that the counter eventually has the value 0 (encoded by $\Diamond x = 0$), we use as abstraction a representation consisting of states and state transitions. This representation is called `TLA-States&Transitions` and consists of states $\langle n \rangle$ representing the state, where x has the value n, and state transitions represented by $\forall n : \langle n \rangle \Rightarrow \langle n - 1 \rangle$. This is an abstraction of the transition made by the program, i.e. $x \geq 0 \wedge x' = x - 1$. Finding a proof-plan for $\Diamond x = 0$, consists in finding a proof for $\langle 0 \rangle$ on the (abstract) representation `TLA-States&Transitions`. $\Box$

The second component of a reasoning level are the *reasoning procedures*, which organise the proof search within the reasoning level. We adopt a general notion for a reasoning procedure, by defining it to be an algorithm in some programming language, which interacts with the proof by applying basic inference rules. Additionally a reasoning procedure comes with a notion of `success` and `failure` to reflect about its own behaviour. E.g. in case of `failure` a reasoning procedure might invoke backtracking to some previous proof state. This notion of a reasoning procedure is general enough to subsume various kinds of specific reasoning procedures developed in automated theorem proving, like, e.g., a tactic in a tactical theorem prover, the various algorithms in an automated theorem prover, or a proof-planning algorithm in some proof-planner.

In summary, we have defined a reasoning level to consist of two parts, namely *representations* defining the syntax of the reasoning objects and *reasoning procedures* to organise the proof-search inside the reasoning level. This can be sketched by the following equation:

$$\mathsf{Reasoning\ level} = \mathsf{Representation} + \mathsf{Reasoning\ procedures}$$

The next step consists in relating reasoning levels. To this end *vertical structures* are introduced in the next section, that describe the mechanisms to switch between reasoning levels.

3 Structuring Reasoning Levels

In this section we define the mechanisms to structure the different reasoning levels of the framework (cf. Fig. 1). There are two dual notions of *vertical structures*: first, *abstractions* can be defined which describe how objects of a higher reasoning level can be constructed from objects of a lower reasoning levels. Formally, an abstraction is a mapping of a set of objects of a source representation R_S to a set of objects of some target representation R_T. For the purpose of the framework, we define an abstraction to consist of a *name*, the source and target representations R_S and R_T, and an *abstraction procedure* (i.e. an algorithm) to compute the mapping.

Example 2. Consider as an example the source representation `NAT` of natural numbers and labelled fragments (`LF-NAT`) as respective target representation. *Labelled fragments* are known abstractions in the context of inductive theorem proving, where they are used to abstract over the specific differences (so-called *wave-fronts*) between two terms sharing a common *skeleton*. The representation `NAT` consists of a signature for natural numbers and – among others – of two axioms defining the addition of natural numbers

$$\forall y\,.\,0 + y = y \tag{1}$$
$$\forall x, y\,.\,s(x) + y = s(x + y) \tag{2}$$

The representation `LF-NAT` consists on the one hand of a signature extending the signature of `NAT` by a function $\bullet : Nat \to Nat$ that is used to represent the wave-fronts, and on the other hand of the axioms

$$\forall y\,.\,\bullet(y) = y \tag{3}$$

Refinement

Refinement

Abstraction

Abstraction

Reasoning Level 3 = Representation-3 + Reasoning Procedures 3

Reasoning Level 2 = Representation-2 + Reasoning Procedures 2

Reasoning Level 1 = Representation-1 + Reasoning Procedures 1

Fig. 1. Structuring reasoning levels

$$\forall x, y \,.\, \bullet(x) + y = \bullet(x + y) \tag{4}$$

$$\forall x \,.\, \bullet(s(x)) = s(\bullet(x)) \tag{5}$$

The *labelled-fragment abstraction* is an abstraction from `NAT` to `LF-NAT` and its associated *abstraction procedure* maps the singleton $\{(1)\}$ to the singleton $\{(3)\}$, and the singleton $\{(2)\}$ to the set $\{(4), (5)\}$. $\square$

