φ Logic Programming Reasoner

Qiyuan Xu

29 April 2023

theory *Phi-Logic-Programming-Reasoner*
 imports *Main HOL−Eisbach.Eisbach HOL−Eisbach.Eisbach-Tools Phi-Document.Base*
 keywords *except* @*action* :: *quasi-command*
  and *φreasoner φreasoner-ML* :: *thy-decl* % *ML*
  and *print-φreasoners* :: *diag*
 abbrevs
  <*premise*> = premise
 and <*simprem*> = simprem
 and <@*GOAL*> = **@GOAL**
 and <*threshold*> = threshold
begin

0.0.1 Prelude Settings

ML ‹*Timing.cond-timeit false asd* (*fn* () => *OS.Process.sleep* (*seconds 1.0*))›

ML-file ‹*library/pattern.ML*›
ML-file ‹*library/helpers.ML*›
ML-file ‹*library/handlers.ML*›
ML-file ‹*library/pattern-translation.ML*›
ML-file-debug ‹*library/tools/simpset.ML*›

definition r*Require* :: ‹*prop* ⇒ *prop*› (r*REQUIRE* - [*2*] *2*) **where** [*iff*]: ‹r*Require* *X* ≡ *X*›

typedecl *action*

definition *Action-Tag* :: ‹*prop* ⇒ *action* ⇒ *prop*› (- @*action* - [*3*,*4*] *3*)
 where [*iff*]: ‹*Action-Tag P A* ≡ *P*›

lemma *Action-Tag-I*:
‹$P \Longrightarrow P$ @*action* A›
unfolding *Action-Tag-def* **.**

ML-file-debug ‹*library/reasoner.ML*›

lemma r*Require-I*[φ*reason 1000*]: ‹*PROP P* $\Longrightarrow$ *PROP* r*Require P*› **unfolding** r*Require-def* **.**

1 Introduction

φ-Logic Programming Reasoner is a extensible reasoning engine based on logic programming like Prolog. It allows arbitrary user reasoners to be integrated freely, and applies them selectively by matching the pattern of the goals.

The reasoning is a depth-first heuristic search guided by *priority* of each branch. A reasoning state is represented by a *pair* of `Proof.context` and a sequent, of type `Proof.context * thm`. Search branches on a reasoning state are admissible reasoners on the sequent. A reasoner is admissible on a sequent if the sequent matches the pattern of the reasoner (cf. patterns in section 2).

The reasoning accepts several reasoning states, and outputs *one* reasoning state which is the first one satisfies the termination condition, *or* none if every search branches fail.

The priorities of rules demonstrate which rules are better among admissible reasoners. The priority makes sense only locally, among all admissible reasoners on a reasoning state. The accumulation of priority values (i.e. the sum of the priority of all applied reasoners) of a reasoning state is meaningless and merely for debug-usage. Because it is a DFS, the first reached result is the optimal one w.r.t each search branches in a greedy sense. (the global maximum is senseless here because the priority accumulation is meaningless).

The sequent of the reasoning state is a Harrop Formula (HF), e.g.,

$$Antecedent1 \Longrightarrow Antecedent2 \Longrightarrow Conclusion,$$

where antecedents represent sub-goals that have to be reasoned *in order*.

The φ-LPR engine reasons antecedents in order, invoking the reasoners that match the pattern of the leading antecedent best (cf. Priority).

An antecedent can be augmented by conditions that can be utilized during the reasoning. It can also be universally quantified.

$$(\bigwedge x.\ \mathit{P1}\ x \Longrightarrow \mathit{P2}\ x \Longrightarrow \mathit{Conclusion\text{-}of\text{-}Antecedent1}\ x) \Longrightarrow \mathit{A2} \Longrightarrow C$$

A typically reasoner is to deduce the conclusion of the antecedent by applying an introduction rule like $\mathit{A11}\ x \Longrightarrow \mathit{A12}\ x \Longrightarrow \mathit{Conclusion\text{-}of\text{-}Antecedent1}\ x$, resulting in

$$(\bigwedge x.\ \mathit{P1}\ x \Longrightarrow \mathit{P2}\ x \Longrightarrow \mathit{A11}\ x) \Longrightarrow (\bigwedge x.\ \mathit{P11}\ x \Longrightarrow \mathit{P12}\ x \Longrightarrow \mathit{A12}\ x) \Longrightarrow \mathit{A2} \Longrightarrow C.$$

Then, the engine reasons the currently heading antecedent $(\bigwedge x.\ \mathit{P1}\ x \Longrightarrow \mathit{P2}\ x \Longrightarrow \mathit{A11}\ x)$ recursively. The antecedent list of a reasoning state resembles a calling stack of usual programs. From this perspective, the introduction rule of *Antecedent1* invokes two 'sub-routines' (or the reasoners of) *A11* and *A22*.

2 The Engine & The Concepts

The engine is implemented in `library/reasoner.ML`.

```
structure Phi_Reasoner = struct

(*Reasoning state*)
type context_state = Proof.context * thm
type name = term (* the name as a term is just for pretty printing
*)

val pattern_on_conclusion : term -> pattern
val pattern_on_condition  : term -> pattern

(*A reasoner is a quintuple*)
type reasoner = {
  name: name,
  pos: Position.T,
  pattern: pattern list,
  blacklist: pattern list,
  tactic: context_state -> context_state Seq.seq
}
```

```
type priority = int
val add : priority * reasoner -> Context.generic -> Context.generic
val del : name -> Context.generic -> Context.generic
val reason : context_state -> context_state option

val auto_level : int Config.T

exception Success of context_state
exception Global_Cut of context_state

...
end
```

Patterns The **pattern** and the **blacklist** stipulate the range in which a reasoner will be invoked. A reasoner is invoked iff the antecedent matches at least one pattern in the pattern list and none in the blacklist.

There are two kinds of patterns, that match on conclusion and that on condition, constructed by `pattern_on_conclusion` and `pattern_on_conclusion` respectively.

Prefix `var`. A schematic variable in a pattern can have name prefix `var_`. In this case, the variable only matches schematic variables.
Remark: It is important to write schematic variables in patterns explicitly. The engine does not convert any free variables to schematic variables implicitly.

Automatic Level by `auto_level` is a general configuration deciding whether the engine applies some aggressive tactics that may consume considerable time or never terminate.
There are 3 levels:

0 : the most safe, which may mean manual mode for some reasoner. It does not exclude non-termination or blocking when some tactics are necessary for the features. Method *simp* and *clarify* are acceptable on this level.

1 : relatively safe automation, where aggressive tactics are forbidden but non-termination is still possible. Method *auto* is forbidden in this level because it blocks too easily.

2 : the most powerful automation, where no limitation is imposed on automation strategies.

Priority The reasoning is a depth-first search and every reasoner is registered with a priority deciding the order of attempting the reasoners. Reasoners with higher priority are attempted first.

According to the priority of reasoners, reasoners fall into 3 sorts corresponding to different pruning optimization strategy.

1. When the priorities of the candidate reasoners on a certain reasoning state are all less than 1000, the reasoning works in the normal behavior where it attempts the highest candidate and once fails backtracks to the next candidate.

2. When the highest priority of the candidates $\geq$ 1000 and $<$ than 1000,000, this candidate becomes a *local cut*. The reasoning attempts only the local cut and if it fails, no other candidates will be attempted, but the backtrack is still propagated to the upper layer (of the search tree). Any presence of a candidate with priority $\geq$ 1000, causes the reasoning (at this point) is confident (in the sense that no alternative search branch will be attempted).

