Testing Strong Equivalence of Datalog Programs - Implementation and Examples

Ausarbeitung im Zuge der LVA
10.0 PR Wahlfachpraktikum

Patrick Traxler
Matrikelnummer 0027287

Betreuer:
Dr. Wolfgang Faber
o. Univ.-Prof. Dr. Thomas Eiter

Abteilung für Wissensbasierte Systeme (E184-3),
TU Wien

9. September 2004

Abstract

In this work *strong equivalence* of disjunctive first order datalog programs under the stable model semantic is considered. The problem is reduced to the unsatisfiability problem of Bernays-Schönfinkel formulas. An implementation is described in detail.

Contents

1 Introduction

Two disjunctive datalog programs Π_1 and Π_2 are strongly equivalent iff for every set of rules R the programs $\Pi_1 \cup R$ and $\Pi_2 \cup R$ are equivalent, i.e. iff they have the same stable models [2]. The programs may have variables, and the universe is countable. Strong equivalence is used to optimize programs. For a given program Π and a rule r in Π it is tested if $\Pi - \{r\}$ is strongly equivalent to Π. If so, $\Pi - \{r\}$ might be used instead of Π. Testing strong equivalence is complete for coNEXPTIME [5].

In [4] it was shown how to reduce the problem of testing strong equivalence to the unsatisfiability problem of Bernays-Schönfinkel formulas. This reduction is refined in section 3. In section 4 an implementation is described. Two DLV programs Π_1 and Π_2 are translated into two Darwin clausal forms φ_1 and φ_2 such that Π_1 and Π_2 are strongly equivalent iff φ_1 and φ_2 are unsatisfiable. DLV[1] is an answer set solver and Darwin[2] is an automated theorem prover. In [1] it was pointed out that the model evolution calculus terminates for Bernays-Schönfinkel formulas. Darwin implements this calculus.

In section 5 some examples are studied.

2 Preliminaries

Disjunctive Datalog Programs Let l_i denote a *literal*, i.e. a first order atom or its negation. A *disjunctive datalog program* (program for short) is a set of *rules* where each rule has the form

$$l_1, ..., l_k \leftarrow l_{k+1}, ..., l_m, \text{not} l_{m+1}, ..., \text{not} l_n$$

For a rule r

- the *head* $H(r)$ is $l_1, ..., l_k$,
- the *body* $B(r)$ is $l_{k+1}, ..., l_m, \text{not} l_{m+1}, ..., \text{not} l_n$,
- the *positive body* $B^+(r)$ is $l_{k+1}, ..., l_m$, and
- the *negative body* $B^-(r)$ is $l_{m+1}, ..., l_n$.

A *constraint* is a rule with an empty head, and a *disjunctive fact* is a rule with an empty body. A program may contain constraints or disjunctive facts. If the head consists of exactly one literal, and the body is not empty then the rule is called *normal*, and if the negative body is empty then the rule is called *positive*. A program is called *normal* (resp., *positive*) if every rule is normal (resp., positive).

Note, that a program has no function symbols with positive arity. Constant symbols are allowed.

[1] www.dlvsystem.com
[2] www.mpi-sb.mpg.de/~baumgart/DARWIN/

A Herbrand interpretation consists of ground literals. A ground literal is a possibly negated relational symbol with constant symbols from the Herbrand base as arguments. The Herbrand base is a finite subset of the countable universe which contains all constants of the considered program. A Herbrand interpretation I is *consistent* iff there are no two literals in I where one of them is the negation of the other. Let $Gr(\Pi)$ be the grounding of Π. A consistent set of literals M is *closed under* Π iff for all $r \in Gr(\Pi)$:

$$\text{if } B^+(r) \subseteq M,\ B^-(r) \cap M = \{\} \text{ then } H(r) \cap M \neq \{\}.$$

A *stable model* of a positive program Π is a minimal consistent total Herbrand interpretation which is closed under Π. For a Herbrand interpretation I of Π

$$\Pi^I := \{H(r) \leftarrow B^+(r) : B^-(r) \cap I = \{\}\}$$

is called the *Gelfond-Lifschitz reduct*. A *stable model* of Π is a stable model of Π^I. Let $\mathrm{SM}(\Pi)$ denote the stable models of Π.

Two programs Π_1 and Π_2 are *equivalent* iff they have the same stable models. They are *strongly equivalent* iff for every set R of rules $\Pi_1 \cup R$ and $\Pi_2 \cup R$ are equivalent.

In the following only $\neg$-free programs are considered. A program with $\neg$ can be translated into one without $\neg$; see [3].

