Testing Strong Equivalence of Datalog Programs - Implementation and Examples*

Thomas Eiter[1], Wolfgang Faber[2,**], and Patrick Traxler[1]

[1] Institute of Information Systems, Vienna University of Technology, 1040 Vienna, Austria
eiter@kr.tuwien.ac.at, e0027287@student.tuwien.ac.at
[2] Department of Mathematics, University of Calabria, 87030 Rende (CS), Italy
faber@mat.unical.it

Abstract. In this work we describe a system for determining *strong equivalence* of disjunctive *non-ground* datalog programs under the stable model semantics. The problem is tackled by reducing it to the unsatisfiability problem of first-order formulas in the Bernays-Schönfinkel fragment. We then employ a tableaux-based theorem prover, which (unlike most other currently available provers) is guaranteed to terminate for these formulas. To the best of our knowledge, this is the first strong equivalence tester for disjunctive non-ground datalog.

1 Introduction

Answer Set Programming (ASP) [1] is by now an acknowledged tool for knowledge representation and reasoning. The availability of efficient solvers has furthermore stimulated its use in practical applications in recent years. This development had quite some implications on ASP research. For example, increasingly large applications require features for modular programming. Another requirement is the fact that in applications, ASP code is often generated automatically by so-called frontends, calling for optimization methods which remove redundancies, as also found in database query optimizers. For these purposes, the more recently suggested notion of strong equivalence for programs [2,3] can be used. Indeed, if two ASP programs are strongly equivalent, they can be used interchangeably in any context. This gives a handle on showing the equivalence of ASP modules. If a program is strongly equivalent to a subprogram of itself, then one can always use the subprogram instead of the original program, a technique which serves as an effective optimization method.

So far, work on strong equivalence has mostly focused on propositional, or variable-free programs. The complexity of deciding whether two variable-free datalog programs are strong equivalent is in co-NP [4], however, when admitting variables, we obtain completeness for co-NEXPTIME [5]. Several systems have been proposed for testing strong equivalence of variable-free programs, some of which encode the problem again in ASP (e.g. [6]) or in propositional satisfiability [7,4].

In this work, we build on [4] and use a variant of the reduction described there, which in the non-ground case produces first-order formulas in the Bernays-Schönfinkel class which are unsatisfiable iff the original logic programs are strongly equivalent.

* This work was supported by FWF under project P18019-N04 and the European Commission under projects IST-2001-37004 WASP and IST-2001-33570 INFOMIX.
** Funded by an APART grant of the Austrian Academy of Sciences.

2 Preliminaries

Disjunctive Datalog Programs. A *(disjunctive) rule* r is a formula

$$a_1(\bar{x}_1)\vee\cdots\vee a_n(\bar{x}_n) \;\texttt{:-}\; b_1(\bar{y}_1),\cdots,b_k(\bar{y}_k),\texttt{not}\; b_{k+1}(\bar{y}_{k+1}),\cdots,\texttt{not}\; b_m(\bar{y}_m). \quad (1)$$

$n \geq 0$, $m \geq k \geq 0$, where all $a_i(\bar{x}_i)$ and $b_j(\bar{y}_j)$ are function-free atoms; if $n = 0$, r is also called a *constraint*. A *disjunctive datalog program* $\mathcal{P}$ is a finite set of rules and constraints. Two programs Π_1 and Π_2 are *strongly equivalent* [2] iff every program extensions $\Pi_1 \cup R$ and $\Pi_2 \cup R$ have the same answer sets [1].

Bernays-Schönfinkel Fragment of First-Order Logic. Any first-order sentence ψ of form

$$\exists x_1 ... x_k \forall y_1 ... y_l \varphi(x_1, ..., x_k, y_1, ..., y_l) \quad (2)$$

where φ is quantifier-free and without function and constant symbols, is a *Bernays-Schönfinkel formula*. Deciding satisfiability of such formulas is NEXPTIME-complete.

3 Reduction

In this section, we describe a reduction from the complementary problem of strong equivalence to satisfiability of Bernays-Schönfinkel formulas (whose quantifier-free part is in CNF), which is similar to the reduction defined in [4].

