Facets of the *PIE* Environment for Proving, Interpolating and Eliminating on the Basis of First-Order Logic

Christoph Wernhard

Berlin, Germany

Abstract. *PIE* is a Prolog-embedded environment for automated reasoning on the basis of first-order logic. Its main focus is on formulas, as constituents of complex formalizations that are structured through formula macros, and as outputs of reasoning tasks such as second-order quantifier elimination and Craig interpolation. It supports a workflow based on documents that intersperse macro definitions, invocations of reasoners, and LaTeX-formatted natural language text. Starting from various examples, the paper discusses features and application possibilities of *PIE* along with current limitations and issues for future research.

1 Introduction

First-order logic is used widely and in many roles in philosophy, mathematics, and artificial intelligence as well as other branches of computer science. Many practically successful reasoning approaches can be viewed as derived from reasoning in first-order logic, for example, SAT solving, logic programming, database query processing and reasoning in description logics. The overall aim of the *PIE* environment is to support the *practical mechanized reasoning in first-order logic*. Approaching this aim consequently leads from first-order theorem proving in the strict sense to tasks that *compute first-order formulas*, in particular second-order quantifier elimination and Craig interpolation, whose integrated support characterizes *PIE*. The system is written and embedded in *SWI-Prolog* [58] and provides, essentially as a library of Prolog predicates, a number of functionalities:

- Support for a Prolog-readable syntax of first-order logic formulas.
- Formula pretty-printing in Prolog syntax and in LaTeX.
- A versatile formula macro processor.
- Support for processing documents that intersperse formula macro definitions, reasoner invocations and LaTeX-formatted natural language text.
- Interfaces to external first-order and propositional reasoners.
- A built-in Prolog-based first-order theorem prover.
- Implemented reasoning techniques that compute formulas:
  - Second-order quantifier elimination on the basis of first-order logic.
  - Computation of first-order Craig interpolants.

- Formula conversions for use in preprocessing, inprocessing and output presentation.

The system is available as free software from its homepage

`http://cs.christophwernhard.com/pie`.

The distribution includes several example documents whose source files as well as rendered LaTeX presentations can also be accessed directly from the system Web page. *Inspecting Gödel's Ontological Proof* is there an advanced application, where the interplay of elimination and modal axioms is applied in several contexts. The system was first presented at the 2016 workshop *Practical Aspects of Automated Reasoning* [55]. Here we show various application possibilities, features and also issues for further research that become apparent with the system by starting from a number of examples. The paper is itself written as a *PIE* document and thus includes fragments generated by *PIE* and the included or integrated reasoners.

The rest of this paper is structured as follows: After introducing in Sect. 2 the document-oriented workflow supported by *PIE*, we show in Sect. 3 how it applies to the invocation of second-order quantifier elimination in the system. Section 4 provides an application example of elimination, a certain form of abduction, which is shown together with basic features of the *PIE* macro system. We proceed in Sect. 5 to outline how systems for theorem proving in the strict sense are embedded into *PIE*. In Sect. 6 the computation of circumscription is discussed as another example of second-order quantifier elimination with *PIE*, along with further features of the macro system and the general issue of finding good presentations of computed formulas that are essentially just characterized semantically. Section 7 sketches a further application of second-order quantifier elimination: a potential way of logic programming with second-order formulas as used for theoretical considerations in descriptive complexity. Further features of *PIE* are summarized in Sect. 9, and Sect. 10 concludes the paper.

Related work is discussed in the respective contexts. The bibliography is somewhat extensive, reflecting that the system relates to methods as well as implementation and application aspects in a number of areas, including first-order theorem proving, Craig interpolation, second-order quantifier elimination and knowledge representation.

2 *PIE* Documents

The main way to interact with *PIE* is by developing or modifying a *PIE document*, a file that intersperses definitions of formula macros, specifications of reasoning tasks, and LaTeX-formatted natural language text in the fashion of literate programming [28]. Such a document can be *loaded* into the Prolog environment like a source code file. Reasoner invocations, where the defined macros are available, can then be submitted as inputs on the Prolog console. The document can also be *processed*, which results in a generated LaTeX document: Macro definitions are pretty-printed in LaTeX, specified reasoner invocations are executed

and a pretty-printed LaTeX result presentation is inserted, and LaTeX fragments are inserted directly. The generated LaTeX document can then be displayed in PDF format.

Aside of indentation, the LaTeX pretty-printer can apply certain symbol conversions to subscripted or primed symbols. Also a compact syntax where parentheses to separate arguments from functors and commas between arguments are omitted is available as an option for both Prolog and LaTeX forms.

PIE source documents can be re-loaded into the Prolog environment such that mechanized formalizations can be developed in a workflow similar to programming in AI languages like Prolog and Lisp.

First-order reasoners are often heavily dependent on configuration settings. A *PIE document* specifies all information needed to reproduce the results of reasoner invocations in a convenient way. Effective configuration parameters are combined from system defaults, defaults declared in the document and options supplied with particular specifications of reasoner invocations.

