Computing with Logic as Operator Elimination: The ToyElim System

Christoph Wernhard

Technische Universität Dresden
christoph.wernhard@tu-dresden.de

Abstract. A prototype system is described whose core functionality is, based on propositional logic, the elimination of second-order operators, such as Boolean quantifiers and operators for projection, forgetting and circumscription. This approach allows to express many representational and computational tasks in knowledge representation – for example computation of abductive explanations and models with respect to logic programming semantics – in a uniform operational system, backed by a uniform classical semantic framework.

1 Computation with Logic as Operator Elimination

We pursue an approach to computation with logic emerging from three theses:

> *1. Classical first-order logic extended by some second-order operators suffices to express many techniques of knowledge representation.*

Like the standard logic operators, second-order operators can be defined *semantically*, by specifying the requirements on an interpretation to be a model of a formula whose principal functor is the operator, depending only on semantic properties of the argument formulas. Neither control structure imposed over formulas (e.g. Prolog), nor formula transformations depending on a particular syntactic shape (e.g. Clark's completion) are involved. Compared to classical first-order formulas, the second-order operators give additional expressive power. Circumscription is a prominent knowledge representation technique that can be expressed with second-order operators, in particular predicate quantifiers [1].

> *2. Many computational tasks can be expressed as elimination of second-order operators.*

Elimination is a way to computationally process second-order operators, for example Boolean quantifiers with respect to propositional logic: The input is a formula which may contain the operator, for example a quantified Boolean formula such as $\exists q\,((p \leftarrow q) \wedge (q \leftarrow r))$. The output is a formula that is equivalent to the input, but in which the operator does not occur, such as, with respect to the formula above, the propositional formula $p \leftarrow r$. Let us assume that the method used to eliminate the Boolean quantifiers returns formulas in which not just the quantifiers but also the quantified propositional variables do not occur. This syntactic condition is usually met by elimination procedures. Our method

then subsumes a variety of tasks: Computation of uniform interpolants, QBF and SAT solving, as well as computation of certain forms of abductive explanations, of propositional circumscription, and of stable models, as will be outlined below.

> *3. Depending on the application, outputs of computation with logic are conveniently represented by formulas meeting syntactic criteria.*

If results of elimination are formulas characterized just up to semantics, they may contain redundancies and be in a shape that is difficult to comprehend. Thus, they should be subjected to simplification and canonization procedures before passed to humans or to machine clients. The output format depends on the application problem: What is a CNF of the formula? Are certain facts consequences of the formula? What are the models of the formula? What are its minimal models? What are its 3-valued models with respect to some encoding into 2-valued logics? Corresponding answers can be computed on the basis of normal form representations of the elimination outputs: CNFs, DNFs, full DNFs, and prime implicant forms. Of course, transformation into such normal forms might by itself be an expensive task. Second-order operators allow to counter this by specifying a small set of application relevant symbols that should be included in the output, e.g. by Boolean quantification upon the irrelevant atoms.

2 Features of the System

ToyElim[1] is a prototype system developed to investigate operator elimination from a pragmatic point of view with small applications. For simplicity, it is based on propositional logic, although its characteristic features should transfer to first-order logic. It supports a set of second-order operators that have been semantically defined in [11, 15, 13].

Formula Syntax. As the system is implemented in Prolog, formulas are represented by Prolog terms, the standard connectives corresponding to `true`/0, `false`/0, `~`/1, `,`/2, `;`/2, `->`/2, `<-`/2, `<->`/2. Propositional atoms are represented by Prolog atoms or compound ground terms. The system supports propositional expansion with respect to finite domains of formulas containing first-order quantifiers.

Forgetting. Existential Boolean quantification $\exists p\ F$ can be expressed as *forgetting* [11, 4] in formula F about atom p, written $\mathsf{forget}_{\{p\}}(F)$, represented by `forg([p], F')` in system syntax, where F' is the system representation of F. To get an intuition of forgetting, consider the equivalence $\mathsf{forget}_{\{p\}}(F) \equiv F[p\backslash\mathsf{true}] \vee F[p\backslash\mathsf{false}]$, where $F[p\backslash\mathsf{true}]$ ($F[p\backslash\mathsf{false}]$) denotes F with all occurrences of p replaced by true (false). Rewriting with this equivalence constitutes a naive method for eliminating the forgetting operator. The formula $\mathsf{forget}_{\{p\}}(F)$ can be said to express the same as F about all other atoms than p, but nothing about p.

[1] http://cs.christophwernhard.com/toyelim/, under GNU Public License.