Thus, an *abstraction* consists of the source and target *representations* and a function mapping objects from the source representation to objects of the target representation. This can be expressed in an intuitive manner by the following equation:

Abstraction = Source & Target Representations + Abstraction procedure

The dual *vertical structuring* mechanism is the *refinement* of higher-level reasoning procedures to lower-level reasoning procedures. Take as an example of a high-level reasoning procedure some planning algorithm, which computes a proof-plan from methods. This proof-plan is a “proof” on the reasoning level of the proof-planning procedure, and consists of the application of basic rewriting rules defined by the pre- and postconditions of the methods. We can associate to this planning procedure a *refinement procedure*, which takes generated proof-plan and refines it by calling the tactics wrapped inside the different methods. In this scenario the refinement information is associated to each basic reasoning rule of the higher reasoning level via the methods. A method describes that an abstract proof step satisfying a certain pattern (described by the pre- and postcondition of the method) could be refined by the tactic wrapped inside the method. The refinement procedure exploits this knowledge in order to refine a proof plan. In general it is not possible to describe this refinement information by patterns. In order to deal with the general case, we allow to associate a refinement procedure to some reasoning procedure. Such a refinement procedure takes the proof generated by the reasoning procedure as a proof-plan to compute a proof on the lower reasoning level (using the lower-level reasoning procedures). Similarly to reasoning procedures, refinement procedures come with an explicit `success` and `failure` semantics to reflect about their own behaviour. This allows for example to backtrack the refinement process and to try alternative refinements.

From the above we can summarise the notion of refinement by stating that a refinement is composed of a reasoning procedure and an associated refinement procedure. This can be sketched by the following equation:

$$\mathsf{Refinement} = \mathsf{Reasoning\ procedure} + \mathsf{Refinement\ procedure}$$

Note that a refinement procedure for some reasoning procedure $\mathcal{P}$ can dually be interpreted as some kind of abstraction, where all the (lower-level) reasoning procedures used by the refinement procedure are abstracted to the reasoning procedure $\mathcal{P}$. This notion of abstraction is different from the notion presented above, since it is an abstraction of reasoning procedures while the former is an abstraction of objects and axioms. Furthermore do refinement procedures exploit knowledge about object abstractions during the refinement. Hence, an object abstraction also gives rise to some notion of refinement. The reason why we introduced abstractions mainly for objects and refinement mainly for reasoning procedures is because this is the usual way those are used.

4 Handling Soundness

In this section we define a uniform mechanism to handle soundness for a variety of logics. For sake of the adequacy of representation, we want to support directly the soundness wrt. some logic without altering the syntax of formulas. Subsequently, we do not want to embed a logic in some meta-logic, like e.g. intuitionistic higher-order logic. In comparison to the *representations* introduced in Sect. 2, a *logic* also defines the syntax of reasoning objects, but comes with its own formal proof-theory (whereas a representation is relative to some logic, and thus "inherits" the proof-theory from this logic). Thereby its proof-theory determines, for example, whether in a sequent calculus for this logic an elimination rule for some object-level quantifier must satisfy the *Eigenvariable* condition, or how other parts of a sequent are affected by the elimination rule for some $\Box$ or $\Diamond$ in some modal logic.

A background mechanism to handle the proof-theory of different logics must allow for a uniform handling of soundness and completeness results for those logics. Such a basis for propositional and first-order, classical, modal and intuitionistic logics is provided by *indexed formula trees* [12] and relies on polarities of (sub-)formulas and uniform notation [12, 4, 3].

The starting point for uniform notation is the concept of *signed formulas*, which are formulas φ annotated by some positive (φ^+) or negative (φ^-) polarity. Intuitively, a signed formula φ^+ occurs in the succedent of a sequent, i.e. $\Gamma \vdash \varphi, \Delta$, and φ^- occurs in the antecedent of a sequent, i.e. $\Gamma, \varphi \vdash \Delta$. The signed formulas can be categorised into different classes, according to their behaviour during the proof search: consider as an example the sequent $\vdash \varphi \vee \psi$. The application of the respective sequent decomposition rule leads to the single sequent $\vdash \varphi, \psi$. All those kinds of signed formulas are said to be of type α. The complete list of signed formulas of this type are given in the table for α in Fig. 2.