3 Second-Order Quantifier Elimination in *PIE*

Second-order quantifier elimination is the task of computing for a given formula with second-order quantifiers, that is, quantifiers upon predicate or function symbols, an equivalent first-order formula. *PIE* so far just supports second-order quantification upon predicate symbols, or *predicate quantification*. Here is an example of *PIE*'s LaTeX representation of the invocation of a reasoner that performs second-order quantifier elimination:

Input: $\exists p\,(\forall x\,(\mathsf{q}(x) \to p(x)) \wedge \forall x\,(p(x) \to \mathsf{r}(x)))$.
Result of elimination:

$$\forall x\,(\mathsf{q}(x) \to \mathsf{r}(x)).$$

The source code in the *PIE* document that effects this output is:

```
:- ppl_printtime(ppl_elim(ex2(p, (all(x, (q(x) -> p(x))),
                                  all(x, (p(x) -> r(x))))))).
```

The directive `ppl_printtime` effects that its argument is evaluated at "print time", that is, at *processing*, when the LaTeX presentation is generated.[1] The argument is an invocation of the elimination reasoner with the predicate `ppl_elim`. It has a formula as argument, possibly with predicate quantifiers. If called at "print time" it prints inputs and outputs formatted in LaTeX as shown above for the example. It can also be invoked in the context of plain Prolog processing, where it just effects that the output is pretty printed in Prolog syntax. The following interaction would, for example, be possible in the Prolog console:

```
?- ppl_elim(ex2(p, (all(x, (q(x) -> p(x))), all(x, (p(x) -> r(x)))))).
all(x, (q(x)->r(x)))
true.
```

[1] The prefix `ppl_` of this and related predicates should suggest *pretty-print in LaTeX format.*

Printing the output is performed there as a side effect. *SWI-Prolog* afterwards prints `true.` to indicate that the invocation of `ppl_elim` was successful. To access the output formula from a program, *PIE* provides two alternate means: With an option list `[printing=false, r=Result]` as second argument, `ppl_elim` does not effect that the elimination result is printed, but instead bound to the Prolog variable `Result` for further processing. The second way to access the result formula of the last reasoner invocation is with the supplied predicate `last_ppl_result(`*`Result`*`)`. This predicate may itself be used in macro definitions.

Let us take a brief look at the syntax of the argument formula of `ppl_elim` in the example. It represents a second-order formula as a Prolog ground term. Conjunction is represented as in Prolog by `,`/2 and implication by `->`/2, with standard operator settings from Prolog. The universal first-order quantifier is expressed by `all`/2 and the existential second-order quantifier by `ex2`/2.

PIE performs second-order quantifier elimination by an included Prolog implementation of the *DLS* algorithm [14], a method based on formula rewriting until second-order subformulas have a certain shape that allows elimination in one step by rewriting with Ackermann's lemma, an equivalence due to [1]. Implementing *DLS* brings about many subtle and interesting issues [23,10,54], for example, incorporation of non-deterministic alternative courses, dealing with un-Skolemization, simplification of formulas in non-clausal form and ensuring success of the method for certain input classes. The current implementation in *PIE* is far from optimum solutions of these issues, but can nevertheless be used in nontrivial applications and might contribute to improvements by making experiments possible.

Of course, second-order quantifier elimination on the basis of first-order logic does not succeed in general. Nevertheless, along with variants termed *forgetting*, *uniform interpolation* or *projection*, it has many applications, including deciding fragments of first-order logic [36,3], computation of frame correspondence properties from modal axioms [19,14,43], computation of circumscription [14], embedding nonmonotonic semantics in a classical setting [51,50], abduction with respect to classical and to nonmonotonic semantics [32,15,52], forgetting in knowledge bases [33,49,29,34,13], and approaches to modularization of knowledge bases derived from the notion of conservative extension [21,22,35]. Further applications of second-order quantifier elimination are described in the monograph [20].

For second-order quantifier elimination and similar operations there are several implementations based on modal and description logics, but very few on first-order logic: A Web service[2] invokes an implementation [17] of the *SCAN* algorithm [19]. *DLSForgetter* [2] is a recent system that implements the *DLS* algorithm [14]. An earlier implementation [23] of *DLS* seems to be no longer available.

[2] Available at `http://www.mettel-prover.org/scan/`.

4 Abduction with Second-Order Quantifier Elimination – Basic Use of *PIE* Macros

In the simplest case, a *PIE* formula macro serves as a formula label that may be used in subformula position in other formulas and is expanded into its definiens. Here is an example of such a *PIE* macro definition in the LaTeX presentation:

---

kb_1

Defined as

$$\begin{array}{ll} (\mathsf{sprinkler\_was\_on} \rightarrow \mathsf{wet(grass)}) & \wedge \\ (\mathsf{rained\_last\_night} \rightarrow \mathsf{wet(grass)}) & \wedge \\ (\mathsf{wet(grass)} \rightarrow \mathsf{wet(shoes)}). & \end{array}$$

The corresponding source is:

```
def(kb1) ::
(sprinkler_was_on -> wet(grass)),
(rained_last_night -> wet(grass)),
(wet(grass) -> wet(shoes)).
```

The source statement has the form `def(`*`MacroName`*`) :: `*`ExpansionFormula`*`.`, where `::` is an infix operator with lower precedence than the operators used as connectives for logical formulas. Formula kb_1 is now defined as a small knowledge base that expresses a variant of a scenario often used to illustrate abduction. Actually, we use it now to show how a certain form of computing abductive explanations can be considered as second-order quantifier elimination. It is based on the notion of *weakest sufficient condition* [32,15,51], which is basically a second-order formula that expresses the weakest formula in a given vocabulary that needs to be conjoined to given axioms to make a given theorem candidate an actual theorem. This second-order formula as such is not very informative as it contains the axioms and the theorem as constituents, with disallowed symbols bound by quantifiers and possibly renamed but still present. However, the result of applying elimination to that second-order formula provides the weakest sufficient condition in the proper sense, or, considered with respect to abduction, the weakest explanation.