3. When the highest priority of the candidates $\geq$ 100,000, this candidate becomes a *global cut*, which forgets all the previous search history. No backtrack will be propagated to the past before the global cut so it improves the performance. Once the reasoning of the branch of the cut fails, the whole reasoning fails.

   Reasoners of priority $\geq$ 1000 are named *confident reasoners* and others are *submissive reasoners*.

   Remark: a local cut reasoner can throw `Global_Cut s` to trigger a global cut with the reasoning state `s`.

Termination The reasoning terminates when:

- Any reasoning state has no antecedent any more or all its designated leading antecedents are solved. This reasoning state is returned.
- Any reasoner throws `Success result`.
- All accessible search paths are traversed.

r*Success* is an antecedent that throws `Success`. Therefore it remarks the reasoning is succeeded. A typical usage of r*Success* is shown in the following sequent,

$$A1 \Longrightarrow A2 \Longrightarrow \mathrm{r}Success \Longrightarrow P \Longrightarrow Q$$

which expresses the reasoning succeeds after solving $A1$, $A2$, and it outputs result $P \Longrightarrow Q$.

Pure.prop P is helpful to protect remaining antecedents if you only want to reason the beginning several antecedents instead of all antecedents, e.g.,

$$Solve\text{-}A1 \Longrightarrow Pure.prop\ (Protect\text{-}A2 \Longrightarrow C)$$

Output The output reasoning state can be:

- The first traversed reasoning state that has no antecedent or all the designated leading antecedents are solved.
- The `result` threw out by `Success result`.

If none of the above are reached during a reasoning process, the process returns nothing (`None` or `Seq.empty`). The reasoning only outputs *milestone states* representing the problem is indeed solved partially instead of any unfinished intermediate reasoning state. Milestone states are explicitly annotated by user (e.g., by antecedent r*Success* or by setting the priority to 1000,000). Any other intermediate reasoning state is not considered a successfully finished state so that is not outputted.

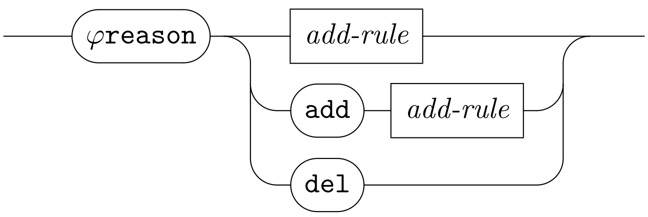
3 Provide User Reasoners & Apply the Engine

φ-LPR can be augmented by user reasoners. The system predefines a resolution based reasoner using introducing rules and elimination rules. Other arbitrary reasoners can also be built from tactics or ML code.

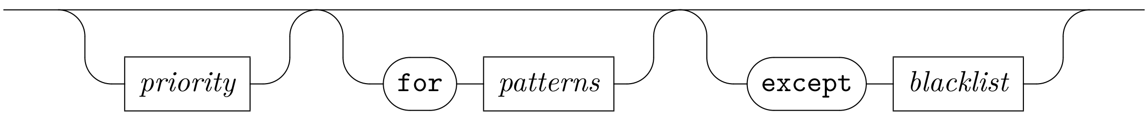
3.1 Reasoning by Rules

Attributes *φreason* is provided for introducing resolution rules.

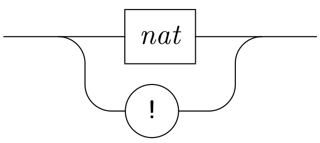
φreason : *attribute*



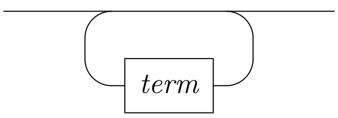
add-rule



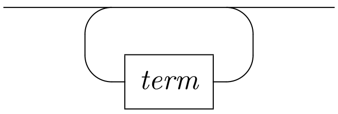
priority



patterns



blacklist



φreason `add` declares reasoning rules used in φ-LPR. *φreason* `del` removes the reasoning rule. If no keyword `add` or `del` is given, `add` is the default option.

The *patterns* and *blacklist* are that described in section 2. For introduction rules, the patterns and the blacklist match only the conclusion of the leading antecedent; for elimination rules, they match only the conditions of the leading antecedent.

Patterns can be omitted. For introduction rule, the default pattern is the conclusion of the rule; for elimination rule, the default is the first premise.

priority can be a natural number or, an exclamation mark denoting the priority of 1000,000, i.e., the minimal priority for a global cut. If the priority is not given explicitly, by default it is 100.

Remark: Rules of priority $\geq$ 1000 are named *confident rules* and others are *submissive rules.*

Remark: Attribute *φreason* can be used without any argument. `[[\xphi reason]]` denotes `[[\xphi reason add]]` exactly. However, the usage of empty arguments is not recommended due to technical reasons that in this case of empty argument the attribute cannot get the position of the associated reasoning rule, and this position is displayed in debug printing.

Example **declare** *conjI*[*φreason add*] *TrueI*[*φreason 1000*]

r**Feasible** Cut rules including local cut and global cut are those of priority $\geq$ 1000. A cut rule can have at most one special r*Require* antecedent at the leading position, which determines the condition of the rule to be applied, e.g. the following rule can be applied only if *A1* and *A2* are solvable.

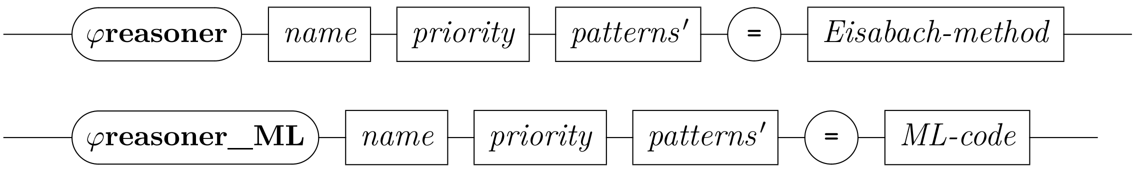
r*Require* (*A1* &&& *A2*) $\Longrightarrow$ *A3* $\Longrightarrow$ *C*

It provides a mechanism to constrain semantic conditions of applying the rule, whereas the pattern matches mentioned earlier are only able to check the syntactical conditions.

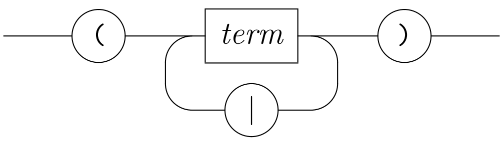
3.2 Reasoners by Isar Methods and ML code

There are two commands defining reasoners, respectively by Eisbach expression and by ML code.

$$
\begin{array}{rcl}
\varphi\textbf{reasoner} & : & \textit{local-theory} \rightarrow \textit{local-theory} \\
\varphi\textbf{reasoner-ML} & : & \textit{local-theory} \rightarrow \textit{local-theory}
\end{array}
$$



patterns′



φ**reasoner** defines a reasoner using an Eisabach expression. The Eisabach expression defines a proof method in Isabelle/Isar and this proof method is invoked on the leading antecedent as a sub-goal when *patterns′* match.

φ**reasoner-ML** defines a reasoner from ML code. The given code should be a ML function of type `context_state -> context_state Seq.seq`, i.e., a contextual tactic.