α	α_0	α_1
$(\varphi \vee \psi)^+$	φ^+	ψ^+
$(\varphi \Rightarrow \psi)^+$	φ^-	ψ^+
$(\varphi \wedge \psi)^-$	φ^-	ψ^-
$(\neg\varphi)^+$	φ^-	–
$(\neg\varphi)^-$	φ^+	–

β	β_0	β_1
$(\varphi \wedge \psi)^+$	φ^+	ψ^+
$(\varphi \vee \psi)^-$	φ^-	ψ^-
$(\varphi \Rightarrow \psi)^-$	φ^+	ψ^-

γ	$\gamma_0(c)$
$(\forall x \,.\, \varphi)^-$	$(\varphi[x/c]^-$
$(\exists x \,.\, \varphi)^+$	$(\varphi[x/c]^+$

δ	$\delta_0(c)$
$(\forall x \,.\, \varphi)^+$	$(\varphi[x/c]^+$
$(\exists x \,.\, \varphi)^-$	$(\varphi[x/c]^-$

ν	ν_0
$(\Box\varphi)^-$	φ^-
$(\Diamond\varphi)^+$	φ^+

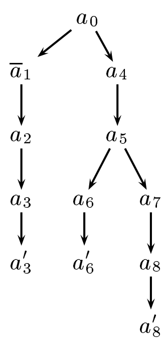
π	π_0
$(\Box\varphi)^+$	φ^+
$(\Diamond\varphi)^-$	φ^-

Fig. 2. Uniform notation for signed formulas

Furthermore, those signed formulas whose decomposition leads to a split on the sequent proof tree are said to be of type β (cf. Fig. 2). Signed formulas of type γ correspond to instantiation rules, without restrictions to the substituted terms, while for δ-type signed formulas the instantiation must respect the *Eigenvariable* condition (e.g. $(\forall x \,.\, \varphi)^+$, which corresponds to $\vdash \forall x \,.\, \varphi)$. For modal logic formulas, the decomposition of ν-type signed formulas does not affect the other formulas in a sequent (e.g. $\Gamma, \Box\varphi \vdash \Delta$), while the decomposition of π-type signed formulas changes the other parts of the sequent (e.g. $\Gamma \vdash \Box\varphi, \Delta$).

In Fig. 2 the rules also indicate how the polarity is inherited to the sub-formulas of a signed formula. Using these rules, polarities can be assigned to all sub-formulas of a formula and the polarity of a subexpression indicates whether this sub-formula will occur in the antecedent or the succedent of a sequent, if all structural sequent decomposition rules are applied until this formula. E.g. the signed formula $(\varphi \Rightarrow \psi)^-$ corresponds to $\Gamma, \varphi \Rightarrow \psi \vdash \Delta$. The β-rule reduces this signed formula into the signed formulas φ^+ and ψ^-. This corresponds to what happens if the $\Rightarrow$*-Left*-decomposition rule is applied on $\Gamma, \varphi \Rightarrow \psi \vdash \Delta$, since it results in a split of the proof (indicated by β), and φ occurs in the succedent of the first premise $\Gamma \vdash \varphi, \Delta$, and ψ occurs in the antecedent of the second premise $\Gamma, \psi \vdash \Delta$.

Indexed formula trees as introduced in [12] exploit the tree structure of formulas and annotate each sub-formula according to the reduction rules with their respective polarity and uniform type. Consider the indexed formula tree in Fig. 3 of the signed formula $((\Box\forall x \,.\, P(x)) \Rightarrow (\exists y \,.\, (P(y) \wedge \Box P(y))))^+$. For each sub-formula there is a position a_i that is either a position constant or position variable (which are over-lined, like $\overline{a}_1$). A set of properties associated to each position is listed in the adjacent table: those are the actual sub-formula (called the *label* of the position), its polarity, its uniform type (called *primary type*), and its *secondary type*, which is its type inside the parent formula reduction rule. According to those reduction rules, only sub-formulas that are related by an α-type position may occur together in a final sequent (after application of the respective sequent calculus decomposition rules). This can be illustrated by the sub-formulas $\forall x \,.\, P(x)$ and $P(y)$ that are related by the position a_0, or, similarly, the sub-formulas $\forall x \,.\, P(x)$ and $\Box P(y)$). Hence, from the indexed formula tree we obtain a simple notion of a *context* for some subexpression, i.e. those