PIE allows to specify macros with parameters that are represented by Prolog variables. We utilize this to specify schematically the weakest explanation (or weakest sufficient condition) of observation *Obs* on the complement of *Na* as assumables (*Na* should suggest ***n**on-**a**ssumables*) within knowledge base *Kb*:

$explanation(Kb, Na, Obs)$

Defined as

$$\forall Na\,(Kb \rightarrow Obs).$$

The corresponding source code is:

```
def(explanation(Kb, Na, Ob)) ::
all2(Na, (Kb -> Ob)).
```

`all2`/2 represents the universal second-order quantifier in *PIE*'s input formula syntax. The first argument of `all2` specifies the quantified predicates, either as a single Prolog atom or as list of atoms. In the example, there is the macro parameter *Na* that needs to be instantiated correspondingly when the macro is expanded. The expression $explanation(kb_1, [\mathsf{wet}], \mathsf{wet}(\mathsf{shoes}))$ expands into the following "non-informative" version of the weakest sufficient condition:

$$\begin{array}{l} \forall p\,((\mathsf{sprinkler\_was\_on} \rightarrow p(\mathsf{grass})) \wedge \\ \quad (\mathsf{rained\_last\_night} \rightarrow p(\mathsf{grass})) \wedge \\ \quad (p(\mathsf{grass}) \rightarrow p(\mathsf{shoes})) \qquad\qquad \rightarrow \\ \quad p(\mathsf{shoes})). \end{array}$$

Second-order quantifier elimination applied to this formula yields the proper weakest explanation for the observation `wet(shoes)` in which the predicate `wet` itself does not occur, with respect to the background knowledge base kb_1:

Input: $explanation(kb_1, [\mathsf{wet}], \mathsf{wet}(\mathsf{shoes}))$.
Result of elimination:

$$\mathsf{rained\_last\_night} \vee \mathsf{sprinkler\_was\_on}.$$

It was obtained by the following directive in the source document:

```
:- ppl_printtime(ppl_elim(explanation(kb1,[wet],wet(shoes)))).
```

In [52] this approach to abduction has been generalized to non-monotonic semantics of logic programming, including the three-valued partial stable models semantics.

5 Invoking Theorem Provers from *PIE*

The abductive explanation computed in the previous section can be validated with a theorem prover. The presentation of the prover invocation and the result is in *PIE* as follows:

This formula is valid: $kb_1 \wedge (\mathsf{rained\_last\_night} \vee \mathsf{sprinkler\_was\_on}) \rightarrow \mathsf{wet}(\mathsf{shoes})$.

The corresponding source directive is

```
:- ppl_printtime(ppl_valid((kb1, (rained_last_night ; sprinkler_was_on)
                           -> wet(shoes)))).
```

The semicolon `;`/2 represents disjunction, as in Prolog. The reasoner invocation predicate `ppl_valid` by default first calls the model searcher *Mace4* with a short timeout, and, if it can not find a "counter"-model of the negated formula, calls the prover *Prover9*, again with a short timeout.[3] Correspondingly, `ppl_valid`

[3] *Prover9* and *Mace4* were developed between 2005 and 2010 by William McCune. Their homepage is `https://www.cs.unm.edu/~mccune/prover9/`.

prints a representation of one of three result values: *valid*, *not valid* or *failed to validate* and in LaTeX "print time" mode also the input formula, as shown above.

Like `ppl_elim`, also `ppl_valid` can be called with a list of options as second argument. This allows to obtain Prolog term representations of *Prover9*'s resolution proof or *Mace4*'s model, to skip the call to *Mace4*, modify the configuration of *Mace4* and *Prover9*, or to specify another theorem prover to be called.

Other provers can be incorporated through a generic interface to the *TPTP* [47] syntax for proving problems, supported by most current first-order provers. In addition, *DIMACS* and *QDIMACS*, the common formats of SAT and QBF solvers, respectively, are supported by *PIE*. Large propositional formulas are handled there efficiently with an internal representation implemented with destructive term operations. Most of the support of propositional formulas is inherited from the precursor system *ToyElim* [53].

PIE also includes a Prolog-based first-order prover, *CM,* whose calculus can be understood as model elimination, clausal tableau construction [31], or the connection method [**?**], similar to provers of the *leanCoP* family [40,26,27]. Its implementation follows the compilation-based *Prolog Technology Theorem Prover (PTTP)* paradigm [46]. It computes proofs that are represented by Prolog terms and can be used to compute Craig interpolants (Sect. 8). Details and evaluation results are available at `http://cs.christophwernhard.com/pie/cmprover`.

6 Computing Circumscription as Second-Order Quantifier Elimination – *PIE* Macros with Prolog Bodies, Result Simplifications

The circumscription of a predicate P in a formula F is a formula whose models are the models I of F that are minimal with respect to P. That is, there is no model I' of F that is like I except that the extension of P in I' is a strict subset of the extension of P in I. Predicate circumscription can be expressed by a second-order schema such that the *computation* of circumscription is second-order quantifier elimination [14]. The second-order circumscription of predicate P in formula F can thus be defined as a *PIE* macro as follows:

$circ(P, F)$

Defined as

$$F \wedge \neg \exists P' \, (F' \wedge T_1 \wedge \neg T_2),$$

where

$F' := F[P \mapsto P']$,
$A :=$ arity of P in F,
$T_1 :=$ transfer clauses $[P/A\text{-}\mathsf{n}] \rightarrow [P']$,
$T_2 :=$ transfer clauses $[P'] \rightarrow [P/A\text{-}\mathsf{n}]$.

This definition utilizes that *PIE* macro definitions may contain a Prolog body that permits expansions involving arbitrary computations. Utility predicates with pretty-printing templates for use in these bodies are provided for common tasks. The source of the above definition reads:

```
def(circ(P, F)) ::
F, ~ex2(P_p, (F_p, T1, ~T2)) ::-
        mac_rename_free_predicate(F, P, pn, F_p, P_p),
        mac_get_arity(P, F, A),
        mac_transfer_clauses([P/A-n], p, [P_p], T1),
        mac_transfer_clauses([P/A-n], n, [P_p], T2).
```