Given two logic programs Π and Π', let for each predicate p occurring in $\Pi \cup \Pi'$, be p' a fresh predicate of the same arity. Then

$$\Sigma(\bar{x}) := \textstyle\bigwedge_{p \in Pred(\Pi \cup \Pi')}(p'(\bar{x}) \vee \neg p(\bar{x}))$$

For any rule r of the form (1), we define γ_r as the formula ($\bar{z} = \bar{x}_1 \cdots \bar{x}_n \bar{y}_1 \cdots \bar{y}_m$):

$$\forall \bar{z} \begin{pmatrix} (a_1(\bar{x}_1)\vee\cdots\vee a_n(\bar{x}_n) \vee b'_{k+1}(\bar{y}_{k+1})\vee\cdots\vee b'_m(\bar{y}_m) \vee \neg b_1(\bar{y}_1)\vee\cdots\vee\neg b_k(\bar{y}_k)) \\ \bigwedge \\ (a'_1(\bar{x}_1)\vee\cdots\vee a'_n(\bar{x}_n) \vee b'_{k+1}(\bar{y}_{k+1})\vee\cdots\vee b'_m(\bar{y}_m) \vee \neg b'_1(\bar{y}_1)\vee\cdots\vee\neg b'_k(\bar{y}_k)) \end{pmatrix}$$

For a program Π, we then define $\Gamma_\Pi := \bigwedge_{r \in \Pi} \gamma_r$, which we can easily rewrite to $\forall \bar{x} W_\Pi(\bar{x})$ where $W_\Pi(\bar{x})$ is a quantifier-free CNF. We next define a formula encoding the unique name assumption for the constants $\bar{c} = c_1, ..., c_n$ occurring in Π and Π':

$$U := \textstyle\bigwedge_{i=1}^{n}(U_i(c_i) \wedge \bigwedge_{j \in \{1,...,n\} \setminus \{i\}} \neg U_i(c_j)).$$

For a formula φ, let φ^x_y be the formula with y replaced by x, and for a set S of formulas, let $S^x_y = \{\varphi^x_y \mid \varphi \in S\}$. As shown in [4], Π and Π' are *not* strongly equivalent iff at least one of the following two Bernays-Schönfinkel sentences is finitely satisfiable:

- $\exists \bar{u} \exists \bar{y} \forall \bar{z} \forall \bar{x} (U^{\bar{u}}_{\bar{c}}(\bar{u}) \wedge \Sigma(\bar{z}) \wedge W_\Pi{}^{\bar{u}}_{\bar{c}}(\bar{x}, \bar{u}) \wedge \neg W_{\Pi'}{}^{\bar{u}}_{\bar{c}}(\bar{y}, \bar{u}))$, resp.
- $\exists \bar{u} \exists \bar{y} \forall \bar{z} \forall \bar{x} (U^{\bar{u}}_{\bar{c}}(\bar{u}) \wedge \Sigma(\bar{z}) \wedge W_{\Pi'}{}^{\bar{u}}_{\bar{c}}(\bar{x}, \bar{u}) \wedge \neg W_\Pi{}^{\bar{u}}_{\bar{c}}(\bar{y}, \bar{u}))$.

(By the finite model property of Bernays-Schönfinkel, this is tantamount to unrestricted satisfiability.) Note that U, Σ, W_Π, and $W_{\Pi'}$ are CNFs, while $\neg W_\Pi$ and

$\neg W_{\Pi'}$ are not (moving negation inside, they are in DNF). Instead of the simple conversion to CNF, which is exponential in the worst case, we may for our purpose replace them with CNFs W^*_{Π} and $W^*_{\Pi'}$, respectively, which are equivalent with respect to satisfiability. To this end, we use the following transformation of a quantifier-free DNF $D(\bar{x}) = \bigvee_{i=1}^{n} \tau_i(\bar{x}_i)$ with free variables $\bar{x} = \bar{x}_1 \cdots \bar{x}_n$, where $\tau_i(\bar{x}_i) = l_{i,1}(\bar{x}_{i,1}) \wedge \cdots \wedge l_{i,m_i}(\bar{x}_{i,m_i})$ into a CNF $D^*(\bar{x})$ which is satisfiability-equivalent if the $\bar{x}_i$ and $\bar{x}_j$ are pairwise disjoint:

$$D^*(\bar{x}) = (s(d_1) \vee \cdots \vee s(d_n)) \wedge \bigwedge_{1\leq i\leq n} \bigwedge_{1\leq j\leq m_i} (l_{i,j}(\bar{x}_{i,j}) \vee \neg s(d_j))$$

where s is a new unary predicate symbol and $d_1, \ldots, d_n$ are fresh constant symbols.

Lemma 1. *$\forall\bar{x}D(\bar{x})$ is satisfiable iff $\forall\bar{x}D^*(\bar{x})$ is satisfiable, if all $\bar{x}_i$ and $\bar{x}_j$ are disjoint.*

We note that the size of $(\neg W_{\Pi})^*$ is linear in the size of Π, since it is linear in the size of W_{Π}, which in turn is linear in the size of Π. Let n_r and n_c be the number of predicate and constant symbols, respectively, in Π and Π'. Then, the size of Σ is linear in n_r and the size of U is quadratic in n_c.

