# A Reconstruction of the Lloyd-Topor Transformation using Partial Evaluation (Extended Abstract)*

Stephen-John Craig and Michael Leuschel

Institut für Informatik
Heinrich-Heine-Universität Düsseldorf
D-40225, Düsseldorf, Germany
{craig,leuschel}@cs.uni-duesseldorf.de

**Abstract.** The Lloyd-Topor transformation is a classical transformation that translates extended logic programs with logical connectives and explicit quantifiers into normal logic programs. In this paper we show that this translation can be achieved in a natural way by specialising a meta-interpreter for extended logic programs. For this we use the LOGEN partial evaluation system, extended to handle coroutining.

## 1 Vanilla Self-interpreter

An interesting application for partial evaluation [7] is the specialisation of interpreters. The object program to interpret is typically **static** and known at specialisation time, while the runtime goal remains **dynamic**. Partial evaluation can remove the overhead of interpretation, performing all the interpretation tasks at specialisation time and leaving behind a much more efficient "compiled" program. A classic interpreter is the vanilla self-interpreter (see, e.g., [1, 4]). Listing 1.1 contains a variation of the vanilla interpreter. The predicate `solve_literal/1` looks up the clause definition in the clause database and calls `solve/1` its body. Clauses are represented by a `head` and a conjunction of body literals. An auxiliary module (`prolog_reader`) is used to load clauses from a file, it defines `load_file/1` and `get_clause/2`. Listing 1.1 has the entry point `solve_file/2` which takes as arguments the name of the file containing the clauses and an entry goal.

**Listing 1.1.** A Vanilla self-interpreter for Prolog

```
:- use_module(prolog_reader).
solve(true).
solve(',' (A,T)) :- solve(A), solve(T).
solve(A) :- nonvar(A), A\= ','(_,_), solve_literal(A).

solve_literal(A) :- prolog_reader:get_clause(A,B), solve(B).
solve_file(File,A) :- prolog_reader:load_file(File),solve_literal(A).
```

While it may look like a simple program it still presents enough challenges for partial evaluation and has attracted considerable attention (see, e.g., [8, 14, 17]). In recent work [9] (see also [18]) we have shown how one can effectively specialise this interpreter using an offline specialiser (such as LOGEN [10]), achieving so-called *Jones optimality* [6, 13]. A key ingredient for success was the careful annotation of the interpreter, using the *nonvar* annotation which indicates that the top-level function symbol of some argument is known.

* This research is being carried out as part of the EU funded project IST-2001-38059 ASAP (Advanced Specialization and Analysis for Pervasive Systems).

Indeed, the argument to `solve_literal/1` cannot be marked as **static** (as it can contain variables), and marking it as **dynamic** would mean that no useful specialisation could be achieved. In the rest of this paper we build upon this to reconstruct the classical Lloyd-Topor transformation by specialising an interpreter derived from Listing 1.1.

## 2 The Lloyd Topor Transformation

Lloyd and Topor [11, 12] introduced extended programs and goals for logic programming. The extended programs can contain clauses that have an arbitrary first-order formula in their body. The only requirement for executing the transformed programs is a sound form of the negation as failure rule. In [11, 12] an extended program $P$ is transformed into a normal program $P'$, called the *normal form* of $P$, using a set of transformation rules. The rules **(a)** . . . **(j)** are applied until no more transformations can be applied. [11, 12] proves this process terminates and always gives a normal program.

**(a)** Replace $A \leftarrow \alpha \ \wedge \neg(V \wedge W) \wedge \ \beta$
by $A \leftarrow \alpha \ \wedge \neg V \wedge \ \beta$ and $A \leftarrow \alpha \ \wedge \neg W \wedge \ \beta$

**(b)** Replace $A \leftarrow \alpha \ \wedge \forall x_1 \dots x_n W \wedge \ \beta$ by $A \leftarrow \alpha \ \wedge \neg\exists x_1 \dots x_n \neg W \wedge \ \beta$

**(c)** Replace $A \leftarrow \alpha \ \wedge \neg\forall x_1 \dots x_n W \wedge \ \beta$ by $A \leftarrow \alpha \ \wedge \exists x_1 \dots x_n \neg W \wedge \ \beta$

**(d)** Replace $A \leftarrow \alpha \ \wedge V \leftarrow W \wedge \ \beta$
by $A \leftarrow \alpha \ \wedge V \wedge \ \beta$ and $A \leftarrow \alpha \ \wedge \neg W \wedge \ \beta$

**(e)** Replace $A \leftarrow \alpha \ \wedge \neg(V \leftarrow W) \wedge \ \beta$ by $A \leftarrow \alpha \ \wedge W \wedge \neg V \wedge \ \beta$

**(f)** Replace $A \leftarrow \alpha \ \wedge (V \vee W) \wedge \ \beta$ by $A \leftarrow \alpha \ \wedge V \wedge \ \beta$ and $A \leftarrow \alpha \ \wedge W \wedge \ \beta$