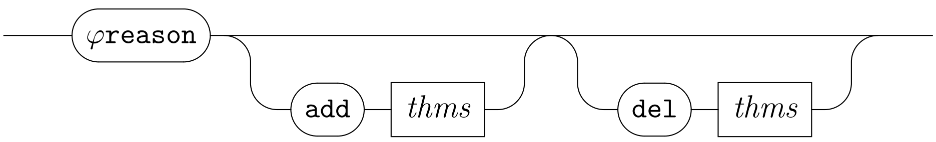
3.3 Apply the Engine

There are two ways to use the reasoning engine, from ML code by using `Phi_Reasoner.reason`, and as a proof method.

3.3.1 Proof Method

There are two commands defining reasoners, respectively by Eisbach expression and by ML code.

φ*reason* : *method*



φ*reason* `add` *a* `del` *b* applies φ-LPR on the proof state (which is a HHF sequent [**?**]). It means subgoals of the proof are regarded as antecedents and φ-LPR reasons them one by one in order.

Optional modifier `add` *a* adds introduction rules *a* temporarily with default patterns (the conclusion of the rule) and default priority (100). Modifier `del` *b* removes introductions rules *b* temporarily. We do not provide modifiers to alter elimination rules now.

4 Predefined Antecedents, Reasoners, and Rules

4.1 Auxiliary Structures

4.1.1 Isomorphic Atomize

The system *Object-Logic.atomize* and *Object-Logic.rulify* is not isomorphic in the sense that for any given rule *R*, *Object-Logic.rulify* (*Object-Logic.atomize* *R*) does not exactly equal *R*. The section gives a way addressing this issue.

ML-file ‹*library/iso-atomize.ML*›

definition ‹*pure-imp-embed* ≡ (⟶)›
definition *pure-all-embed* :: ‹(*'a* ⇒ *bool*) ⇒ *bool*› (**binder** ‹$\forall_{embed}$ › *10*)
— We give it a binder syntax to prevent eta-contraction which deprives names of quantifier variables
where ‹*pure-all-embed* ≡ (*All*)›
definition ‹*pure-conj-embed* ≡ (∧)›

definition ‹*pure-prop-embed* $x \equiv x$›

lemma [*iso-atomize-rules*, *symmetric*, *iso-rulify-rules*]:
‹$(P \Longrightarrow Q) \equiv$ *Trueprop* (*pure-imp-embed* P Q)›
unfolding *atomize-imp pure-imp-embed-def* **.**

lemma [*iso-atomize-rules*, *symmetric*, *iso-rulify-rules*]:
‹$(P$ &&& $Q) \equiv$ *Trueprop* (*pure-conj-embed* P Q)›
unfolding *atomize-conj pure-conj-embed-def* **.**

lemma [*iso-atomize-rules*, *symmetric*, *iso-rulify-rules*]:
‹$(\bigwedge x.\ P\ x) \equiv$ *Trueprop* (*pure-all-embed* $(\lambda x.\ P\ x)$)›
unfolding *atomize-all pure-all-embed-def* **.**

lemma [*iso-atomize-rules*, *symmetric*, *iso-rulify-rules*]:
‹*PROP Pure.prop* (*Trueprop* P) $\equiv$ *Trueprop* (*pure-prop-embed* P)›
unfolding *Pure.prop-def pure-prop-embed-def* **.**

4.1.2 Action

In the reasoning, antecedents of the same form may have different purposes, e.g., antecedent $P = ?Q$ may except a complete simplification or numeric calculation only or any other specific conversion. Of different purposes, antecedents are expected to be processed by different reasoners. To achieves this, because the engine selects reasoners by syntactic pattern, this section proposes a general structure tagging the purpose of antecedents.

The purpose is denoted by *action* type, which is an unspecified type because it serves only for syntactic purpose.

‹P @*action* A› tags antecedent P by the specific purpose denoted by A.

The type variable *'category* enables to classify actions by types and type classes. For example, some operation may be designed for any generic action *?act* :: (*?'ty*::*cls*) *action* that fall into class *cls*.

Comment: I am thinking this category type variable is a bad design because the indexing data structure (Net) we are using doesn't support type sort, causing this feature is actually not indexed at all, causing the reasoning here becomes searching one by one in linear time! Maybe classification should be done by some term-level structure. Let's think when have time!

definition *Action-Tag-embed* :: ‹*bool* ⇒ *action* ⇒ *bool*›
where ‹*Action-Tag-embed* P $A \equiv P$›

lemma [*iso-atomize-rules*, *symmetric*, *iso-rulify-rules*]:
‹*PROP Action-Tag* (*Trueprop P*) $A \equiv$ *Trueprop* (*Action-Tag-embed P A*)›
unfolding *Action-Tag-def Action-Tag-embed-def* **.**

lemma *Action-Tag-D*:
‹*P* @*action A* $\Longrightarrow$ *P*›
unfolding *Action-Tag-def* **.**

lemma *Conv-Action-Tag-I*:
‹$X = X$ @*action A*›
unfolding *Action-Tag-def* **..**

ML-file ‹*library/action-tag.ML*›

4.1.3 Mode

Modes are general annotations used in various antecedents, which may configure for the specific reasoning behavior among slight different options. The exact meaning of them depend on the specific antecedent using them. An example can be found in section 4.4.

type-synonym *mode* = *action*

We provide a serial of predefined modes, which may be commonly useful.

consts *default* :: *mode*
consts *MODE-SIMP* :: *mode* — relating to simplification
consts *MODE-COLLECT* :: *mode* — relating to collection
consts *MODE-AUTO* :: *mode* — something that will be triggered automatically

4.2 General Rules

Schematic variables are able to be instantiated (assigned) by reasoners. The instantiation of an schematic variable *?v* updates all the occurrences of *?v* in the remaining sequent, and this instantion can be seen as assigning results of the execution of the antecedent. For example,

$$1 + 2 = ?result \Longrightarrow Print\ ?result \Longrightarrow Done$$

the reasoning of antecedent $1 + 2 = ?result$ instantiates *?result* to *3*, and results in

$$Print\ 3 \Longrightarrow Done$$

If view the antecedent as a program (sub-routine), the schematic variables of the antecedent have a meaning of *output*, and we name them *output variables*.

The following *Try* antecedent is a such example.

4.2.1 Try

definition *Try* :: ‹*bool* ⇒ *bool* ⇒ *bool*› **where** ‹*Try success-or-fail P* = *P*›

The typical usage is ‹*Try ?success-or-fail P*›, where *P* should be an antecedent having some fallback reasoner (not given here), and *?success-or-fail* is an output variable representing whether the *P* is successfully deduced *without* using fallback.

A high priority (800) rule reasons ‹*Try True P*› normally and set the output variable *success-or-fail* to be true.

lemma [*φreason 800* **for** ‹*Try ?S ?P*›]:
‹ *P*
⟹ *Try True P*›
unfolding *Try-def* **.**

Users using ‹*Try True P*› should provide the fallback rule for their own *P*. It depends on the application scenario and there is not a general rule for fallback of course. The fallback rule may has the following form,

$$\textit{Fallback-of-P} \Longrightarrow \textit{Try False P}$$

4.2.2 Compact Representation of Antecedents

Meta-programming is feasible on φ-LPR. The reasoning of an antecedent may generate dynamically another antecedent, and assign it to an output variable of type *bool*.

When multiple antecedents are going to be generated, it is more efficient to contract them into one antecedent using conjunctions (e.g. $A1 \wedge A2 \wedge A3 \wedge \cdots$), so they can be represented by one output variable of type *bool*.