The Prolog body is introduced with the `::-` operator, which is defined with a precedence between `::` and the operators used to represent logical formulas. The unary operator `~` represents negation in formulas.[4] The suffix `_p` used for some variable names is translated to the prime superscript in the LaTeX rendering. We only indicate here the effects of the auxiliary predicates in the Prolog body with an example: The formula $\mathit{circ}(\mathsf{p}, \mathsf{p}(\mathsf{a}))$ expands into:

$$\begin{array}{ll} \mathsf{p}(\mathsf{a}) & \wedge \\ \neg\exists q\,(q(\mathsf{a}) \wedge \forall x\,(q(x) \rightarrow \mathsf{p}(x)) \wedge \neg\forall x\,(\mathsf{p}(x) \rightarrow q(x))). & \end{array}$$

Second-order quantifier elimination can be applied to compute the circumscription for the example:

Input: $\mathit{circ}(\mathsf{p}, \mathsf{p}(\mathsf{a}))$.
Result of elimination:

$$\mathsf{p}(\mathsf{a}) \wedge \forall x\,(\mathsf{p}(x) \rightarrow x = \mathsf{a}).$$

As a more complex example, we consider circumscribing wet in kb_1:

Input: $\mathit{circ}(\mathsf{wet}, kb_1)$.
Result of elimination:

$$\begin{array}{ll} (\mathsf{rained\_last\_night} \rightarrow \mathsf{wet}(\mathsf{grass})) & \wedge \\ (\mathsf{sprinkler\_was\_on} \rightarrow \mathsf{wet}(\mathsf{grass})) & \wedge \\ (\mathsf{wet}(\mathsf{grass}) \rightarrow \mathsf{wet}(\mathsf{shoes})) & \wedge \\ \forall x\,(\mathsf{wet}(x) \rightarrow \mathsf{rained\_last\_night} \vee \mathsf{sprinkler\_was\_on}) & \wedge \\ \forall x\,(\mathsf{wet}(x) \wedge \mathsf{wet}(\mathsf{grass}) \rightarrow x = \mathsf{grass} \vee x = \mathsf{shoes}). & \end{array}$$

The first three implications of this output form the expansion of kb_1. The last two implications are added by the circumscription. This particular form was actually obtained by applying a certain simplification to the formula returned directly by the elimination method:

```
:- ppl_printtime(ppl_elim(circ(wet,kb1), [simp_result=[c6]])).
```

The option `[simp_result=[c6]]` supplied to `ppl_elim` effects that the elimination result is postprocessed by equivalence preserving conversions with the aim to

[4] The standard Prolog negation operator `\+` is not suited to represent classical negation as it symbolizes $\nvdash$, non-provability.

make it more readable. The conversion named `c6` chosen for this example converts to conjunctive normal form, applies various clausal simplifications and then converts back to a quantified first-order formula, involving un-Skolemization if required. That the last conjunct of the result can be replaced by the more succinct $\forall x\,(\mathsf{wet}(x) \rightarrow x = \mathsf{grass} \vee x = \mathsf{shoes})$ is, however, not detected by the current implementation.

Finding good presentations of formulas, in particular in presence of operations that yield formulas with essentially semantic characterizations, is a challenging topic in general.

7 Expressing Graph Colorability by a Second-Order Formula – *PIE* Macros with Parameters in Functor Position

One of the fundamental results of descriptive complexity is the equivalence of NP and expressibility by an existential second-order formula (with respect to finite models), that is, a first order formula prefixed with existential predicate quantifiers. This view allows, for example, to specify 2-colorability[5] with respect to a relation E that specifies a graph as follows:

$col_2(E)$

Defined as

$$\exists r \exists g\,(\forall x\,(r(x) \vee g(x)) \qquad \wedge$$
$$\forall x \forall y\,(E(x,y) \rightarrow \neg(r(x) \wedge r(y)) \wedge \neg(g(x) \wedge g(y)))).$$

The source of this definition is:

```
def(col2(E)) ::
ex2([r,g],
    ( all(x, (r(x) ; g(x))),
      all([x,y], (E(x,y) -> (~((r(x), r(y))), ~((g(x), g(y))))))).
```

The macro parameter `E` appears as a Prolog variable in predicate position.[6] The macro can then be used with instantiating `E` to a predicate symbol, or to a λ-expression that describes a particular graph (we will see examples in a moment).

Specifying algorithms as (existential) second-order formulas seems very elegant, but so far not established as a *practical* approach to logic programming. *PIE* in its current implementation lets become apparent related desiderata: Instantiation with a predicate symbol should be usable as basis for abstract reasoning. Instantiation with a λ-expression (or conjoining a definition of a graph),

[5] 3-colorability, which is NP-complete, can be specified analogously. We consider here 2-colorability for brevity of the involved formulas.

[6] *SWI-Prolog* can be configured to permit variable names as functors, which are read in as atoms with capitalized names. The macro processor of *PIE* compares them to actual variable names in the macro definition.

should permit successful elimination. If adequate, the problem should then automatically be converted to a form that can be processed by a SAT solver.

So far, in the current implementation of *PIE*, such steps just work in part, e.g., by decomposing the overall task manually into intermediate steps with different manually controlled formula simplifications, as illustrated by the following example. The following macro defines the inner, first-order, component of the above specification of 2-colorability:

$fo\_col_2(E)$

Defined as

$$\forall x\,(\mathsf{r}(x) \vee \mathsf{g}(x)) \qquad \wedge$$
$$\forall x \forall y\,(E(x,y) \rightarrow \neg(\mathsf{r}(x) \wedge \mathsf{r}(y)) \wedge \neg(\mathsf{g}(x) \wedge \mathsf{g}(y))).$$

PIE allows to instantiate E in $fo\_col2(E)$ with a predicate constant `e` and eliminate one of the color predicates:[7]

Input: $\exists g\, fo\_col_2(\mathsf{e})$.
Result of elimination:

$$\forall x \forall y\,(\mathsf{e}(x,y) \rightarrow \neg(\mathsf{r}(y) \wedge \mathsf{r}(x)) \wedge (\mathsf{r}(y) \vee \mathsf{r}(x))).$$