Bernays-Schönfinkel Fragment of First Order Logic For a quantifier free formula φ without function and constant symbols a *Bernays-Schönfinkel formula* is of the form

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

For such a formula ψ the satisfiability problem is complete for NEXPTIME. The *clausal form* of ψ is obtained from ψ by replacing the variables $x_1, ..., x_k$ in φ with new constant symbols, transforming the resulting formula φ' into a conjunctive normal form φ'', and putting all clauses of φ'' in a set $\Sigma(\bar{x})$ where $\bar{x}$ are the free variables of φ''. The clausal form $\Sigma(\bar{x})$ is satisfiable iff ψ is satisfiable. In general, a clausal form $\Sigma(\bar{x})$ is said to be satisfiable iff $\forall \bar{x} \Sigma(\bar{x})$ is satisfiable. The notion

$$l_1, ..., l_k \vdash l_{k+1}, ..., l_n$$

stands for the clause

$$l_1 \vee ... \vee l_k \vee \neg l_{k+1} \vee ... \vee \neg l_n.$$

3 Reduction

Testing strong equivalence is coNEXPTIME complete [5]. Here, a reduction of *non* strong equivalence to the Bernays-Schönfinkel satisfiability problem in clausal form is described. This resembles the reduction in [4].

For every relational symbol r_i, $1 \leq i \leq k$, of the programs Π_1, Π_2 let r_i' be a new relational symbol of the same arity. Then

$$\Sigma(\bar{x}_1, ..., \bar{x}_k) := \{r_1'(\bar{x}_1) \vdash r_1(\bar{x}_1), ..., r_k'(\bar{x}_k) \vdash r_k(\bar{x}_k)\}.$$

For each rule

$$h_1(u_1); ...; h_l(u_l) \leftarrow p_1(v_1), ..., p_m(v_m), \mathrm{not} p_{m+1}(v_{m+1}), ..., \mathrm{not} p_n(v_n)$$

in Π, let $\Gamma_\Pi(\bar{u}, \bar{v})$ contain

$$h_1(u_1), ..., h_l(u_l) \vdash p_1(v_1), ..., p_m(v_m), \neg p_{m+1}'(v_{m+1}), ..., \neg p_n'(v_n)$$

and

$$h_1'(u_1), ..., h_l'(u_l) \vdash p_1'(v_1), ..., p_m'(v_m), \neg p_{m+1}'(v_{m+1}), ..., \neg p_n'(v_n).$$

Let U be the set of unique names axioms, i.e. U asserts that no two constants are equal. Therefore a new relational symbol U_i for every constant symbol c_i is introduced:

$$\begin{array}{l} U_1(c_1) \wedge \neg U_2(c_1) \wedge ... \wedge \neg U_k(c_k) \\ \neg U_1(c_1) \wedge U_2(c_1) \wedge ... \wedge \neg U_k(c_k) \\ \vdots \\ \neg U_1(c_l) \wedge \neg U_2(c_1) \wedge ... \wedge U_k(c_k) \end{array}$$

For a formula φ let φ_y^x be the formula with y replaced by x. Analogous for a set of formulas. Let $\bar{c}$ be $c_1, ..., c_k$. In [4] it was shown that Π and Π' are strongly equivalent iff

$$\forall \bar{u} \forall \bar{y} \exists \bar{x} \exists \bar{z} (\neg U_{\bar{c}}^{\bar{u}} \vee \neg \Sigma(\bar{z}) \vee \neg \Gamma_\Pi(\bar{x})_{\bar{c}}^{\bar{u}} \vee \Gamma_{\Pi'}(\bar{y})_{\bar{c}}^{\bar{u}})$$

and

$$\forall \bar{u} \forall \bar{y} \exists \bar{x} \exists \bar{z} (\neg U_{\bar{c}}^{\bar{u}} \vee \neg \Sigma(\bar{z}) \vee \neg \Gamma_{\Pi'}(\bar{x})_{\bar{c}}^{\bar{u}} \vee \Gamma_\Pi(\bar{y})_{\bar{c}}^{\bar{u}})$$

are valid. Therefore, Π and Π' are *not* strongly equivalent iff

$$\exists \bar{u} \exists \bar{y} \forall \bar{x} \forall \bar{z} (U_{\bar{c}}^{\bar{u}} \wedge \Sigma(\bar{z}) \wedge \Gamma_\Pi(\bar{x})_{\bar{c}}^{\bar{u}} \wedge \neg \Gamma_{\Pi'}(\bar{y})_{\bar{c}}^{\bar{u}})$$

or

$$\exists \bar{u} \exists \bar{y} \forall \bar{x} \forall \bar{z} (U_{\bar{c}}^{\bar{u}} \wedge \Sigma(\bar{z}) \wedge \Gamma_{\Pi'}(\bar{x})_{\bar{c}}^{\bar{u}} \wedge \neg \Gamma_\Pi(\bar{y})_{\bar{c}}^{\bar{u}})$$

is satisfiable.