$(\wedge_r)$ and $(\forall_r)$ are used to contract antecedents and embed universally quantified variables respectively.

definition *Compact-Antecedent* :: ‹*bool* ⇒ *bool* ⇒ *bool*› (**infixr** $\wedge_{\mathrm{r}}$ *35*)
where [*iff*]: ‹*Compact-Antecedent* = (∧)›

definition *Compact-Forall* :: ‹(*'a* ⇒ *bool*) ⇒ *bool*› (**binder** $\forall_{\mathrm{r}}$ *10*)
where [*iff*]: ‹*Compact-Forall* = *All*›

Assertive rules are given to unfold the compression and reason the antecedents in order.

lemma [*φreason 1000*]:
‹$P \Longrightarrow Q \Longrightarrow P \wedge_{\mathrm{r}} Q$›
unfolding *Compact-Antecedent-def* **..**

lemma [*φreason 1000*]:
‹$(\bigwedge x.\ P\ x) \Longrightarrow \forall_{\mathrm{r}} x.\ P\ x$›
unfolding *Compact-Forall-def* **..**

declare *conjunctionI*[*φreason 1000*] — Meta-conjunction P &&& Q is also a compression.

4.2.3 Matches

Antecedent *Matches pattern term* asserts *pattern* matches *term*; *NO-MATCH pattern term* asserts *pattern* does not match *term*.

definition *Matches* :: ‹$'a \Rightarrow 'a \Rightarrow bool$› **where** ‹*Matches - - = True*›

lemma *Matches-I*: ‹*Matches pattern term*› **unfolding** *Matches-def* **..**

*φ***reasoner-ML** *Matches 2000* (‹*Matches ?pattern ?term*›) =
‹*fn* (*ctxt*, *sequent*) =>
let
val (***const***‹*Trueprop*› $ (*Const* (***const-name***‹*Matches*›,-) $ *pattern* $ *term*))
= *Thm.major-prem-of sequent*
in
if Pattern.matches (*Proof-Context.theory-of ctxt*) (*pattern*, *term*)
then Seq.single (*ctxt*, @{*thm Matches-I*} *RS sequent*)
else Seq.empty
end›

lemma *NO-MATCH-I*: *NO-MATCH A B* **unfolding** *NO-MATCH-def* **..**

*φ***reasoner-ML** *NO-MATCH 0* (*NO-MATCH ?A ?B*) = ‹
fn (*ctxt*,*th*) =>
let
val (***const***‹*Trueprop*› $ (*Const* (***const-name***‹*NO-MATCH*›, -) $ *a* $ *b*)) = *Thm.major-prem-of th*
in
if Pattern.matches (*Proof-Context.theory-of ctxt*) (*a*,*b*)
then Seq.empty
else Seq.single (*ctxt*, @{*thm NO-MATCH-I*} *RS th*)
end
›

4.2.4 Proof By Assumption

definition *By-Assumption* :: ‹*prop* ⇒ *prop*› **where** ‹*By-Assumption P* ≡ *P*›
definition *May-By-Assumption* :: ‹*prop* ⇒ *prop*› **where** ‹*May-By-Assumption P* ≡ *P*›

lemma *By-Assumption-I*: ‹*PROP P* ⟹ *PROP By-Assumption P*› **unfolding** *By-Assumption-def* **.**
lemma *May-By-Assumption-I*: ‹*PROP P* ⟹ *PROP May-By-Assumption P*› **unfolding** *May-By-Assumption-def* **.**

```
φreasoner-ML By-Assumption 1000 (‹PROP By-Assumption -›) = ‹fn (ctxt,sequent)
=>
   HEADGOAL (Tactic.assume-tac ctxt) (@{thm By-Assumption-I} RS sequent)
     |> Seq.map (pair ctxt)
›
```

```
φreasoner-ML May-By-Assumption 1000 (‹PROP May-By-Assumption -›) = ‹fn
(ctxt,sequent) =>
  let val sequent′ = @{thm May-By-Assumption-I} RS sequent
   in (HEADGOAL (Tactic.assume-tac ctxt) ORELSE Seq.single) sequent′
       |> Seq.map (pair ctxt)
  end
›
```

4.3 Cut

The cuts have been introduced in section 2.

Antecedent r*Cut* triggers a global cut.

definition r*Cut* :: *bool* **where** ‹r*Cut* = *True*›

lemma [*iff*, *φreason 1000000*]: ‹r*Cut*› **unfolding** r*Cut-def* **..**

Antecedent r*Success* terminates the reasoning successfully with the reasoning state as the result.

definition r*Success* :: *bool* **where** ‹r*Success* = *True*›
lemma r*Success-I*[*iff*]: ‹r*Success*› **unfolding** r*Success-def* **..**

```
φreasoner-ML rSuccess 10000 (‹rSuccess›) = ‹fn (ctxt,sequent) =>
  raise Phi-Reasoner.Success (ctxt, @{thm rSuccess-I} RS sequent)›
```

4.4 Proof Obligation & Guard of Rule

definition *Premise* :: *mode* ⇒ *bool* ⇒ *bool* **where** *Premise* - *x* = *x*

abbreviation *Normal-Premise* (premise - [*27*] *26*)
  where *Normal-Premise* ≡ *Premise default*
abbreviation *Simp-Premise* (simprem - [*27*] *26*)
  where *Simp-Premise* ≡ *Premise MODE-SIMP*
abbreviation *Proof-Obligation* (obligation - [*27*] *26*)
  where *Proof-Obligation* ≡ *Premise MODE-COLLECT*

Premise mode P represents an ordinary proposition has to be proved during the reasoning. There are different modes expressing different roles in the reasoning.

simprem *P* is a *guard* of a rule, which constrains that the rule is appliable only when *P* can be solved *automatically* during the reasoning. If *P* fails to be solved, even if it is actually valid, the rule will not be applied. Therefore, *P* has to be as simple as possible. The tactic used to solve *P* is *clarsimp*. A more powerful tactic like *auto* is not adoptable because the tactic must be safe and non-blocking commonly. A blocking search branch blocks the whole reasoning, which is not acceptable.

simprem *P* is not for proof obligations that are intended to be solved by users. It is more like 'controller or switch' of the rules, i.e. *guard*.

premise *P* represents a proof obligation. Proof obligations in reasoning rules should be represented by it.

obligation *Q* by contrast represents proof obligations *Q* that are ready to be solved by user (or by automatic tools).

The difference between obligation *Q* and premise *P* is subtle: In a reasoning process, many reasoning rules may be applied, which may generate many premise *P*. The engine tries to solve premise *P* automatically but if it fails the search branch would be stuck. Because the search has not been finished, it is bad to ask users' intervention to solve the goal because the search branch may high-likely fail later. It is *not ready* for user to solve *P* here, and suggestively *P* should be deferred to an ideal moment for user solving obligations. This is 'ideal moment' is obligation *Q*. If any obligation *Q* exists in the antecedents of the sequent, the engine contracts *P* into the latest obligation *Q*, e.g., from

$$\text{premise } P \Longrightarrow A1 \Longrightarrow \text{obligation } Q \Longrightarrow \text{obligation } Q' \Longrightarrow \cdots$$

it deduces

$$A1 \Longrightarrow \text{obligation } Q \wedge P \Longrightarrow \text{obligation } Q' \Longrightarrow \cdots$$

In short, obligation Q collects obligations generated during a reasoning process, and enables user to solve them at an idea moment.

A typical reasoning request (the initial reasoning state namely the argument of the reasoning process) is of the following form,

$$\mathit{Problem} \Longrightarrow \mathrm{r}\mathit{Success} \Longrightarrow \text{obligation } \mathit{True} \Longrightarrow \mathit{Conclusion}$$

The *True* represents empty collection or none obligation. If the reasoning succeeds, it returns sequent in form

$$\text{obligation } \mathit{True} \wedge \mathit{P1} \wedge \mathit{P2} \wedge \cdots \Longrightarrow \mathit{Conclusion}$$

where *P1*, *P2*, $\cdots$ are obligations generated by reasoning *Problem*. And then, user may solve the obligations manually or by automatic tools.