2-colorability for a given graph represented by a λ-expression can be evaluated by *PIE* currently just in two steps with different elimination configurations, as performed by the following Prolog predicate:

```
elim_col2(E) :-
        ppl_elim(ex2([g], fo_col2(E)),
                 [elim_options=[pre=[c6]], printing=false, r=F1]),
        ppl_elim(ex2([r], F1),
                 [elim_options=[pre=[d6]], printing=false, r=F2]),
        ppl_form(E),
        ppl_form(F2).
```

Options `printing=false` suppress the emission of printed representations of the two invocations of the elimination reasoner. Only the input λ-expression and the final result are pretty-printed with calls to `ppl_form`. Options `pre=[c6]` and `pre=[d6]` effect that preprocessing based on conversion to CNF and DNF, respectively, is applied for elimination. Invoking

```
:- ppl_printtime(elim_col2(lambda([u,v],((u=1,v=2); (u=2,v=3))))).
```

yields the following output:

$$\lambda(u,v).(u = 1 \wedge v = 2) \vee (u = 2 \wedge v = 3).$$
$$1 \neq 2 \wedge 2 \neq 3.$$

It expresses that the graph described by the λ-expression is 2-colorable if and only if node 1 is not the same as node 2 and node 2 is not the same as node 3.

[7] One color predicate can also be eliminated from an analogous specification of 3-colorability.

8 Craig Interpolation

By Craig's interpolation theorem [11,12], for given first-order formulas F and G such that F entails G (or, equivalently, $F \rightarrow G$ is valid) a first-order formula H can be constructed such that F entails H, H entails G and H contains only symbols (predicates, functions, constants, free variables) that occur in both F and G. *PIE* supports the computation of Craig interpolants H, for given valid implications $F \rightarrow G$. Here is a propositional example:

Input: $\mathsf{p} \wedge \mathsf{q} \rightarrow \mathsf{p} \vee \mathsf{r}$.
Result of interpolation:

$$\mathsf{p}.$$

The corresponding directive in the source document is:

```
:- ppl_printtime(ppl_ipol((p, q -> (p ; r)))).
```

The predicate `ppl_ipol` invokes the interpolation reasoner. It takes an implication $F \rightarrow G$ as argument and, analogously to `ppl_elim` (Sect. 3), prints an interpolant of F and G.[8] Here is another example of Craig interpolation, where universal and existential quantification need to be combined:[9]

Input: $\forall x\, \mathsf{p}(\mathsf{a}, x) \wedge \mathsf{q} \rightarrow \exists x\, \mathsf{p}(x, \mathsf{b}) \vee \mathsf{r}$.
Result of interpolation:

$$\exists x\, \forall y\, \mathsf{p}(x, y).$$

Craig interpolation has many applications in logics and philosophy, as already shown in [12]. Main applications in computer science are in verification [39] and query reformulation, based on its relationship to definability and construction of definientia in terms of a given vocabulary [48,5,4]. For these applications, actually interpolants that are further constrained, in dependency of further restrictions on the input formulas, are relevant. We do not consider these here, but show how basic definability via Craig interpolation can be expressed in *PIE*.

A formula G is called *definable* in a formula F *in terms of* a set of predicates S if and only if there exists a formula H whose predicates are all in S such that $F \models G \leftrightarrow H$. The formula H is then called a *definiens* of G. Consider, for example, the following formula:

[8] In certain configurations it can also print several different interpolants.

[9] This is an example which involves an inference step with a constant that occurs only on the left side (a) and a constant that occurs only on the right side (b), which can not be handled by certain resolution-based interpolation systems. See [7,30]. In this particular example, the order of the quantifications in the result is not relevant.

kb_2

Defined as

$$\begin{array}{ll} \forall x\,(\mathsf{p}(x) \rightarrow \mathsf{q}(x) \wedge \mathsf{s}(x)) & \wedge \\ \forall x\,(\mathsf{s}(x) \rightarrow \mathsf{r}(x)) & \wedge \\ \forall x\,(\mathsf{q}(x) \wedge \mathsf{r}(x) \rightarrow \mathsf{p}(x)). & \end{array}$$

We can invoke a first-order prover from *PIE* to verify that the formula $\mathsf{p}(\mathsf{a})$ is definable in kb_2 in terms of $\{\mathsf{q}, \mathsf{r}\}$:

This formula is valid: $kb_2 \rightarrow (\mathsf{p}(\mathsf{a}) \leftrightarrow \mathsf{q}(\mathsf{a}) \wedge \mathsf{r}(\mathsf{a}))$.

Actually, since a does not occur in kb_2, we can equivalently verify the following implication, whose right side is a universally quantified first-order definition:

This formula is valid: $kb_2 \rightarrow \forall a\,(\mathsf{p}(a) \leftrightarrow \mathsf{q}(a) \wedge \mathsf{r}(a))$.

We can now utilize the features of *PIE* to formally characterize definability and synthesize definientia:

definiens(G, F, P)

Defined as

$$\exists P\,(F \wedge G) \rightarrow \forall P\,(F \rightarrow G).$$

The interpolants of the left and right side of *definiens*(G, F, P) are exactly the definientia of G in F in terms of all predicates not in P. The implication is valid if and only if definability holds. The second-order quantifications in the implication are existential on the left and universal on the right side.[10] Considering that an implication can be understood as disjunction of the *negated* left side and the right side, if F and G are first-order, then *definiens*(G, F, P) is a formula whose second-order quantifiers are all *universal*. Such a second-order formula is valid if and only if the first-order formula obtained by renaming the quantified predicates with fresh symbols and dropping the second-order quantifiers is valid. This translation is handled automatically by *PIE* such that we can now we verify definability of $\mathsf{p}(\mathsf{a})$ by invoking a first-order prover from *PIE*:

This formula is valid: *definiens*$(\mathsf{p}(\mathsf{a}), kb_2, [\mathsf{p}, \mathsf{s}])$.