The sets U, Σ, $\Gamma_\Pi(\bar{x})$, $\Gamma_{\Pi'}(\bar{y})$ are in clausal form, but $\neg\Gamma_\Pi(\bar{x})$, $\neg\Gamma_{\Pi'}(\bar{y})$ are not. They are in quantifier free disjunctive normal form. It is sufficient to use satisfiability equivalent formulas to gain clausal form. Denote these formulas by $\Gamma_\Pi^*(\bar{x})$, $\Gamma_{\Pi'}^*(\bar{y})$. It follows that Π and Π' are not strongly equivalent iff

$$\exists \bar{u} \exists \bar{y} \forall \bar{x} \forall \bar{z} (U_{\bar{c}}^{\bar{u}} \wedge \Sigma(\bar{z}) \wedge \Gamma_\Pi(\bar{x})_{\bar{c}}^{\bar{u}} \wedge \Gamma_{\Pi'}^*(\bar{y})_{\bar{c}}^{\bar{u}})$$

or

$$\exists \bar{u} \exists \bar{y} \forall \bar{x} \forall \bar{z} (U_{\bar{c}}^{\bar{u}} \wedge \Sigma(\bar{z}) \wedge \Gamma_{\Pi'}(\bar{x})_{\bar{c}}^{\bar{u}} \wedge \Gamma_\Pi^*(\bar{y})_{\bar{c}}^{\bar{u}})$$

is satisfiable. By Skolemization, this is the case iff

$$U \cup \Sigma(\bar{z}) \cup \Gamma_{\Pi}(\bar{x}) \cup \Gamma^*_{\Pi'}$$

or

$$U \cup \Sigma(\bar{z}) \cup \Gamma_{\Pi'}(\bar{x}) \cup \Gamma^*_{\Pi}$$

is satisfiable. In the following denote $U \cup \Sigma(\bar{z}) \cup \Gamma_{\Pi}(\bar{x}) \cup \Gamma^*_{\Pi'}$ by $\Delta(\Pi, \Pi')$.

Now, it is shown by an example how to get Γ^* from a quantifier free disjunctive normal form Γ. Let Γ be $A \vee (B \wedge C)$ and s be a new unary relational symbol. Then, Γ^* is $(A \vee s(c)) \wedge (B \vee \neg s(c)) \wedge (C \vee \neg s(c))$ where c does not occur in Γ. Therefore, $\forall \bar{x}(\Gamma(\bar{x}))$ is satisfiable iff $\forall \bar{x}(\Gamma^*(\bar{x}))$ is satisfiable. The algorithm for computing Γ^* from Γ in general follows.

Input: a quantifier free DNF Γ.
Output: a quantifier free CNF Γ^*, s.t. Γ^* is satisfiable iff Γ is satisfiable.

1. $\Gamma^* := \{\}$, and let s be a new unary relational symbol.
2. For every $t \in \Gamma$:
   Let t be $l_1 \wedge ... \wedge l_k$, and c be a new constant symbol.
   Add $l_1 \vee \neg s(c), ..., l_k \vee \neg s(c)$ to Γ^*.
   Set φ to $\varphi \vee s(c)$.
   Delete t from Γ.
3. Add φ to Γ^*.

Correctness. After construction, Γ^* is in quantifier free CNF. Assume that Γ^* is satisfiable. Since $\varphi \in \Gamma^*$, it holds that $s(c)$ is true for some c. It follows that the $l_1, ..., l_k$ which were added in the step when c was used are true, i.e. $l_1 \wedge ... \wedge l_k$ is true, and therefore Γ is satisfiable. On the other hand, let M be a model of Γ. There exists a true term in Γ for which a constant symbol c in the construction of Γ^* was used. Let M^* be the structure obtained from M such that $s(c)$ holds in M^*. Since the used constants do not occur in Γ it holds that M^* is a model of Γ^*. $\square$

Note, that no function symbols of positive arity were introduced.

4 Implementation

The syntax of DLV and Darwin is described first. The implementation of the reduction is described next.