For antecedent obligation Q, if there is another obligation Q' in the remaining antecedents, the reasoner also defer Q to Q', just like obligation Q is a premise Q.

If no obligation Q' exists in the remaining antecedents, the reasoner of premise P and obligation Q raises an error aborting the whole reasoning, because the reasoning request is not configured correctly.

Semantically, obligation Q represents a proof obligation Q intended to be addressed by user. It can be deferred but the reasoner never attempts to solve obligation Q practically.

Nonetheless, we still provide tool for reasoning obligations automatically, albeit they have to be called separately with the reasoning engine. See `auto_obligation_solver` and `safer_obligation_solver` in `library/reasoners.ML`.

lemma *Premise-I*[*intro*!]: $P \Longrightarrow$ *Premise mode P* **unfolding** *Premise-def* **by** *simp*
lemma *Premise-D*: *Premise mode P* $\Longrightarrow P$ **unfolding** *Premise-def* **by** *simp*
lemma *Premise-E*[*elim*!]: *Premise mode P* $\Longrightarrow (P \Longrightarrow C) \Longrightarrow C$ **unfolding** *Premise-def* **by** *simp*

4.4.1 Implementation of the reasoners

lemma *Premise-True*[*φreason 5000*]: *Premise mode True* **unfolding** *Premise-def* **..**

lemma [*φreason 5000*]:
Premise mode P

$\Longrightarrow$ *Premise mode* (*Premise any-mode P*)
  unfolding *Premise-def* **.**

lemma *Premise-refl*[*φreason 2000* **for** ‹*Premise ?mode* (*?x* = *?x*)›
‹*Premise ?mode* (*?x* = *?var-x*)›
‹*Premise ?mode* (*?var-x* = *?x*)›]:
  Premise mode ($x = x$)
  unfolding *Premise-def* **..**

lemma *contract-obligations*:
  (*Premise mode P* $\Longrightarrow$ obligation $Q \Longrightarrow$ *PROP C*) $\equiv$ (obligation $P \wedge Q \Longrightarrow$ *PROP C*)
  unfolding *Premise-def* **by** *rule simp+*

lemma *contract-premise-true*:
  (*True* $\Longrightarrow$ *Premise mode B*) $\equiv$ *Trueprop* (*Premise mode B*)
  by *simp*

lemma *contract-premise-imp*:
  ($A \Longrightarrow$ *Premise mode B*) $\equiv$ *Trueprop* (*Premise mode* ($A \longrightarrow B$))
  unfolding *Premise-def atomize-imp* **.**

lemma *contract-premise-all*:
  ($\bigwedge x$. *Premise mode* ($P\ x$)) $\equiv$ *Trueprop* (*Premise mode* ($\forall x.\ P\ x$))
  unfolding *Premise-def atomize-all* **.**

declare [[*ML-debugger* = *true*]]

ML ‹
structure Useful-Thms = Named-Thms (
  val name = ***binding***‹*useful*›
  val description = theorems to be inserted in the automatic proving, \
        having the same effect of using the @{*command using*} *command.*
)
›

setup ‹*Useful-Thms.setup*›

ML-file ‹*library/PLPR-Syntax.ML*›
ML-file *library/reasoners.ML*

*φ***reasoner-ML** *Normal-Premise 10* (‹premise *?P*› | ‹obligation *?P*›)

= ‹*Phi-Reasoners.wrap Phi-Reasoners.defer-obligation-tac*›

4.5 Reasoning Frame

definition ‹r*BEGIN* $\longleftrightarrow$ *True*›
definition ‹r*END* $\longleftrightarrow$ *True*›

Antecedents r*BEGIN* and r*END* conform a nested reasoning scope resembling a subroutine for specific reasoning tasks or problems.

$$\ldots \Longrightarrow \mathrm{r}\mathit{BEGIN} \Longrightarrow \mathit{Nested} \Longrightarrow \mathit{Reasoning} \Longrightarrow \mathrm{r}\mathit{END} \Longrightarrow \ldots$$

The scoped antecedents should be regarded as a *unit antecedent* invoking a nested φ-LPR reasoning process and returning *only* the first reached solution (just as the behaviour of φ-LPR engine). During backtracking, search branches before the unit will be backtracked but sub-optimal solutions of the unit are not backtracked. In addition, cut is confined among the search paths in the scope as a unit. Because of the cut and the reduced backtrack behavior, the performance is improved.

Sometimes a cut is admissible (green) as an expected behavior among several rules and reasoners which constitute a loosely-gathered module for a specific problem. However the cut is still not safe to be used because an external rule using the reasoning module may demand the behavior of backtracking but the cut inside the module prevents backtracks in the external rule. In this case, the reasoning scope is helpful to wrap the loosely-gathered module to be confined by closing side effects like cuts.

Specifically, any search path that reaches r*BEGIN* opens a new *frame* namely a space of search paths. The sub-searches continuing the path and before reaching the paired r*END* are in this frame. As φ-LPR works in BFS, a frame can contain another frame just if the search in the frame encounters another r*BEGIN*.

$$\ldots \Longrightarrow \mathrm{r}\mathit{BEGIN} \Longrightarrow A_1 \Longrightarrow \mathrm{r}\mathit{BEGIN} \Longrightarrow A_2 \Longrightarrow \mathrm{r}\mathit{END} \Longrightarrow A_3 \Longrightarrow \mathrm{r}\mathit{END} \Longrightarrow \ldots$$

Once any search path encounters a r*END*, the innermost frame is closed and the sequent of the search path is returned with dropping all other branches in the frame. The mechanism checks whether all r*BEGIN* and r*END* are paired.

Any global cut cuts all and only all search branches in the innermost frame to which the cut belongs. r*Success* is prohibited in the nested scope because we do not know how to process the remain antecedents after the r*Success* and how to return them into the outer scope.

definition r*Call* :: ‹*prop* ⇒ *prop*› (r*CALL* - [*3*] *2*)
  where ‹r*Call* *P* ≡ *PROP* *P*›
  — Call the antecedent *P* in a frame

lemma r*BEGIN-I*: ‹r*BEGIN*› **unfolding** r*BEGIN-def* **..**
lemma r*END-I*: ‹r*END*› **unfolding** r*END-def* **..**
lemma r*Call-I*: ‹*PROP* *P* ⟹ r*CALL* *PROP* *P*› **unfolding** r*Call-def* **.**

ML-file ‹*library/nested.ML*›

*φ***reasoner-ML** r*BEGIN* *1000* (‹r*BEGIN*›) = ‹*PLPR-Nested-Reasoning.enter-scope*›
*φ***reasoner-ML** r*END* *1000* (‹r*END*›) = ‹*PLPR-Nested-Reasoning.exit-scope*›
*φ***reasoner-ML** r*Call* *1000* (‹*PROP* r*Call* -›) = ‹*PLPR-Nested-Reasoning.call*›

definition r*Call-embed* :: ‹*bool* ⇒ *bool*› **where** ‹r*Call-embed* *P* ≡ *P*›

lemma [*iso-atomize-rules*, *symmetric*, *iso-rulify-rules*]:
  ‹r*Call* (*Trueprop* *P*) ≡ *Trueprop* (r*Call-embed* *P*)›
  unfolding r*Call-def* r*Call-embed-def* **.**