u	$Pol(u)$	$Lab(u)$	$Ptype(u)$	$Stype(u)$
a_0	$+$	$(\Box(\forall x\,.\,P(x))) \Rightarrow (\exists y\,.\,(P(y) \wedge \Box P(y)))$	α	$-$
$\overline{a}_1$	$-$	$\Box\forall x\,.\,P(x)$	ν	α_1
a_2	$-$	$\forall x\,.\,P(x)$	γ	ν_0
a_3	$-$	$P(x)$	$-$	γ_0
a'_3	$\circ$	x	$-$	$-$
a_4	$+$	$\exists y\,.\,(P(y) \wedge \Box P(y))$	γ	α_2
a_5	$+$	$P(y) \wedge \Box P(y)$	β	γ_0
a_6	$+$	$P(y)$	$-$	β_1
a'_6	$+$	y	$-$	$-$
a_7	$+$	$\Box P(y)$	π	β_2
a_8	$+$	$P(y)$	$-$	π_0
a'_8	$\circ$	y	$-$	$-$

Fig. 3. Indexed Formula tree

parts of the signed formula tree, that can be "used" for this subexpression. This is exploited in Sect. 5 to handle contextual reasoning.

The tree structure of the indexed formula tree induces an ordering $\prec$ among the positions. A first-order substitution σ instantiates a variable bound on some position a of primary type γ and induces an additional ordering relationship between the position a and γ- or δ-type positions binding variables in $\sigma(x)$. Combined with the original ordering $\prec$, we obtain a relation $\lhd$. Such a first-order substitution is only sound, if $\lhd$ is irreflexive.

Furthermore, we can associate to any position a a string of positions, consisting of some of the positions governing a (i.e. smaller wrt. $\prec$) and that are of primary type ν or π. Such a string is called the modal prefix of the position a. Which positions are actually used in the prefix depends on the modal logic under consideration (K, K4, S4, etc., cf. [12] for details). The variable positions are exactly all positions of primary type ν. *Modal substitutions* are used to instantiate the position variables by modal position strings. Depending on the modal logic under consideration, there is a set of restrictions how the position variables can be instantiated (cf. [12]). Furthermore, similar to first-order substitutions a modal substitution induces an ordering among the ν- and π-type positions and again the induced overall ordering $\lhd$ must be irreflexive. All the requirements about the instantiation of variables, position variables, the irreflexivity of induced overall orderings $\lhd$ among positions are subsumed by the uniform notion of $\mathcal{L}$-admissible substitutions, where $\mathcal{L}$ is either classical first-order logic, or propositional or first-order modal logic K, K4, D, D4, T, S4, or propositional or first-order intuitionistic logic. The uniform notion of $\mathcal{L}$-admissible substitutions is the key concept allowing for a uniform treatment of a large class of logics.

Based on these techniques, we introduce the uniform notion of *valid proof states* (i.e., a signed formula tree with an irreflexive ordering $\lhd$). From [12] we obtain, that the application of $\mathcal{L}$-admissible substitutions preserves the validity of proof states. The problem to compute $\mathcal{L}$-admissible substitutions can be solved by using a standard unification procedure with occur-check for object variables, and the generic prefix unification procedure from [9], which computes only $\mathcal{L}$-admissible substitutions for position variables, where $\mathcal{L}$ is any of the above logics. The notion of valid proof states can subsequently be used to formally prove the $\mathcal{L}$-soundness of any other transformation we might define on the signed formula tree. Hence, using signed formula trees gives us a powerful and uniform mechanism to support a variety of logics inside the framework.