DLV and Darwin syntax The supported DLV syntax in EBNF:

```
rule ::= head '.'  | head ':-' body '.'  | ':-' body '.'
head ::= literal {'v' literal}
body ::= literal {',' literal}
literal ::= atom | 'not' atom
atom ::= symb ['(' term {',' term}')']
term ::= variable | symb
```

```
variable ::= ('A'-'Z' ) {'a'-'z' | 'A'-'Z' | '0'-'9'}
symb ::= ('a'-'z' | '0'-'9') {'a'-'z' | 'A'-'Z' | '0'-'9'}
```

Let L_{DLV} be the language given by this syntax description.

A `rule` can be a disjunctive fact, a (disjunctive) rule or a constraint. The non-terminal `enl` stands for explicitly negated literal.

The used Darwin syntax in EBNF:

```
clause ::= head [':-' body].
head ::= literal {';' literal}
body ::= literal {',' literal}
literal ::= [('-' | '~')] atom | 'true' | 'false'
atom ::= symb | symb '(' term {',' term} ')' | '(' term '=' term ')'
term ::= variable | symb ['(' term {',' term} ')']
variable ::= ('A'-'Z' | '_') {'a'-'z' | 'A'-'Z' | '0'-'9' | '_'}
symb ::= ('a'-'z' | '0'-'9') {'a'-'z' | 'A'-'Z' | '0'-'9' | '_'}
```

Let L_{Darwin} be the language given by this syntax description.

Remarks.

- Equality is not supported. There is no `=` symbol in L_{DLV}.
- Strong (or explicit) negation is not supported.
- Comments are not supported. There is no `%` symbol.
- Anonymous variables are not supported. There is no _ in L_{DLV}.
- The set of atoms in L_{DLV} is a proper subset of the set of atoms in L_{Darwin}.

Output size The sets Π, Π', U, Σ, ... are represented with the languages L_{DLV} and L_{Darwin}. Their size $|\cdot|$ is the number of symbols.

The size of Γ_Π is linear in $|\Pi|$. The size of Γ^*_Π is linear in the size of Γ_Π. Let n_c be the number of constant symbols, and n_r be the number of relational symbols in Π and Π'. The size of Σ is linear in n_r and the size of U is quadratic in n_c.

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

Classes The implementation is written in `C++`. There are two classes:

`class SymbolTable`: This class is used for storing constant and relational symbols. There are two symbol tables, namely `st_CS` and `st_RS`. They are initialized when parsing the input with `yyparse()`.

`void add(const string& s, int arity = 0)`: Adds the symbol identified by `s` which might have a arity.

`bool exists(const string& s)`: Returns true if the symbol identified by `s` exists, false otherwise.

`int getArity(const string& s)`: Returns the arity if `s` exists, `-1` otherwise.

`string newSymbol(const string& prefix, int arity = 0)`: Returns a symbol which does not occur in the symbol table. Note, that this function uses the fact that '_' does not occur in the used DLV syntax. Example:

1. call: `newSymbol("c")` returns `"c_1"`
2. call: `newSymbol("c")` returns `"c_2"`
3. call: `newSymbol("sk")` returns `"sk_3"`
4. call: `newSymbol("c")` returns `"c_4"`

`string iterate(int)`: Iteration through the elements in the symbol table. Usage:

```
SymbolTable st;
...
string s;
for (s = st.iterate(0); s != ""; s = st.iterate(1)) {
    ...
}
```

The first element is retrieved with `iterate(0)` and the subsequent elements with `iterate(1)`. If there is no more element then `""` is returned.

`class Rule`: This class represents a rule.

`bool nextRule(string& s)`: The next rule is extracted from `s` where `s` is a DLV program. Note, that this function does not check `s` to be a correct DLV program. The return value is true if there exists a next rule, false otherwise.

`void skolemize()`: Applys Skolemization to the rule, i.e. all variables are replaced by new constants with respect to `st_CS`. For example, the Skolem form of `"r(X,Y):-s(X),t(Y)"` is `"r(sk_1,sk_2):-s(sk_1),t(sk_2)"`.

`string nextLiteral()`: Returns the next literal, `":-"`, or `""`. For the rule `"r(X,Y):-s(X),t(Y)"` this function works as follows:

1. call: `"r(X,Y)"`
2. call: `":-"`
3. call: `"s(X)"`
4. call: `"t(Y)"`
5. call: `""`

Initialization The two symbol tables `st_CS` and `st_RS` have to be initialized. This is done with the function `yyparse()` which reads a DLV program, checks for the correct syntax, and adds all constant and relational symbols to `st_CS` and `st_RS`. The function `yyparse()` is automatically generated by `flex`[3], a parser generator.