4.6 Pruning

At a reasoning state A, multiple search branches may be emitted parallel to find a solution of the antecedent. A branch may find the solution while other branches from A still remain in the search history. Then the reasoning in DFS manner keeps to solve next antecedent B and we assume B fails. The reasoning then backtrack, and redo the search of A on remaining branches of A. It is not reasonable because the reasoning is redoing a solved problem on A. To address this, a solution is to prune branches of A after A succeeds. In this section we introduce *subgoal* mechanism achieving the pruning. Each antecedent A is tagged with a goal context G, as ‹A **@GOAL** G›. A reasoning rule may check that the goal G has not been solved before doing any substantial computation, e.g.,

$$\mathit{CHK\text{-}SUBGOAL}\ G \Longrightarrow \mathit{Computation} \Longrightarrow (\mathit{Ant}\ \textbf{@GOAL}\ G)$$

Antecedent *CHK-SUBGOAL G* succeeds only when the goal G is not marked solved, *or*, the current search branch is the thread that marked G solved previously. When a rule succeeds, the rule may mark the goal G solved to prune other branches that check G.

$$\mathit{Computation} \Longrightarrow \mathit{SOLVE\text{-}SUBGOAL}\ G \Longrightarrow (\mathit{Ant}\ \textbf{@GOAL}\ G)$$

If a goal G has been marked solved, any other antecedent *SOLVE-SUBGOAL* G marking G again, will fail, unless the current search branch is the thread that marked G solved previously.

A subgoal is represented by an unspecified type which only has a syntactic effect in the reasoning.

typedecl *subgoal*

consts *subgoal-context* :: ‹*subgoal* ⇒ *action*›

abbreviation *GOAL-CTXT* :: *prop* ⇒ *subgoal* ⇒ *prop* (- **@GOAL** - [*2*,*1000*] *2*)
  where (*PROP P* **@GOAL** *G*) ≡ (*PROP P* @*action subgoal-context G*)

definition *CHK-SUBGOAL* :: *subgoal* ⇒ *bool* — Check whether the goal is solved
  where *CHK-SUBGOAL X* ⟷ *True*
definition *SOLVE-SUBGOAL* :: *subgoal* ⇒ *bool*
  where *SOLVE-SUBGOAL X* ⟷ *True*

Subgoals are hierarchical, having the unique top-most goal named ‹*TOP-GOAL*›. New goal contexts are obtained by antecedent ‹*SUBGOAL G ?G′*› which assigns a new subgoal under an unsolved G to output variable *?G′*. The reasoning raises an error if *?G′* is not a schematic variable.

‹*SOLVE-SUBGOAL G*› marks the goal G and all its subgoals solved. The *TOP-GOAL* can never be solved.

consts *TOP-GOAL* :: *subgoal*

definition *SUBGOAL* :: *subgoal* ⇒ *subgoal* ⇒ *bool* **where** *SUBGOAL ROOT NEW-GOAL* = *True*

4.6.1 Implementation of the Subgoal Reasoners

lemma *SUBGOAL-I*[*iff*]: *SUBGOAL ROOT NEWGOAL* **unfolding** *SUBGOAL-def*
..
lemma *CHK-SUBGOAL-I*[*iff*]: *CHK-SUBGOAL X* **unfolding** *CHK-SUBGOAL-def*
..
lemma *SOLVE-SUBGOAL-I*[*iff*]: *SOLVE-SUBGOAL X* **unfolding** *SOLVE-SUBGOAL-def*
..

ML-file ‹*library/Subgoal-Env.ML*›

*φ***reasoner-ML** *SUBGOAL 2000* (‹*SUBGOAL ?ROOT ?NEWGOAL*›) = ‹*Subgoal-Env.subgoal*›
*φ***reasoner-ML** *CHK-SUBGOAL 2000* (‹*CHK-SUBGOAL ?GOAL*›) = ‹*Subgoal-Env.chk-subgoal*›

*φ***reasoner-ML** *SOLVE-SUBGOAL 9900* (‹*SOLVE-SUBGOAL ?GOAL*›) = ‹*Subgoal-Env.solve-subgoal*›

lemma [*φreason 800* **for** ‹ *Try ?S ?P* **@GOAL** *?G*›]:
‹ *P* **@GOAL** *G*
$\Longrightarrow$ *Try True P* **@GOAL** *G*›
unfolding *Try-def* **.**

4.7 Branch

$A \mid\mid\mid B$ is an antecedent way to encode search branch. Compared with the ordinary approach using multiple submissive rules, short-cut is featured by using subgoal. It tries each antecedent from left to right until the first success of solving an antecedent, and none of the remains are attempted.

definition *Branch* :: ‹*prop* ⇒ *prop* ⇒ *prop*› (**infixr** ||| *3*)
where ‹*Branch A B* ≡ (⋀*C*. (*PROP A* $\Longrightarrow$ *C*) $\Longrightarrow$ (*PROP B* $\Longrightarrow$ *C*) $\Longrightarrow$ *C*)›

definition *Branch-embed* :: ‹*bool* ⇒ *bool* ⇒ *bool*›
where ‹*Branch-embed A B* ≡ *A* ∨ *B*›

lemma *atomize-Branch*:
‹*Branch* (*Trueprop A*) (*Trueprop B*) ≡ *Trueprop* (*A* ∨ *B*)›
unfolding *Branch-def or-def atomize-eq atomize-imp atomize-all* **.**

lemma [*iso-atomize-rules*, *symmetric*, *iso-rulify-rules*]:
‹*Branch* (*Trueprop A*) (*Trueprop B*) ≡ *Trueprop* (*Branch-embed A B*)›
unfolding *Branch-embed-def atomize-Branch* **.**

4.7.1 Implementation

lemma *Branch-L*:
‹ *PROP A*
$\Longrightarrow$ *PROP A* ||| *PROP B*›
unfolding *Action-Tag-def Branch-def*
proof −
assume *A*: ‹*PROP A*›
show ‹(⋀*C*. (*PROP A* $\Longrightarrow$ *C*) $\Longrightarrow$ (*PROP B* $\Longrightarrow$ *C*) $\Longrightarrow$ *C*)› **proof** −
fix *C* :: *bool*
assume *A′*: ‹*PROP A* $\Longrightarrow$ *C*›
show ‹*C*› **using** *A′*[*OF A*] **.**
qed
qed

lemma *Branch-R*:

‹ *PROP B*
$\Longrightarrow$ *PROP A* ||| *PROP B*›
unfolding *Action-Tag-def Branch-def*
proof –
assume *B*: ‹*PROP B*›
show ‹($\bigwedge C$. (*PROP A* $\Longrightarrow C$) $\Longrightarrow$ (*PROP B* $\Longrightarrow C$) $\Longrightarrow C$)› **proof** –
fix *C* :: *bool*
assume B': ‹*PROP B* $\Longrightarrow C$›
show ‹*C*› **using** B'[*OF B*] **.**
qed
qed

declare [[*φreason 1000 Branch-L Branch-R* **for** ‹*PROP ?A* ||| *PROP ?B*›]]

4.8 Simplification & Rewrite

‹simplify[*mode*] *?result* : *term*› is generic antecedent for simplifying *term* in different *mode*. The *?result* should be an output variable for the result of the simplification.