Elimination and Pretty Printing of Formulas. The central operation of the ToyElim system, elimination of second-order operators, is performed by the predicate `elim(`F`,`G`)`, with input formula F and output formula G. For example, define as extension of `kb1`/1 a formula (after [3]) as follows:

```
kb1(((shoes_are_wet <- grass_is_wet),                                    (1)
     (grass_is_wet <- rained_last_night),
     (grass_is_wet <- sprinkler_was_on))).
```

After consulting this, we can execute the following query on the Prolog toplevel:

```
?- kb1(F), elim(forg([grass_is_wet], F), G), ppr(G).                      (2)
```

This results in binding `G` to the output of eliminating the forgetting about `grass_is_wet`. The predicate `ppr`/1 is one of several provided predicates for converting formulas into application adequate shapes. It prints its argument as CNF with clauses written as reverse implications:

```
((shoes_are_wet <- rained_last_night),                                   (3)
 (shoes_are_wet <- sprinkler_was_on)).
```

Scopes. So far, the first argument of forgetting has been a singleton set. More generally, it can be an arbitrary set of atoms, corresponding to nested existential quantification. Even more generally, also polarity can be considered: Forgetting can, for example, be applied only to those occurrences of an atom which have negative polarity in a NNF formula. This can be expressed by *literals* with explicitly written sign in the first argument of the forgetting operator. Forgetting about an atom is equivalent to nested forgetting about the positive and the negative literal with that atom. In accord with this observation, we technically consider the first argument of forgetting always as a *set of literals*, and regard an unsigned atom there as a shorthand representing both of its literals. For example, `[+grass_is_wet, shoes_are_wet]` is a shorthand for `[+grass_is_wet, +shoes_are_wet, -shoes_are_wet]`. Not just forgetting, but, as shown below, also other second-order operators have a set of literals as parameter. Hence, we refer to a set of literals in this context by a special name, as *scope*.

Projection. In many applications it is useful to make explicit not the scope that is "forgotten" about, but what is preserved. The *projection* [11] of formula F onto scope S, which can be defined for scopes S and formulas F as $\mathsf{project}_S(F) \equiv \mathsf{forget}_{\mathsf{ALL}-S}(F)$, where ALL denotes the set of all literals, serves this purpose. Vice versa, forgetting could be defined in terms of projection: $\mathsf{forget}_S(F) \equiv \mathsf{project}_{\mathsf{ALL}-S}(F)$. The call to `elim`/2 in the query (2) can equivalently be expressed with projection instead of forgetting by

```
elim(proj([shoes_are_wet, rained_last_night, sprinkler_was_on], F).      (4)
```

User Defined Logic Operators – An Example of Abduction. ToyElim allows the user to specify macros for use in the input formulas of `elim`/2. The following example extends the system by a logic operator `gwsc` for a variant of the weakest sufficient condition [8], characterized in terms of projection:

```
:- define_elim_macro(gwsc(S, F, G), ~proj(complements(S), (F, ~G))).
```
(5)

Here `complements(S)` specifies the set of the literal complements of the members of the scope specified by `S`. The term `gwsc(S, F, G)` is the system syntax for $\mathsf{gwsc}_S(F, G)$, the *globally weakest sufficient condition* [15] of formula G on scope S within formula F, which satisfies the following: A formula H is equivalent to $\mathsf{gwsc}_S(F, G)$ if and only if it holds that (1.) $H \equiv \mathsf{project}_S(H)$; (2.) $F \models H \rightarrow G$; (3.) For all formulas H' such that $H' \equiv \mathsf{project}_S(H')$ and $F \models H' \rightarrow G$ it holds that $H' \models H$. With the `gwsc` operator certain abductive tasks [3] can be expressed. The following query, for example, yields abductive explanations for `shoes_are_wet` in terms of {`rained_last_night`, `sprinkler_was_on`} with respect to the knowledge base (1):

```
?- kb1(F),                                                    (6)
   elim(gwsc([rained_last_night, sprinkler_was_on], F, shoes_are_wet),
        G),
   ppp(G).
```

The predicate `ppp`/1 serves, like `ppr`/1, to convert formulas to application adequate shape. It writes a prime implicate form of its input in list notation. In the example the output has two clauses, each representing an alternate explanation:

```
[[rained_last_night],[sprinkler_was_on]].                     (7)
```

Scope-Determined Circumscription. A further second-order operator supported by ToyElim is *scope-determined circumscription* [15]. The corresponding functor `circ` has, like `proj` and `forg`, a scope specifier and a formula as arguments. It allows to express *parallel predicate circumscription with varied predicates* [5] (only propositional, since the system is based on propositional logic). The scope specifier controls the effect of circumscription: Atoms that occur just in a *positive* literal in the scope are minimized; symmetrically, atoms that occur just *negatively* are maximized; atoms that occur in *both polarities* are fixed; and atoms that do *not at all* occur in the scope are allowed to vary. For example, the scope specifier, `[+abnormal, bird]`, a shorthand for `[+abnormal, +bird, -bird]`, expresses that `abnormal` is minimized, `bird` is fixed, and all other predicates are varied.