Let $\Delta(\Pi, \Pi')$ denote the *clausal form* of $\exists\bar{u}\exists\bar{y}\forall\bar{z}\forall\bar{x}(U^{\bar{u}}_{\bar{c}}(\bar{u}) \wedge \Sigma(\bar{z}) \wedge W_{\Pi}{}^{\bar{u}}_{\bar{c}}(\bar{x}, \bar{u}) \wedge (\neg W_{\Pi'}(\bar{y}){}^{\bar{u}}_{\bar{c}}(\bar{y}, \bar{u}))^*)$ after Skolemization, i.e., the set of clauses in $(U \wedge \Sigma(\bar{z}) \wedge W_{\Pi}(\bar{x}) \wedge (\neg W_{\Pi'}(f\bar{y}, f\bar{u}))^*)$. It can be easily generated, and its size is bounded as follows.

Proposition 1. *$|\Delta(\Pi, \Pi')| \leq k \cdot (|\Pi| + |\Pi'| + n_r + n_c^2)$ for some constant k.*

4 Implementation

The input language is similar to the one of DLV, but add-ons like built-ins, aggregates, weak constraints etc. are not supported. Also comments and anonymous variables are currently unsupported, as well as strong negation.[3]

The implementation is in `C++` employing a `flex/bison`-generated parser. Our basic data structures include a symbol table and a collection of rules. The generation of the clausal forms $\Delta(\Pi, \Pi')$ and $\Delta(\Pi', \Pi)$ is carried out via suitable functions working on these basic structures. We use the DARWIN theorem prover[4] as a back-end to solve the formulas. A distinguishing feature of DARWIN is that it is refutation-complete on our types of formulas, and thus strong equivalence of programs Π and Π', tantamount to refutations of $\Delta(\Pi, \Pi')$ and $\Delta(\Pi', \Pi)$, is definitely answered in all cases. Indeed, we are not aware of other provers which would guarantee this property.

The tool (including some examples) is available at `http://www.kr.tuwien.ac.at/students/prak_setest/`.

5 Examples

Example 1. Consider the program

```
Π: a(k1). a(k2).
   h(X):- a(X).  t(X):- h(X).  a(X):- t(X).   a(X):- h(X).
```

[3] In fact, strong negation $\neg a(\bar{x})$ is realized in DLV and other systems viewing $\neg a$ as a new predicate and adding a constraint $:\!-a(\bar{x}), \neg a(\bar{x})$; this can be respected and handled accordingly.

[4] http://goedel.cs.uiowa.edu/Darwin/

Π states that $a \subseteq h \subseteq t \subseteq a$, i.e. $a = h = t$. By dropping the last rule, we obtain
Π': `a(k1). a(k2). h(X):- a(X). t(X):- h(X). a(X):- t(X).`

The components of the formula $\Delta(\Pi, \Pi')$ are, in Darwin syntax, as follows.

Σ:
```
a_(X1):-a(X1).  t_(X1):-t(X1).  h_(X1):-h(X1).
```

W_{Π}:
```
a(k1). a(k2).    a_(k1). a_(k2).
h(X):- a(X).   h_(X):- a_(X).    t(X):- h(X).   t_(X):- h_(X).
a(X):- t(X).   a_(X):- t_(X).    a(X):- h(X).   a_(X):- h_(X).
```

$W^*_{\Pi'}$:
```
-a(k1):-s__(1).     -a_(k1):- s__(6).   -a(k2):-s__(2).    -a_(k2):-s__(7).
-h(sk_1):-s__(3).   -h_(sk_4):-s__(8).  a(sk_1):-s__(3).   a_(sk_4):-s__(8).
-t(sk_2):-s__(4).   -t_(sk_5):-s__(9).  h(sk_2):-s__(4).   h_(sk_5):-s__(9).
-a(sk_3):-s__(5).   -a_(sk_6):-s__(0).  t(sk_3):-s__(5).   t_(sk_6):-s__(0).
s__(1), s__(2), s__(3), s_(4), s__(5), s__(6), s__(7), s__(8), s__(9), s__(0).
```

U: `u1(k1). -u1(k2). -u2(k1). u2(k2).`

A refutation is found by Darwin for $\Delta(\Pi, \Pi')$, and also for $\Delta(\Pi', \Pi)$. Hence, Π' and Π are strongly equivalent.

Example 2. Consider the two programs

```
Π: t(X,Y):-a(X,Y).                  Π′: t(X,Y):-a(X,Y).
   t(X,Z):-t(X,Y),t(Y,Z).               t(X,Z):-a(X,Y),t(Y,Z).
```

which both compute the transitive closure of `a`. They are, however, not strongly equivalent, since $\Pi \cup \{t(1,2)., t(2,3).\}$ and $\Pi' \cup \{t(1,2)., t(2,3).\}$ have different answer sets. $\Delta(\Pi, \Pi')$ is unsatisfiable and $\Delta(\Pi', \Pi)$ is satisfiable, reflecting this fact.