**(g)** Replace $A \leftarrow \alpha \ \wedge \neg(V \vee W) \wedge \ \beta$ by $A \leftarrow \alpha \ \wedge \neg V \wedge \neg W \wedge \ \beta$

**(h)** Replace $A \leftarrow \alpha \ \wedge \neg\neg W \wedge \ \beta$ by $A \leftarrow \alpha \ \wedge W \wedge \ \beta$

**(i)** Replace $A \leftarrow \alpha \ \wedge \exists x_1 \dots x_n W \wedge \ \beta$ by $A \leftarrow \alpha \ \wedge W \wedge \ \beta$

**(j)** Replace $A \leftarrow \alpha \ \wedge \neg\exists x_1 \dots x_n W \wedge \ \beta$
by $A \leftarrow \alpha \ \wedge \neg p(y_1, \dots, y_k \wedge \ \beta$ and $p(y_1, \dots, y_k) \leftarrow \exists x_1 \dots x_n W$
where $y_1, \dots, y_k$ are the free variables in $\exists x_1 \dots x_n W$ and $p$ is a new predicate not already appearing in the program.

For example, take the definition of subset expressed as an extended program [11, 12]:

$$x \subseteq y \leftarrow \forall u(u \in y \leftarrow u \in x)$$

The definition is written in a clear mathematical way and has been expressed in a form similar to the specification of the problem. However this specification cannot be executed directly in Prolog as it contains $\forall$ and $\leftarrow$ in its body. Transforming the extended program using the transformation rules produces the following normal program:

$$x \subseteq y \leftarrow \neg p(x, y)$$

$$p(x, y) \leftarrow \neg(u \in y) \wedge u \in x$$

This program requires a sound implementation of negation as failure, such as the one provided in [5]. As most Prolog systems do not provide such a negation, we can implement a sound selection rule ourselves by delaying negative literals until they have become ground using `when/2`. The so-obtained normal program, rewritten in standard Prolog syntax is given in Listing 1.2 (also containing sample executions). The $\in$ operator has been replaced by calls to `mem/2`. Below we show how this transformed program can be obtained automatically by specialising an interpreter for extended programs.

**Listing 1.2.** Prolog version of normal subset program using coroutining

```
subset(X,Y) :- when(ground([Y,X]),\+p(Y,X)).
p(X,Y) :- when(ground([A,X]),\+mem(A,X)), mem(A,Y).
mem(A, [A|_]).
mem(A, [_|B]) :- mem(A, B).

| ?- subset([a,b,c], S), S = [d,e,f,a,b,c].
S = [d,e,f,a,b,c] ?
yes
```

## 3 An Interpreter for Extended Programs

We now extend our vanilla interpreter to handle exstended programs as well as a more natural form of input programs. Note that this interpreter is *not* based on the Lloyd-Topor transformation. First, new operators are defined for implication and negation in the interpreter. The functions `forall/2` and `exists/2` are reserved for $\forall$ and $\exists$.

```
:- op(950,yfx,'=>').% implies right
:- op(950,yfx,'<='). % implies left
:- op(850,yfx,'or'). % or
:- op(800,yfx,'&').  % and
:- op(750,fy,'~').   % not
```

Two new predicates are created for handling body literals, a positive (`solve_literal/1`) and a negative one (`not_solve_literal/1`; introduced to overcome the problems of Prolog's unsound negation). The definition for `solve_literal/1` contains the basic clauses for dealing with `true` and `false`, if a *not* operator is encountered control is passed to `not_solve_literal/1`. The `&` operator performs a conjunction of the two arguments and the `or` operator performs a disjunction.

```
solve_literal(true).
solve_literal(false) :- fail.
solve_literal('~'(L)) :- not_solve_literal(L).
solve_literal('&'(A,B)) :- solve_literal(A), solve_literal(B).
solve_literal(or(A,_)) :- solve_literal(A).
solve_literal(or(_,B)) :- solve_literal(B).
solve_literal(A) :- is_user_pred(A),solve_atom(A).
solve_literal(A) :- is_built_in(A),call(A).

solve_atom(A) :- my_clause(A,B), solve(B).
```

The clauses for `not_solve_literal/1` are similar, but define the negated counterparts of `solve_literal/1`. Notice the handling of `&` and `or`, DeMorgan's laws ( $\neg(A \vee B) \equiv (\neg A \wedge \neg B)$ and $\neg(A \wedge B) \equiv (\neg A \vee \neg B)$) are applied and `solve_literal/1` is called. Coroutining is used to delay the negation using `when/2`.

```
not_solve_literal(true) :- solve_literal(false).
not_solve_literal(false) :- solve_literal(true).
not_solve_literal('~'(L)) :- solve_literal(L).
not_solve_literal(or(A,B)) :- solve_literal('&'('~'(A), '~'(B))).
not_solve_literal('&'(A,B)) :- solve_literal(or('~'(A), '~'(B))).
not_solve_literal(A) :- is_user_pred(A), not_solve_atom(A).
not_solve_literal(A) :- is_built_in(A),
    term_variables(A,Vars), when(ground(Vars), \+(call(A))).

not_solve_atom(A) :-
    term_variables(A,Vars), when(ground(Vars), \+(solve_atom(A))).
```