We implement a *default* mode where the system simple-set is used to simplify *term*. Users may configure their mode and their reasoner using different simple-set.

definition *Simplify* :: *mode* $\Rightarrow$ *'a* $\Rightarrow$ *'a* $\Rightarrow$ *bool* (simplify[-] - :/ - [*10*,*1000*,*10*] *9*)
where *Simplify setting result origin* $\longleftrightarrow$ *result* = *origin*

definition *Do-Simplificatin* :: ‹*'a* $\Rightarrow$ *'a* $\Rightarrow$ *prop*›
where ‹*Do-Simplificatin result origin* $\equiv$ (*result* $\equiv$ *origin*)›

lemma [*cong*]: $A \equiv A' \Longrightarrow$ *Simplify s x A* $\equiv$ *Simplify s x* A' **by** *simp*

lemma *Simplify-D*: ‹*Simplify m A B* $\Longrightarrow A = B$› **unfolding** *Simplify-def* **.**
lemma *Simplify-I*: ‹$A = B \Longrightarrow$ *Simplify m A B*› **unfolding** *Simplify-def* **.**

lemma *Do-Simplification*:
‹*PROP Do-Simplificatin A B* $\Longrightarrow$ *Simplify s A B*›
unfolding *Do-Simplificatin-def Simplify-def atomize-eq* **.**

lemma *End-Simplification* : ‹*PROP Do-Simplificatin A A*› **unfolding** *Do-Simplificatin-def*
.
lemma *End-Simplification'*: ‹premise $A = B \Longrightarrow$ *PROP Do-Simplificatin A B*›
unfolding *Do-Simplificatin-def Premise-def atomize-eq* **.**

ML-file ‹*library/simplifier.ML*›

hide-fact *End-Simplification′ End-Simplification Do-Simplification*

4.8.1 Default Simplifier

abbreviation *Default-Simplify* :: $'a \Rightarrow 'a \Rightarrow bool$ (simplify - : - [*1000*,*10*] *9*)
where *Default-Simplify* ≡ *Simplify default*

φ**reasoner-ML** *Default-Simplify 1000* (‹*Default-Simplify ?X′ ?X*›)
= ‹*PLPR-Simplifier.simplifier NONE I*›

φ**reasoner-ML** *Simp-Premise 10* (‹simprem *?P*›)
= ‹*PLPR-Simplifier.simplifier NONE I*›

4.9 Optimal Solution

φ-LPR is priority-driven DFS searching the first reached solution which may not be the optimal one for certain measure. The section gives a way to find out the solution of the minimum cost among a given set of candidates.

definition *Optimum-Solution* :: ‹*prop* ⇒ *prop*› **where** [*iff*]: ‹*Optimum-Solution P* ≡ *P*›
definition [*iff*]: ‹*Begin-Optimum-Solution* ⟷ *True*›
definition [*iff*]: ‹*End-Optimum-Solution* ⟷ *True*›

Each individual invocation of *Optimum-Solution P* invokes an individual instance of the optimal solution reasoning. The reasoning of P is proceeded exhaustively meaning exploring all backtracks except local cuts.

Candidates The candidates are all search branches diverged from the antecedents marked by

For the antecedents marked by r*Choice*, the mechanism traverses exhaustively all combinations of their (direct) solvers, but for other not marked antecedents, the strategy is not changed and is as greedy as the usual behavior — returning the first-reached solution and discarding the others.
As an example, in *Begin-Optimum-Solution* ⟹ r*Choice* $A \Longrightarrow B \Longrightarrow$ r*Choice* $C \Longrightarrow$ *End-Optimum-Solution* ⟹ . . . , assuming both A,B,C have 2 solvers A_1,A_2,B_1,B_2,C_1,C_2 and assuming B_1 have higher priority than B_2 and can success, the mechanism traverses 4 combination of the solvers A_1,C_1, A_1,C_2, A_2,C_1, A_2,C_2, i.e., only exhaustively on r*Choice*-marked antecedents but still greedy on others.

Note, even marked by r*Choice*, local cuts are still valid and cuts search branches. Global cut is disabled during the whole reasoning because it kills other search branches. r*Success* is available and the mechanism ensures it is always the optimal one invokes the r*Success*.

Cost The cost is measured by reports from the following antecedents inserted in the user rules.

definition *Incremental-Cost* :: ‹*int* ⇒ *bool*› **where** [*iff*]: ‹*Incremental-Cost* - = *True*›
definition *Threshold-Cost* :: ‹*int* ⇒ *bool*› (threshold) **where** [*iff*]: ‹*Threshold-Cost* - = *True*›

The final cost of a reasoning process is the sum of all the reported *Incremental-Cost* or the maximum *Threshold-Cost*, the one which is larger.

If the cost of two branches are the same, the first reached one is considered better.

4.9.1 Implementation

definition *Optimum-Solution-embed* :: ‹*bool* ⇒ *bool*› **where** ‹*Optimum-Solution-embed* *P* ≡ *P*›

lemma [*iso-atomize-rules*, *symmetric*, *iso-rulify-rules*]:
‹*Optimum-Solution* (*Trueprop* *P*) ≡ *Trueprop* (*Optimum-Solution-embed* *P*)›
unfolding *Optimum-Solution-embed-def* *Optimum-Solution-def* **.**

lemma *Incremental-Cost-I*: ‹*Incremental-Cost* *X*› **unfolding** *Incremental-Cost-def* **..**

lemma *Threshold-Cost-I*: ‹*Threshold-Cost* *X*› **unfolding** *Threshold-Cost-def* **..**

lemma *Begin-Optimum-Solution-I*: ‹*Begin-Optimum-Solution*› **unfolding** *Begin-Optimum-Solution-def* **..**
lemma *End-Optimum-Solution-I*: ‹*End-Optimum-Solution*› **unfolding** *End-Optimum-Solution-def* **..**

lemma *Do-Optimum-Solution*:
‹ *PROP X*
⟹ *End-Optimum-Solution*
⟹ *PROP Optimum-Solution X*›
unfolding *Optimum-Solution-def* **.**

ML-file-debug ‹*library/optimum-solution.ML*›

```
φreasoner-ML Incremental-Cost 1000 (‹Incremental-Cost -›) = ‹fn (ctxt,sequent)
=> Seq.make (fn () =>
  let val - $ (- $ N) = Thm.major-prem-of sequent
      val (-, n) = HOLogic.dest-number N
      val sequent′ = @{thm Incremental-Cost-I} RS sequent
   in Seq.pull (PLPR-Optimum-Solution.report-cost (n,0) (ctxt,sequent′))
   end
)›
```

```
φreasoner-ML Threshold-Cost 1000 (‹Threshold-Cost -›) = ‹fn (ctxt,sequent)
=> Seq.make (fn () =>
  let val - $ (- $ N) = Thm.major-prem-of sequent
      val (-, n) = HOLogic.dest-number N
      val sequent′ = @{thm Threshold-Cost-I} RS sequent
   in Seq.pull (PLPR-Optimum-Solution.report-cost (0,n) (ctxt,sequent′))
  end
)›
```

```
φreasoner-ML Optimum-Solution 1000 (‹PROP Optimum-Solution -›) = ‹
   apsnd (fn th => @{thm Do-Optimum-Solution} RS th)
#> PLPR-Optimum-Solution.start
›
```

```
φreasoner-ML Begin-Optimum-Solution 1000 (‹Begin-Optimum-Solution›) = ‹
   apsnd (fn th => @{thm Begin-Optimum-Solution-I} RS th)
#> PLPR-Optimum-Solution.start
›
```

```
φreasoner-ML End-Optimum-Solution 1000 (‹End-Optimum-Solution›) = ‹
   apsnd (fn th => @{thm End-Optimum-Solution-I} RS th)
#> PLPR-Optimum-Solution.finish
›
```