And, we can apply Craig interpolation to compute a definiens:

Input: *definiens*$(\mathsf{p}(\mathsf{a}), kb_2, [\mathsf{p}, \mathsf{s}])$.
Result of interpolation:

$$\mathsf{q}(\mathsf{a}) \wedge \mathsf{r}(\mathsf{a}).$$

[10] We actually encountered right side of the implication before in Sect. 4 as the weakest sufficient condition in the macro definition of *explanation*.

The implementation of the computation of Craig interpolants in *PIE* operates by a novel adaption of Smullyan's interpolation method [45,18] to clausal tableaux [57]. Suitable clausal tableaux can be constructed by the Prolog-based prover *CM* that is included in *PIE*. The system also supports the conversion of proof terms returned by the hypertableau prover *Hyper* [41] to such tableaux and thus to interpolants, but this is currently at an experimental stage.[11]

The interpolants H constructed by *PIE* strengthen the requirements for Craig interpolants in that they are actually Craig-Lyndon interpolants, that is, predicates occur in H only in polarities in which they occur in both F and G. Symmetric interpolation [38, Sect. 5] is supported in *PIE*, implemented by computing a conventional interpolant for each of the input formulas, corresponding to the induction suggested with [12, Lemma 2].

It seems that most other implementations of Craig interpolation are on the basis of propositional logic with theory extensions and specialized for applications in verification [4]. Craig interpolation for first-order logic is supported by *Princess* [9,8] and by extensions of *Vampire* [25,24]. The incompleteness indicated in footnote 9 applies to these *Vampire* extensions and was observed by their authors. It also appears that the *Vampire* extensions do not preserve the polarity constraints of Craig-Lyndon interpolants [4].

9 Further Features of *PIE*

In this section we briefly describe further features of *PIE* that were not illustrated by the examples in the previous sections. First we consider the formula macro system. It utilizes Prolog variables to mimic further features of the processing of λ-expressions by automatically binding a Prolog variable that is free after computing the user-specified part of the expansion to a freshly generated symbol. With a macro declaration, properties of its lexical environment, in particular configuration settings that affect the expansion, are recorded. Macros with parameters are processed by pattern matching to choose the effective declaration for expansion, allowing structural recursion in macro declarations.

A Craig interpolant for formulas F and G is extracted in *PIE* from a Prolog term that represents a closed clausal tableau, a proof of the validity of $F \to G$. *PIE* supports the visualization of such tableaux as graph, rendered by the *Graphviz* tool. Here is an example:

Input: $\forall x\, \mathsf{p}(x) \wedge \forall x\, (\mathsf{p}(x) \to \mathsf{q}(x)) \to \mathsf{q}(\mathsf{c})$.
Result of interpolation:

$$\forall x\, \mathsf{q}(x).$$

The respective directive for this interpolation task in the source is:

[11] Hypertableaux, either obtained from a hypertableau prover or obtained from a clausal tableau prover like *CM* by restructuring the tableau seem interesting as basis for interpolant extraction in query reformulation, as they allow to ensure that the interpolants are range restricted. Some related preliminary results are in [57].

```
:- ppl_printtime(ppl_ipol((all(x, p(x)), all(x, (p(x) -> q(x))) -> q(c)),
                          [ip_dotgraph=printstyle('/tmp/tmp01.png'),
                           ip_simp_sides=false])).
```

The `ip_dotgraph` option effects that an image representing the tableau is generated. The `ip_simp_sides` option suppresses preprocessing of the interpolation input, which, in the example, would in essence be already sufficient to compute the interpolant, yielding a trivial tableau. The generated image can then be included into the *PIE* document with standard LaTeX means, here, for example as Fig. 1. Siblings in the tableau represent a ground clause used in the proof. As the tableau is used for interpolant extraction, decoration indicates whether the clause stems from the left or the right side of the input formula. The decoration of the closing marks indicate the side of the connection partner. The Skolem constant `sk1` is converted to a quantified variable in a postprocessing operation. For a description of the interpolant extraction procedure, see [57].

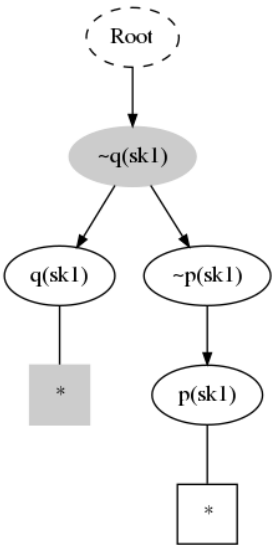

Fig. 1. A clausal tableau.

Aside of the shown representation of quantified first-order formulas by Prolog ground terms, the system also supports a representation of clausal formulas as list of lists of terms (logic literals), with variables represented by Prolog variables. The system functionality can be accessed by Prolog predicates, also without using the document processing facilities.

Practically successful reasoners usually apply in some way conversions of low complexity as far as possible: as preprocessing on inputs, potentially during reasoning, which has been termed *inprocessing*, and to improve the syntactic shape of output formulas as discussed in Sect. 6. Abstracting from these situations, we subsume these conversions under *preprocessing operations*. Also the low complexity might be taken more or less literally and, for example, be achieved simply by trying an operation within a threshold limit of resources. *PIE* includes a number of preprocessing operations including normal form conversions, also in variants that produce structure preserving normalizations, various simplifications of clausal formulas, and an implementation of McCune's un-Skolemization algorithm [37]. While some of these preserve equivalence, others preserve equivalence just with respect to a set of predicates, for example, purity simplification with respect to predicates that are not deleted or structure preserving clausification with respect to predicates that are not added. This can be understood as preserving the second-order equivalence

$$\exists q_1 \ldots \exists q_n \, F \;\equiv\; \exists q_1 \ldots \exists q_n \, G,$$

where F and G are inputs and outputs of the conversion and $q_1, \ldots, q_n$ are those predicates that are permitted to occur in F or G whose semantics needs

not to be preserved. If $q_1, \ldots, q_n$ includes all permitted predicates, the above equivalence expresses equi-satisfiability. Some of the simplifications implemented in *PIE* allow to specify explicitly a set of predicates whose semantics is to be preserved, which makes them applicable for Craig interpolation and second-order quantifier elimination.