5 Intuitive and Contextual Reasoning

For the design of reasoning and refinement procedures we support an intuitive reasoning style. To this end we aim to focus the reasoning process on arbitrary sub-expressions of some actual expression and to have a strong support of contextual reasoning inside a focus. Window inference [11, 6] has been introduced as a formalisation of an intuitive and hierarchical reasoning style. The basic objects are windows

$$\langle [] \vdash P(a) \Rightarrow \neg(a = b)\rangle^{+}$$

which represents that the focus is on the positive (+) formula $P(a) \Rightarrow \neg(a = b)$ (denoted as the *content* of the window) and the context of this focus is empty ([]). The basic window inference rules are

1. to focus on some subexpression of the content of a window

   $$\langle [] \vdash P(a) \Rightarrow \neg(a = b)\rangle^{+} \quad \longmapsto \quad \langle [a = b] \vdash P(a)\rangle^{-} \quad \longmapsto \quad \langle [a = b] \vdash a\rangle^{\circ},$$

   where $\longmapsto$ denotes a focus derivation step. Note that this rule must be defined for any logical connective (positive $\Rightarrow$ in the example) and argument position (first argument of $\Rightarrow$ in the example), in order to adjust the context in a consistent manner. While this is not difficult for classical logics, where the context of a sub-focus is either equal or an extension of the previous context, this gets difficult as soon as we deal with non-classical logics.
2. to transform the content of a window using information from the context

   $$\langle [a = b] \vdash a\rangle^{\circ} \quad \longmapsto \quad \langle [a = b] \vdash b\rangle^{\circ},$$
3. and to leave a focus and return to the parent focus

   $$\langle [a = b] \vdash b\rangle^{\circ} \quad \longmapsto \quad \langle [a = b] \vdash P(b)\rangle^{-} \quad \longmapsto \quad \langle [] \vdash P(b) \Rightarrow \neg(a = b)\rangle^{+}.$$

Furthermore, window inference reasoning also allows one to focus on the condition of a window. To this set of basic rules we added a general rule to allow one to introduce a case analysis:

$$\langle [H] \vdash t\rangle^{p} \quad \longmapsto \quad \langle [A \mid H] \vdash t\rangle^{p} \text{ and } \langle [B \mid H] \vdash t\rangle^{p},$$

In order to ensure the soundness of the case analysis[1] the condition $\langle [H] \vdash A \vee B\rangle^{+}$ is generated as an additional goal. In order to allow for an intuitive design of reasoning procedures, we integrated window inference reasoning with

[1] Even for intuitionistic logic.

indexed formula trees (cf. Sec. 4) by annotating positions in the indexed formula trees with windows. As focusing on subexpressions is along the tree structure, opening a sub-window corresponds to adding a window on the respective position. Similarly, leaving a focus means removing the window from a position and returning to the previous position.

The integration of indexed formula trees with windows results in the following structure of the framework: inside the framework, proof states are represented by indexed formula trees with windows. They serve as a soundness backbone managing any proof transformation, like e.g. the instantiation of variables or the application of rules from the context. The indexed formula trees are invisible to the reasoning procedures, which communicate with them via the windows annotated to specific positions and applying window inference rules, which result in a change of the proof state. An example for this is the opening of a sub-window, the instantiation of variables, etc. The difficulty remains that contextual information must be provided to the reasoning procedures, like what is the polarity of a window, which are the instantiable variables, and especially which are possible rules from the context of a window and whether they are applicable. These difficulties are elegantly resolved by using indexed formula trees:

Polarity of windows. The polarity of a window is simply the polarity of the position it is attached to in the indexed formula tree.

Instantiable Variables. The *instantiable* variables of some window are simply all variables that are bounded on some γ-type position in the formula tree. In order to check the admissibility of the instantiation of a variable, we check the $\mathcal{L}$-admissibility of the substitution (i.e. the irreflexivity of the new induced ordering $\lhd$, cf. Sec. 4).