4.9.2 Derivations

definition *Optimum-Among* :: ‹*prop* ⇒ *prop*› **where** ‹*Optimum-Among Candidates* ≡ *Candidates*›
— We leave it as a syntax merely

definition *Optimum-Among-embed* :: ‹*bool* ⇒ *bool*› **where** ‹*Optimum-Among-embed X* ≡ *X*›

lemma [*iso-atomize-rules*, *symmetric*, *iso-rulify-rules*]:
‹*Optimum-Among* (*Trueprop P*) ≡ *Trueprop* (*Optimum-Among-embed P*)›
unfolding *Optimum-Among-embed-def Optimum-Among-def* **.**

4.10 Environment Variables

definition *Push-Envir-Var* :: ‹′*name* ⇒ ′*a*::{} ⇒ *bool*›
where ‹*Push-Envir-Var Name Val* ⟷ *True*›
definition *Pop-Envir-Var* :: ‹′*name* ⇒ *bool*› **where** ‹*Pop-Envir-Var Name* ⟷ *True*›
definition *Get-Envir-Var* :: ‹′*name* ⇒ ′*a*::{} ⇒ *bool*›
where ‹*Get-Envir-Var Name Return* ⟷ *True*›
definition *Get-Envir-Var′* :: ‹′*name* ⇒ ′*a*::{} ⇒ ′*a* ⇒ *bool*›
where ‹*Get-Envir-Var′ Name Default Return* ⟷ *True*›

4.10.1 Implementation

ML-file ‹*library/envir-var.ML*›

lemma *Push-Envir-Var-I*: ‹*Push-Envir-Var N V*› **unfolding** *Push-Envir-Var-def* **..**
lemma *Pop-Envir-Var-I*: ‹*Pop-Envir-Var N*› **unfolding** *Pop-Envir-Var-def* **..**
lemma *Get-Envir-Var-I* : ‹*Get-Envir-Var N V*› **for** *V* :: ‹′*v*::{}› **unfolding** *Get-Envir-Var-def* **..**
lemma *Get-Envir-Var′-I*: ‹*Get-Envir-Var′ N D V*› **for** *V* :: ‹′*v*::{}› **unfolding** *Get-Envir-Var′-def* **..**

*φ***reasoner-ML** *Push-Envir-Var 1000* (‹*Push-Envir-Var - -*›) = ‹*fn* (*ctxt*,*sequent*) => *Seq.make* (*fn* () =>
let val - $ (- $ *N* $ *V*) = *Thm.major-prem-of sequent*
val - = *if maxidx-of-term V* <> ~*1*
then warning PLPR Envir Var: *The value to be assigned has schematic variables* \
which will not be retained!
else ()
in SOME ((*PLPR-Env.push* (*PLPR-Env.name-of N*) *V ctxt*,
@{*thm Push-Envir-Var-I*} *RS sequent*),
Seq.empty) *end*
)›

*φ***reasoner-ML** *Pop-Envir-Var 1000* (‹*Pop-Envir-Var -*›) = ‹*fn* (*ctxt*,*sequent*) => *Seq.make* (*fn* () =>
let val - $ (- $ *N*) = *Thm.major-prem-of sequent*

in SOME ((*PLPR-Env.pop* (*PLPR-Env.name-of N*) *ctxt*, @{*thm Pop-Envir-Var-I*} *RS sequent*),
Seq.empty) *end*
)›

*φ***reasoner-ML** *Get-Envir-Var 1000* (‹*Get-Envir-Var - -*›) = ‹*fn* (*ctxt*,*sequent*) => *Seq.make* (*fn* () =>
let val - $ (- $ *N* $ -) = *Thm.major-prem-of sequent*
val idx = *Thm.maxidx-of sequent* + *1*
in case PLPR-Env.get (*PLPR-Env.name-of N*) *ctxt*
of NONE => *Phi-Reasoner.error*
(*No enviromental variable* ^ *PLPR-Env.name-of N* ^ *is set*)
| *SOME V′* =>
let val V = *Thm.incr-indexes-cterm idx* (*Thm.cterm-of ctxt V′*)
in SOME ((*ctxt*, (@{*thm Get-Envir-Var-I*}
|> *Thm.incr-indexes idx*
|> *Thm.instantiate* (*TVars.make* [(((′*v*,*idx*),[]), *Thm.ctyp-of-cterm V*)],
Vars.make [(((*V*, *idx*),*Thm.typ-of-cterm V*), *V*)])
) *RS sequent*),
Seq.empty)
end
end
)›

*φ***reasoner-ML** *Get-Envir-Var′ 1000* (‹*Get-Envir-Var′ - - -*›) = ‹*fn* (*ctxt*,*sequent*) => *Seq.make* (*fn* () =>
let val - $ (- $ *N* $ *D* $ -) = *Thm.major-prem-of sequent*
val idx = *Thm.maxidx-of sequent* + *1*
val V = *Thm.cterm-of ctxt* (*case PLPR-Env.get* (*PLPR-Env.name-of N*) *ctxt*
of SOME V => *V* | *NONE* => *D*)
|> *Thm.incr-indexes-cterm idx*
in SOME ((*ctxt*, (@{*thm Get-Envir-Var′-I*}
|> *Thm.incr-indexes idx*
|> *Thm.instantiate* (*TVars.make* [(((′*v*,*idx*),[]), *Thm.ctyp-of-cterm V*)],
Vars.make [(((*V*, *idx*),*Thm.typ-of-cterm V*), *V*)])
) *RS sequent*),
Seq.empty)
end
)›

4.11 Recursion Guard

definition r*Recursion-Guard* :: ‹*'a*::{} ⇒ *prop* ⇒ *prop*› (r*RECURSION'-GUARD'*(-')/ - [*2*,*2*] *2*)
  where [*iff*]: ‹(r*RECURSION-GUARD*(*X*) (*PROP P*)) ≡ *PROP P*›

r*RECURSION-GUARD*(*X*) *PROP P* annotates the reasoning of *P* is about goal *X*. It remembers *X* and once in the following reasoning the same goal *X* occurs again, it aborts the search branch because an infinite recursion happens.

definition r*Recursion-Guard-embed* :: ‹*'a*::{} ⇒ *bool* ⇒ *bool*›
  where ‹r*Recursion-Guard-embed* - *P* ≡ *P*›

lemma [*iso-atomize-rules*, *symmetric*, *iso-rulify-rules*]:
  ‹r*Recursion-Guard X* (*Trueprop P*) ≡ *Trueprop* (r*Recursion-Guard-embed X P*)›
  unfolding r*Recursion-Guard-embed-def* r*Recursion-Guard-def* **.**

4.11.1 Implementation

definition r*Recursion-Residue* :: ‹*'a*::{} ⇒ *bool*›
  where ‹r*Recursion-Residue* - ≡ *True*›

lemma *Do*-r*Recursion-Guard*:
  ‹ *PROP P*
⟹ r*Recursion-Residue X*
⟹ r*RECURSION-GUARD*(*X*) (*PROP P*) ›
  unfolding r*Recursion-Guard-def* **.**

lemma [*φreason 1000*]:
  ‹r*Recursion-Residue X*›
  unfolding r*Recursion-Residue-def* **..**

ML-file ‹*library/recursion-guard.ML*›

*φ***reasoner-ML** r*Recursion-Guard 1000* (‹r*RECURSION-GUARD*(*?X*) (*PROP ?P*)›) = ‹*PLPR-Recursion-Guard.reason*›

hide-fact *Do*-r*Recursion-Guard*

end