In addition to the implementation of the *DLS* algorithm, *PIE* includes further experimental implementations of variants of second-order quantifier elimination. In particular, a variant of the method shown in [33] for elimination with respect to ground atoms, which always succeeds on the basis of first-order logic. A second-order quantifier is there, so-to-speak, just upon a particular ground instance of a predicate. The *Boolean solution problem* or *Boolean unification with predicates* is a computational task related to second-order quantifier elimination [44,42,56]. So far, *PIE* includes experimental implementations for special cases: Quantifier-free formulas with a technique from [16] and a version for finding solutions with respect to ground atoms, in analogy to the elimination of ground atoms.

10 Conclusion

PIE tries to supplement what is needed to use automated first-order proving techniques for developing and analyzing formalizations. Its main focus is not on proofs but on *formulas*, as constituents of complex formalizations that are composed and structured through macros, and as computed outputs of second-order quantifier elimination, Craig interpolation and formula conversions that preserve semantics with respect to given predicates. All of these operations utilize some natural relationships between first- and second-order logic.

The system mediates between high-level logical presentation and detailed configuration of reasoning systems: Working practically with first-order provers typically involves experimenting with a large and developing set of related proving problems, for example with alternate axiomatizations or different candidate theorems, and is thus often accompanied with some meta-level technique to compose and relate the actual proof tasks submitted to first-order reasoners. With the macro system, the supported document-oriented workflow, LaTeX pretty-printing, and integration into the Prolog environment, *PIE* offers to organize this in a systematic way through mechanisms that remain in the spirit of first-order logic, which in mathematics is actually often used with schemas.

Aside of the current suitability for non-trivial applications, *PIE* shows up a number of challenging and interesting open issues for research, for example improving practical realizations of second-order quantifier elimination, strengthenings of Craig interpolation that ensure application-relevant properties such as range restriction, and conversion of computed formulas that are basically just semantically characterized to comprehensible presentations. Progress in these issues can be directly experienced and verified with the system.

References

1. Ackermann, W.: Untersuchungen über das Eliminationsproblem der mathematischen Logik. Mathematische Annalen **110**, 390–413 (1935)
2. Alassaf, R., Schmidt, R.: DLS-Forgetter: An implementation of the DLS forgetting calculus for first-order logic. In: GCAI 2019. EPiC, vol. 65, pp. 127–138 (2019)
3. Behmann, H.: Beiträge zur Algebra der Logik, insbesondere zum Entscheidungsproblem. Mathematische Annalen **86**(3–4), 163–229 (1922)
4. Benedikt, M., Kostylev, E.V., Mogavero, F., Tsamoura, E.: Reformulating queries: Theory and practice. In: IJCAI 2017. pp. 837–843. ijcai.org (2017)
5. Benedikt, M., Leblay, J., ten Cate, B., Tsamoura, E.: Generating Plans from Proofs: The Interpolation-based Approach to Query Reformulation. Morgan & Claypool (2016)
6. Bibel, W.: Matings in matrices. Commun. ACM **26**(11), 844–852 (1983)
7. Bonacina, M.P., Johansson, M.: On interpolation in automated theorem proving. Journal of Automated Reasoning **54**(1), 69–97 (2015)
8. Brillout, A., Kroening, D., Rümmer, P., Wahl, T.: Beyond quantifier-free interpolation in extensions of Presburger arithmetic. In: VMCAI 2011. pp. 88–102. Springer (2011)
9. Brillout, A., Kroening, D., Rümmer, P., Wahl, T.: An interpolating sequent calculus for quantifier-free Presburger arithmetic. Journal of Automated Reasoning **47**(4), 341–367 (2011)
10. Conradie, W.: On the strength and scope of DLS. Journal of Applied Non-Classsical Logics **16**(3–4), 279–296 (2006)
11. Craig, W.: Linear reasoning. A new form of the Herbrand-Gentzen theorem. Journal of Symbolic Logic **22**(3), 250–268 (1957)
12. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. Journal of Symbolic Logic **22**(3), 269–285 (1957)
13. Delgrande, J.P.: A knowledge level account of forgetting. Journal of Artificial Intelligence Research **60**, 1165–1213 (2017)
14. Doherty, P., Łukaszewicz, W., Szałas, A.: Computing circumscription revisited: A reduction algorithm. Journal of Automated Reasoning **18**(3), 297–338 (1997)
15. Doherty, P., Łukaszewicz, W., Szałas, A.: Computing strongest necessary and weakest sufficient conditions of first-order formulas. In: IJCAI-01. pp. 145–151. Morgan Kaufmann (2001)
16. Eberhard, S., Hetzl, S., Weller, D.: Boolean unification with predicates. Journal of Logic and Computation **27**(1), 109–128 (2017)
17. Engel, T.: Quantifier Elimination in Second-Order Predicate Logic. Master's thesis, Max-Planck-Institut für Informatik, Saarbrücken (1996)
18. Fitting, M.: First-Order Logic and Automated Theorem Proving. Springer, 2nd edn. (1995)
19. Gabbay, D., Ohlbach, H.J.: Quantifier elimination in second-order predicate logic. In: KR'92. pp. 425–435. Morgan Kaufmann (1992)
20. Gabbay, D.M., Schmidt, R.A., Szałas, A.: Second-Order Quantifier Elimination: Foundations, Computational Aspects and Applications. College Publications (2008)
21. Ghilardi, S., Lutz, C., Wolter, F.: Did I damage my ontology? A case for conservative extensions in description logics. In: KR 06. pp. 187–197. AAAI Press (2006)