Sampling contextual rules. The context of a window is uniformly determined by the position in the indexed formula: all parts of the indexed formula, that are related to the actual window by an α-type position, are in the *logical* context. Indeed, according to the reduction rules in Fig. 2, only α-related formulas may occur together in some final sequent after applying respective calculus decomposition rules (cf. Sec. 4). Hence, usable rules to change the content of a window are for example all *negative* equations and implications in the context. The application of a negative equation is basically a paramodulation rule, whereas the application of a negative implication is a resolution step. The *conditions* for such a rule are all the formulas that are β-related to this rule. From this we can define a uniform concept of a *replacement rule* $[\varphi]\ u \rightarrow v$: intuitively, this rule means that we can replace a content u of a window by v, if we prove the conditions φ. This notion can be defined formally as well as if it is admissible wrt. some focus (i.e. position).

Definition 1 (Admissible replacement rules). *Let a be a position of label s and polarity p in some indexed formula tree. Then $[\Phi]\ u \rightarrow v$ is an* admissible replacement rule *for a, if*

1. *u and v are either the left- and right-hand side of a negative equation, that is α-related to a, or u and v are β-related formulas with the same modal or intuitionistic prefixes, of polarities $-p$ and p respectively, and α-related to a,*
2. *and Φ contains all other formulas that are β-related to u and v.* □

Application of replacement rules. To apply a replacement rule on some focus, we must syntactically unify the left-hand side of the rule with the context of the focus as well as their modal prefixes. Furthermore, if the replacement rule is not an equation, the focus and the left-hand side of the rule must have opposite polarities. For unification we use standard unification procedures for object level variables and rely on the generic prefix unification procedure from [9], in order to compute only substitutions that are admissible for the actual logic. This allows to prove for all logics considered in [12] the soundness of the replacement rule application.

In summary, the integration of window inference reasoning with indexed formulas allows on the one hand for the design of intuitive reasoning procedures. On the other hand it provides a simple but powerful mechanism to support contextual reasoning.

6 Organising Proof Search

The overall proof search inside the framework is organised by the reasoning and refinement procedures, and some additional language constructs to combine them. The reasoning procedures are based on windows and interact with the proof in two manners: first, the content of a window can be modified by applying a replacement rule from the context of the window. Second, they can introduce a case analysis over some arbitrary formulas $A_1 \vee \ldots \vee A_n$ at any stage of the reasoning process. Following the intuitive reasoning paradigm, reasoning procedures can focus the proof search on some sub-window by calling another reasoning procedure on the sub-window. Finally, they can invoke the reasoning on some higher reasoning level by calling a reasoning procedure on some abstraction of the actual window. To this end they indicate the abstraction to be used and the reasoning procedure of the higher reasoning level to call. In case the abstract reasoning procedure succeeds, the refinement of the (abstract) proof is achieved by calling the refinement procedure of the abstract reasoning procedure. Only if the refinement procedure succeeds, the proof planning attempt is successful.

Example 3. Take as an example a reasoning procedure `prove-reachable` that is specialised to prove temporal properties of the form $\Diamond\varphi$ (cf. Fig. 4). Hence, the argument of the reasoning procedure is declared to be from the representation `TLA`. Furthermore let `abstract-prove-reachable` be the reasoning procedure on the representation `TLA-States&Transitions` introduced in Sect. 2, and let `Reachable-Abstraction` be the abstraction from the representation `TLA` to the representation `TLA-States&Transitions`. In Fig. 4 we sketch the

```
procedure abstract-prove-reachable (window:TLA-States&Transitions)
   ...
endprocedure

refinement abstract-prove-reachable-refine
   for abstract-prove-reachable
   ...
endrefinement

procedure prove-reachable (window:TLA)
 call-on-abstraction Reachable-Abstraction window abstract-prove-reachable
endprocedure
```

Fig. 4. TLA reasoning and refinement procedures

body of `prove-reachable`: this procedures calls the abstract reasoning procedure via `call-on-abstraction`, which abstracts the argument window to the representation `TLA-States&Transitions` and calls the abstract reasoning procedure `abstract-prove-reachable`. If this procedure succeeds, then the corresponding refinement procedure `abstract-prove-reachable-refine` is called, which refines the abstract proof on `TLA-States&Transitions` to a proof on `TLA` formulas. □