6 Benchmarks

When experimenting with our tool, we have found that it often terminates quickly (less than one second), for instance for the examples presented in the previous section or for pairs of programs which differ substantially. We have been looking for parametric benchmark examples which create formulas that are increasingly hard to solve. These examples should be (1) scalable and (2) sufficiently similar to each other. We find that the following example interesting in this respect:
Example 3. (n-Layer TC Programs) Let Π_n be the following n-layer transitive closure:

```
t1(X,Y):- r(X,Y).      t1(X,Y):- r(X,Z), t1(Z,Y).
t2(X,Y):- t1(X,Y).     t2(X,Y):- t1(X,Z), t2(Z,Y).
   ...
tn(X,Y):- tn−1(X,Y).   tn(X,Y):- tn−1(X,Z), tn(Z,Y).
```

Π'_n is similar but with one additional redundant rule for each layer except the first:

```
t1(X,Y):- r(X,Y).       t1(X,Y):- r(X,Z), t1(Z,Y).
t2(X,Y):- t1(X,Y).      t2(X,Y):- t1(X,Z), t2(Z,Y).
t2(X,Y):- r(X,Z), t2(Z,Y).
   ...
tn(X,Y):- tn−1(X,Y).    tn(X,Y):- tn−1(X,Z), tn(Z,Y).
tn(X,Y):- r(X,Z), tn(Z,Y).
```

Table 1. Run-times for n-layer transitive closure

n	10	20	30	40	50	60
CPU time (sec)	0.38	1.43	3.66	7.93	13.55	23.56

The programs Π_n and Π'_n are strongly equivalent. We have tested this for various n on an 800MHz PowerPC with 1GB main memory, running GNU/Linux and DARWIN in version 08-27-04. The results are shown in Table 1. We can observe that the runtimes roughly double when increasing n by 10. Thus the scaling shows exponential behavior. On the other hand, viewed from computational complexity the n-layer TC is not among the "hard" instances of the problem; such hard instances could be systematically generated from complexity proofs.

We have also considered variants of these programs, where we added the rule

```
in(X,Y) ∨ out(X,Y) :- tn.
```

to Π_n, arriving at $\Pi_n^{final,\vee}$. To Π_n, we added the two body-shift variants of this rule

```
in(X,Y) :-tn(X,Y), not out(X,Y).  out(X,Y) :-tn(X,Y), not in(X,Y).
```

to obtain $\Pi_n^{final,\neg}$. The programs $\Pi_n^{final,\vee}$ and $\Pi_n^{final,\neg}$ are not strongly equivalent, and our tool was always very fast (less than one second) to decide this. Finding testcases which are not strongly equivalent and hard for our tool remains as an open issue.

7 Conclusion

We have implemented a non-ground strong equivalence tester, which works by a reduction to unsatisfiability of Bernays-Schönfinkel formulas, which are solved by the theorem prover Darwin, which is guaranteed to terminate on these formulas. The size of the resulting Darwin programs is nearly linear in the size of the input programs. Hence, the overall performance of testing strong equivalence depends heavily on the automated theorem prover. We have made a positive experience with our tool. We could find a class of problems which is apparently hard for the employed prover. With similar examples that are not strongly equivalent, its performance was, however, very good.

References

1. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing **9** (1991) 365–385
2. Lifschitz, V., Pearce, D., Valverde, A.: Strongly Equivalent Logic Programs. ACM Transactions on Computational Logic **2** (2001) 526–541
3. Turner, H.: Strong Equivalence Made Easy: Nested Expressions and Weight Constraints. Theory and Practice of Logic Programming **3** (2003) 602–622
4. Lin, F.: Reducing Strong Equivalence of Logic Programs to Entailment in Classical Propositional Logic. In: Proc. KR-2002. 170–176
5. Eiter, T., Faber, W., Greco, G., Fink, M., Lembo, D., Tompits, H., Woltran, S.: Methods and Techniques for Query Optimization. TR D5.3, EC Project IST-2001-33570 (INFOMIX) (2004) Available at `http://www.mat.unical.it/infomix/`.
6. Eiter, T., Fink, M., Tompits, H., Woltran, S.: Simplifying logic programs under uniform and strong equivalence. In Proc. LPNMR-7, Springer (2004) 87–99
7. Pearce, D., Tompits, H., Woltran, S.: Encodings for Equilibrium Logic and Logic Programs with Nested Expressions. In Proc. EPIA 2001. 306–320