22. Grau, B.C., Horrocks, I., Kazakov, Y., Sattler, U.: Modular reuse of ontologies: Theory and practice. Journal of Artificial Intelligence Research **31**(1), 273–318 (2008)
23. Gustafsson, J.: An implementation and optimization of an algorithm for reducing formulae in second-order logic. Tech. Rep. LiTH-MAT-R-96-04, Univ. Linköping (1996)
24. Hoder, K., Holzer, A., Kovács, L., Voronkov, A.: Vinter: A Vampire-based tool for interpolation. In: APLAS 2012. pp. 148–156. Springer (2012)
25. Hoder, K., Kovács, L., Voronkov, A.: Interpolation and symbol elimination in Vampire. In: IJCAR 2010. pp. 188–195. Springer (2010)
26. Kaliszyk, C.: Efficient low-level connection tableaux. In: TABLEAUX 2015. pp. 102–111. Springer (2015)
27. Kaliszyk, C., Urban, J.: FEMaLeCoP: Fairly efficient machine learning connection prover. In: LPAR-20. pp. 88–96. Springer (2015)
28. Knuth, D.E.: Literate programming. The Computer Journal **27**(2), 97–111 (1984)
29. Koopmann, P., Schmidt, R.A.: Uniform interpolation of $\mathcal{ALC}$-ontologies using fixpoints. In: FroCoS 2013. pp. 87–102. Springer (2013)
30. Kovács, L., Voronkov, A.: First-order interpolation and interpolating proof systems. In: LPAR-21. pp. 49–64. EasyChair (2017)
31. Letz, R.: First-order tableau methods. In: Marcello D'Agostino, Dov M. Gabbay, R.H., Posegga, J. (eds.) Handbook of Tableau Methods, pp. 125–196. Kluwer Academic Publishers (1999)
32. Lin, F.: On strongest necessary and weakest sufficient conditions. Artificial Intelligence **128**, 143–159 (2001)
33. Lin, F., Reiter, R.: Forget It! In: Working Notes, AAAI Fall Symp. on Relevance. pp. 154–159 (1994)
34. Ludwig, M., Konev, B.: Practical uniform interpolation and forgetting for $\mathcal{ALC}$ TBoxes with applications to logical difference. In: KR'14. AAAI Press (2014)
35. Lutz, C., Wolter, F.: Foundations for uniform interpolation and forgetting in expressive description logics. In: IJCAI-11. pp. 989–995. AAAI Press (2011)
36. Löwenheim, L.: Über Möglichkeiten im Relativkalkül. Mathematische Annalen **76**, 447–470 (1915)
37. McCune, W.: Un-Skolemizing clause sets. Information Processing Letters **29**(5), 257–263 (1988)
38. McMillan, K.L.: Applications of Craig interpolants in model checking. In: TACAS 2005. pp. 1–12. Springer (2005)
39. McMillan, K.L.: Interpolation and model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 421–446. Springer (2018)
40. Otten, J.: Restricting backtracking in connection calculi. AI Communications **23**(2-3), 159–182 (2010)
41. Pelzer, B., Wernhard, C.: System description: E-KRHyper. In: CADE-21. pp. 503–513. Springer (2007)
42. Rudeanu, S.: Boolean Functions and Equations. Elsevier (1974)
43. Schmidt, R.A.: The Ackermann approach for modal logic, correspondence theory and second-order reduction. Journal of Applied Logic **10**(1), 52–74 (2012)
44. Schröder, E.: Vorlesungen über die Algebra der Logik. Teubner (1890–1905)
45. Smullyan, R.M.: First-Order Logic. Dover publications, New York (1995), corrected republication of the original edition by Springer-Verlag, New York, 1968
46. Stickel, M.E.: A Prolog technology theorem prover: implementation by an extended Prolog compiler. Journal of Automated Reasoning **4**(4), 353–380 (1988)

47. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. Journal of Automated Reasoning **59**(4), 483–502 (2017)
48. Toman, D., Weddell, G.: Fundamentals of Physical Design and Query Compilation. Morgan and Claypool (2011)
49. Wernhard, C.: Semantic knowledge partitioning. In: JELIA 04. pp. 552–564. Springer (2004)
50. Wernhard, C.: Circumscription and projection as primitives of logic programming. In: Technical Communications ICLP'10. LIPIcs, vol. 7, pp. 202–211. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2010)
51. Wernhard, C.: Projection and scope-determined circumscription. Journal of Symbolic Computation **47**, 1089–1108 (2012)
52. Wernhard, C.: Abduction in logic programming as second-order quantifier elimination. In: FroCoS 2013. pp. 103–119. Springer (2013)
53. Wernhard, C.: Computing with logic as operator elimination: The ToyElim system. In: INAP 2011/WLP 2011. pp. 289–296. Springer (2013)
54. Wernhard, C.: Second-order quantifier elimination on relational monadic formulas – A basic method and some less expected applications. In: TABLEAUX 2015. pp. 249–265. Springer (2015)
55. Wernhard, C.: The PIE system for proving, interpolating and eliminating. In: PAAR 2016. pp. 125–138. CEUR-WS.org (2016)
56. Wernhard, C.: The Boolean solution problem from the perspective of predicate logic. In: FroCoS 2017. LNCS (LNAI), vol. 10483, pp. 333–350. Springer (2017)
57. Wernhard, C.: Craig interpolation and access interpolation with clausal first-order tableaux. ArXiv e-prints (2018), `https://arxiv.org/abs/1802.04982`
58. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming **12**(1-2), 67–96 (2012)