Generating the Output To generate $\Delta(\Pi, \Pi') = U \cup \Sigma(z) \cup \Gamma_{\Pi}(x) \cup \Gamma^*_{\Pi'}$ the functions `genSigma`, `genU`, `genGamma(string prog)`, `genGammaStar(string prog)` are used. For example,

```
cout << genSigma() << genU() << genGamma(prog1) <<
        genGammaStar(prog2) << endl;
```

Before, `yyparse()` should have been applied to `prog1` and `prog2` since `yyparse()` generates the symbol tables `st_CS` and `st_RS`. These symbol tables are used in `genSigma()` and `genU()`. If `st_CS` or `st_RS` changes so does the output of `genSigma()` and `genU()`. The functions `genGamma()` and `genGammaStar()` use the class `Rule` to iterate through the DLV program and transform it.

The `main` function The function `int main(int argc, char** argv)` awaits two command line arguments, the names of the two DLV programs. The two programs are parsed with `yyparse()`, and read in. Next, two output files are generated, `out.1.tme` and `out.2.tme`. These files are provided to Darwin. If in both cases Darwin finds a refutation the DLV programs are strongly equivalent.

[3] www.gnu.org

5 Examples

Example 1 Let Π be

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

The program Π states that $a \subseteq h \subseteq t \subseteq a$, i.e. $a = h = t$. Therefore, the last rule is useless. Let Π' be

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

In the following the single parts of the resulting formula $\Delta(\Pi, \Pi')$ are given in Darwin syntax.

Σ:

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

$\Gamma(\Pi)$:

```
 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).
```

$\Gamma^*(\Pi')$:

```
 -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).
```

Using Darwin a refutation is found. This also holds for $\Delta(\Pi', \Pi)$. Hence, Π' and Π are strongly equivalent.

Example 2 Let Π be

```
b(X):- a(X).
c(X):- b(X).
```

and Π' be

```
c(X):- a(X).
b(X):- c(X).
```

These programs are equivalent, but not strongly equivalent since $\mathrm{SM}(\Pi \cup \{b(1).\}) \neq \mathrm{SM}(\Pi' \cup \{b(1).\})$. Darwin indeed says that $\Delta(\Pi, \Pi')$ is satisfiable.

Example 3 Let Π be

```
a(X):-not b(X).
a(X):-c(X).
c(X):-a(X).
c(X):-not b(X).
```

and Π' be

```
a(X):-not b(X).
a(X):-c(X).
c(X):-a(X).
```

These programs are strongly equivalent.

Example 4 Let Π be

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

and Π' be

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

Both programs compute the transitive closure. They are not strongly equivalent since $\mathrm{SM}(\Pi \cup \{t(1,2).,t(2,3).\}) \neq \mathrm{SM}(\Pi' \cup \{t(1,2).,t(2,3).\})$. Darwin says that $\Delta(\Pi, \Pi')$ is unsatisfiable and $\Delta(\Pi', \Pi)$ is satisfiable.

Example 5 Let Π be

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

and Π' be

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

```
t1(X,Y):- r(X,Z), t1(Z,Y).
```

These programs are strongly equivalent.

6 Conclusion

Testing strong equivalence is achieved by a reduction to the Bernays-Schönfinkel unsatisfiability problem. In the implementation two DLV programs are transformed into two Darwin programs. The size of the resulting Darwin programs is nearly linear in the size of the DLV programs. Hence, the overall performance of testing strong equivalence depends heavily on the automated theorem prover.

References

[1] P. Baumgartner, C. Tinelli: *The Model Evolution Calculus*, Fachberichte Informatik 1-2003, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz (2003).

[2] M. Gelfond, V. Lifschitz: *Classical negation in logic programs and disjunctive databases*, New Generation Computing (1991).

[3] V. Lifschitz et al.: *Strongly equivalent logic programs*, ACM Transactions on Computational Logic (2001).

[4] F. Lin: *Reducing Strong Equivalence of Logic Programs to Entailment in Classical Propositional Logic*, in D. Fensel, F. Giunchiglia, D. McGuiness, and M.-A. Williams, editors, Proceedings Eight International Conference on Principles of Knowledge Representation and Reasoning (KR-02), April 22-25, Toulouse, France, Morgan Kaufmann (2002).

[5] T. Eiter et al.: *Methods and Techniques for Query Optimization*, Infomix Consortium, Technical Report D5.3 (2004).