7 Conclusion

In this paper we took up the motivation that the task to construct a proof is an engineering task. This necessitates a methodology to design and apply proof engineering techniques and we emphasised that the techniques developed in proof-planning aim at these goals. Based on that we developed a proof-planning framework, that comes with an explicit notion of different reasoning levels. The reasoning levels can be structured via abstraction and refinement relations. Furthermore, we emphasised that cognitive adequate reasoning is a key issue for the design of higher reasoning levels. This entails on the one hand, that the representation of reasoning objects of the different reasoning levels should be intuitive. Subsequently, a uniform mechanism to support directly soundness wrt. a large variety of logics has been introduced, without having to alter the representation of logical formulas. Furthermore, we integrated this mechanism with an extension of the window inference reasoning style to allow for the design of intuitive reasoning procedures. A major benefit of this integration is a straightforward and elegant support of contextual reasoning, a key concept for intuitive reasoning. Finally, we sketched how proof-search engineering methods like reasoning procedures, abstractions and refinements can be defined in the framework, and how these techniques are linked together and subsequently can be applied for proof-search.

The framework serves as the basis for the ongoing implementation of the core reasoning mechanisms in the INKA 5.0 system [1]. Future work consists of finishing the implementation of the framework, and to validate the adequacy of the framework by implementing reasoning levels and structuring mechanisms for specific domains and problems. Theoretical future work is concerned with the integration of further logics into the framework.

References

1. Serge Autexier, Dieter Hutter, Heiko Mantel, and Axel Schairer. System description: INKA 5.0 – a Logic Voyager. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE)*, LNAI 1632, Trento, Italy, 1999. Springer.
2. Christoph Benzmüller, Lassaad Cheikhrouhou, Detlef Fehrer, Armin Fiedler, Xiarong Huang, Manfred Kerber, Michael Kohlhase, Karsten Konrad, Erica Melis, Andreas Meier, Wolf Schaarschmidt, Jörg Siekmann, and Volker Sorge. Ωmega: Towards a mathematical assistant. In W. McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction (CADE)*, LNAI 1249, Townsville, North Queensland, Australia, 1997. Springer.
3. M. Fitting. *First-Order Logic and Automated Theorem Proving / 2nd Edition*. Springer-Verlag New York Inc., 1996. ISBN 0-387-94593-8.
4. Melvin Fitting. Tableau methods of proof for modal logics. *Notre Dame Journal of Formal Logic*, XIII:237–247, 1972.
5. Fausto Giunchiglia and Toby Walsh. A Theory of Abstraction. *Journal of Artificial Intelligence*, 56(2-3):323–390, 1992. Also as technical report IRST-Technical Report 9001-14.
6. Jim Grundy. Window inference in the HOL system. In *Proceedings of the International Workshop on the HOL Theorem Proving System and its Applications*, 1991.
7. Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
8. J. Loeckx, H.-D. Ehrig, and M Wolf. *Specification of Abstract Data Types*. Teubner, Chichester;New York;Brisbane, 1996. ISBN 3-519-02115-3.
9. Jens Otten and Christoph Kreitz. T-string unification: Unifying prefixes in non-classical proof methods. In P Miglioli, U Moscato, and et. al., editors, *Proceedings of 5th Workshop on theorem Proving with analytic tableaux and related methods*, LNAI 1071, pages 244–260. Springer Verlag, 1996.
10. Julian D.C. Richardson, Alan Smaill, and Ian M. Green. System description: proof planning in higher-order logic with λ-clam. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction (CADE-98)*, LNAI 1421. Springer, 1998.
11. Peter D. Robinson and John Staples. Formalizing a hierarchical structure of practical mathematical reasoning. In *Journal of Logic and Computation*, volume 3, pages 47–61, 1993.
12. Lincoln Wallen. *Automated proof search in non-classical logics: efficient matrix proof methods for modal and intuitionistic logics.* MIT Press series in artificial intelligence, 1990.