Predicate Groups and Systematic Renaming. Semantics for knowledge representation sometimes involve what might be described as handling different occurrences of a predicate differently – for example depending on whether they are subject to negation as failure. If such semantics are to be modeled with classical logic, then these occurrences can be identified by using distinguished predicates, which are equated with the original ones when required. To this end, ToyElim supports the handling of *predicate groups*: The idea is that each

predicate actually is represented by several *corresponding* predicates $p^0, \ldots, p^n$, where the superscripted index is called *predicate group*. In the system syntax, the predicate group of an atom is represented within its main functor: If the group is larger than 0, the main functor is suffixed by the group number; if it is 0, the main functor does not end in a number. For example $p(a)^0$ and $p(a)^1$ are represented by `p(a)` and `p1(a)`, respectively. In scope specifiers, a number is used as shorthand for the set of all literals whose atom is from the indicated group, and a number in a sign functor for the set of those literals which have that sign and whose atom is from the indicated group. For example, `[+(0), 1]` denotes the union of the set of all positive literals whose atom is from group 0 and of the set of all literals whose atom is from group 1. Systematic renaming of all atoms in a formula that have a specific group to their correspondents from another group can be expressed in terms of forgetting [13]. The ToyElim system provides the second-order operator `rename` for this. For example, `rename([1-0], F)` is equivalent to F after eliminating second-order operators, followed by replacing all atoms from group 1 with their correspondents from group 0.

An Example of Modeling a Logic Programming Semantics. Scope-determined circumscription and predicate groups can be used to express the characterization of the stable models semantics in terms of circumscription [7] (described also in [6, 13]). Consider the following knowledge base:

```
kb2(((shoes_are_wet <- grass_is_wet),                                    (8)
     (grass_is_wet <- sprinkler_was_on, ~sprinkler_was_abnormal1),
     sprinkler_was_on)).
```

Group 1 is used here to indicate atoms that are subject to negation as failure: All atoms in (8) are from group 0, except for `sprinkler_was_abnormal1`, which is from 1. The user defined operator `stable` renders the stable models semantics:

```
:- define_elim_macro(stable(F), rename([1-0], circ([+(0),1], F))).       (9)
```

The following query then yields the stable models:

```
:- kb2(F), elim(stable((F)), G), ppp(G).                                 (10)
```

The result is displayed with `ppp`/1, as in query (6). It shows here a DNF with a single clause, representing a single model. The positive members of the clause constitute the answer set

```
[[grass_is_wet, shoes_are_wet, ~sprinkler_was_abnormal,                  (11)
  sprinkler_was_on]].
```

If it is only of interest whether `shoes_are_wet` is a consequence of the knowledge base under stable models semantics, projection can be applied to obtain a smaller result. The query

```
:- kb2(F), elim(proj([shoes_are_wet], stable(F)), G), ppp(G).            (12)
```

will effect that the DNF `[[shoes_are_wet]]` is printed.

3 Implementation

The ToyElim system is implemented in SWI-Prolog and can invoke external systems such as SAT and QBF solvers. It runs embedded in the Prolog environment, allowing for example to pass intermediate results between its components through Prolog variables, as exemplified by the queries shown above.

The implementation of the core predicate `elim`/2 maintains a formula which is gradually rewritten until it contains no more second-order operators. It is initialized with the input formula, preprocessed such that only two primitively supported second-order operators remain: forgetting and renaming. It then proceeds in a loop where alternately equivalence preserving simplifying rewritings are applied, and a subformula is picked and handed over for elimination to a specialized procedure. The simplifying rewritings include distribution of forgetting over subformulas and elimination steps that can be performed with low cost [12]. Rewriting of subformulas with the Shannon expansion enables low-cost elimination steps. It is performed at this stage if the expansion, combined with low-cost elimination steps and simplifications, does not lead to an increase of the formula size. The subformula for handing over to a specialized method is picked with the following priority: First, an application of forgetting upon the whole signature of a propositional argument, which can be reduced by a SAT solver to either true or false, is searched. Second, a subformula that can be reduced analogously by a QBF solver, and finally a subformula which properly requires elimination of forgetting. For the latter, ToyElim schedules a portfolio of different methods, where currently two algorithmic approaches are supported: Resolvent generation (SCAN, Davis-Putnam method) and rewriting of subformulas with the Shannon expansion [10, 12]. Recent SAT preprocessors partially perform variable elimination by resolvent generation. *Coprocessor* [9] is such a preprocessor that is configurable such that it can be invoked by ToyElim for the purpose of performing the elimination of forgetting.

4 Conclusion

We have seen a prototype system for computation with logic as elimination of second-order operators. The system helped to concretize requirements on systems following this approach, concerning the user interface and the processing methods. In the long run, such a system should be based on more expressive logics than propositional logic. ToyElim is just a first pragmatic attempt, taking advantage of recent advances in SAT solving. A major difference in a first-order setting is that computations of elimination tasks then inherently do not terminate for all inputs.

Research on the improvement of elimination methods includes further consideration of techniques from SAT preprocessors, investigation of tableau and DPLL-like techniques [12, 2], and, in the context of first-order logic, the so called *direct methods* [1]. In addition, it seems worth to investigate further types of output: incremental construction, like enumeration of model representations, and representations of proofs.

So far, the system has been applied in teaching and to investigate logic programming semantics. Some application examples are provided on its Web page. One of them shows generalizations of several logic programming semantics that allow to exempt specified predicates from the closed world assumption [14], another one shows how skeptical abduction with respect to the stable models semantics and to the three-valued partial stable models semantics can be expressed and implemented on the basis of the globally weakest sufficient condition. A third example includes a small toy knowledge base about a touristic real-world scenario to illustrate a range of further applications in a rudimentary way: Extraction of knowledge concerning a given signature and of knowledge at a particular level of abstraction, as well as certain forms of schema mapping, abduction, intensional answers, knowledge base modularization and data protection.

The approach of computation with logic by elimination leads to a system that provides a uniform user interface covering many tasks, like satisfiability checking, computation of abductive explanations and computation of models for various logic programming semantics. Variants of established concepts can be easily expressed on a clean semantic basis and made operational. The approach supports the co-existence of different knowledge representation techniques in a single system, backed by a single classical semantic framework. This seems a necessary precondition for logic libraries that accumulate knowledge independently of some particular application.

References

1. D. M. Gabbay, R. A. Schmidt, and A. Szałas. *Second-Order Quantifier Elimination: Foundations, Computational Aspects and Applications*. College Publications, 2008.
2. J. Huang and A. Darwiche. DPLL with a trace: From SAT to knowledge compilation. In *IJCAI-05*, pages 156–162. Morgan Kaufmann, 2005.
3. A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.
4. J. Lang, P. Liberatore, and P. Marquis. Propositional independence – formula-variable independence and forgetting. *Journal of Artificial Intelligence Research*, 18:391–443, 2003.
5. V. Lifschitz. Circumscription. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 3, pages 298–352. Oxford University Press, 1994.
6. V. Lifschitz. Twelve definitions of a stable model. In *ICLP 2008*, volume 5366 of *LNCS*, pages 37–51. Springer, 2008.
7. F. Lin. *A Study of Nonmonotonic Reasoning*. PhD thesis, Stanford University, 1991.
8. F. Lin. On strongest necessary and weakest sufficient conditions. *Artificial Intelligence*, 128:143–159, 2001.
9. N. Manthey. Coprocessor 2.0 – a flexible CNF simplifier. In *SAT 2012*, volume 7317 of *LNCS*, pages 436–441. Springer, 2012.
10. N. V. Murray and E. Rosenthal. Tableaux, path dissolution and decomposable negation normal form for knowledge compilation. In *TABLEAUX 2003*, volume 2796 of *LNAI*, pages 165–180. Springer, 2003.

11. C. Wernhard. Literal projection for first-order logic. In *JELIA 08*, volume 5293 of *LNAI*, pages 389–402. Springer, 2008.
12. C. Wernhard. Tableaux for projection computation and knowledge compilation. In *TABLEAUX 2009*, volume 5607 of *LNAI*, pages 325–340. Springer, 2009.
13. C. Wernhard. Circumscription and projection as primitives of logic programming. In *Tech. Comm. of the ICLP 2010*, volume 7 of *LIPIcs*, pages 202–211, 2010.
14. C. Wernhard. Forward human reasoning modeled by logic programming modeled by classical logic with circumscription and projection. Technical Report Knowledge Representation and Reasoning 11-07, Technische Universität Dresden, 2011.
15. C. Wernhard. Projection and scope-determined circumscription. *Journal of Symbolic Computation*, 47:1089–1108, 2012.