Implication is handled by transforming it into a disjunction (using $A \rightarrow B \equiv (\neg A \vee B)$ and $A \leftarrow B \equiv (A \vee \neg B)$) , which will in turn be handled by the previous clauses.

```
solve_literal('=>'(A,B)) :- solve_literal(or('~'(A),B)).
solve_literal('<='(A,B)) :- solve_literal(or(A,'~'(B))).

not_solve_literal('=>'(A,B)) :- solve_literal('~'(or('~'(A),B))).
not_solve_literal('<='(A,B)) :- solve_literal('~'(or(A,'~'(B)))).
```

The `exists(X,A)` operator is implemented by making a copy of the atom, `A`, and renaming all occurrences of `X`. This is done to avoid name clashes and `solve_literal/1` is called on the copy. The `forall(X,A)` is transformed into a `exists(X,~A)` and the negative version of `forall(X,A)` is transformed into a positive `exists` using standard logic laws. The negative version of `exists/2` must be handled directly, a `when/2` declaration is added to ensure the negation is only selected when the arguments are instantiated.

```
solve_literal(exists(X,A)) :- rename(X,A,CopyA),solve_literal(CopyA).
solve_literal(forall(X,A)) :- not_solve_literal(exists(X,'~'(A))).

not_solve_literal(forall(X,A)) :- solve_literal(exists(X,'~'(A))).
not_solve_literal(exists(X,A)) :-
    force_not_solve_literal(exists(X,A)).
force_not_solve_literal(G) :- get_free_variables(G,[],[],Vars),
    when(ground(Vars), \+(solve_literal(G))).
```

## 4 Specialising the Interpreter

In order to apply the LOGEN [10] offline specialiser we first had to extend it to handle the `when` declarations found in the above interpreter. Basically, we allow a `when` to be marked as **static**, i.e., the `when` will be executed at specialisation time and the guard must be satisfied at some point during the specialisation, or as **dynamic** in wich case the body of the declaration is specialised and then wrapped inside a `when` declaration.

**Listing 1.3.** Subset extended program in interpreter form

```
subset(A,B) :- forall(C,member(C,B)<=member(C,A)).
```

Specialising the interpreter for the subset extended program in Listing 1.3 produces the residual program Listing 1.4. The specialised program is almost identical to the program in Listing 1.2. All negations have been enclosed by `when/2`, this will ensure the negation will only be performed when the goal is properly ground. This transformation was performed by specialising an interpreter. Importantly the interpreter was not simply an encoding of the transformation rules from [12] but an intuitive interpreter for Prolog handling the extended program syntax.

**Listing 1.4.** Specialising the interpreter for the subset extended program (Listing 1.3)

```
solve_atom(subset(A,B)) :-  solve_atom__0(A, B).
solve_atom__0(A,B) :- when(ground([A,B]),\+solve_literal__1(A,B,_)).
solve_literal__1(A,B,_) :-
        when(ground([B,C]), \+member(C,B)), member(C,A).
```

Experimental results for the subset example can be found in Table 1, run on a 2.4 GHz Pentium 4 with 512 Mb memory, Gentoo 2.6 Linux and running SICStus 3.11.1.

| Benchmark | Original Program Size | Specialised Program Size | Relative Program Size |
|---|---|---|---|
| Lloyd Topor | 19877 bytes | 1685 bytes | 0.08 |

| Benchmark | Iterations | Specialisation Time | Original Runtime | Specialised Runtime | Speedup |
|---|---|---|---|---|---|
| Lloyd Topor | 100000 | 60ms | 8580ms | 2370ms | 3.57 |

**Table 1.** Benchmark figures for the Lloyd Topor interpreter

As the LOGEN system uses the cogen approach [10], we were able to produce a compiler which can be used to compile extended logic programs into ordinary Prolog programs from the command-line. For this we had to add an ability to LOGEN to feed command-line arguments into the specialization goal. In our case, the name of the file containing the extended Prolog program is passed as a static argument to the `solve_file/2` predicate (Listing 1.1). We have also extended the interpreter to handle command-line flags which allow to turn co-routining and debugging on and off; if co-routining is turned off we would get Listing 1.4 without `when` declarations. These programs cannot be (safely) run on most Prologs, but can be easier to read and resemble more directly the result of the Lloyd-Topor transformation. The debugging feature is useful as it presents debugging information in terms of the original extended program, and not in terms of the transformed program. More details, as well as more experimental results will be presented in the full paper.

We can now relate the output of the specialisation process more formally to the result of the Lloyd-Topor transformation. Let $E$ be an extended program. By $E \downarrow_{p/n}$ we denote the subset of the clauses of $E$ reachable in the predicate dependency graph from the predicate $p/n$.

**Proposition 1.** *Specialisation of the annotated interpreter for extended programs terminates for any extended object program $E$ and entry predicate $p/n$. After removing the when declarations, the residual program is identical (up to predicate and variable renaming) to the result obtained by the Lloyd-Topor transformation on $E \downarrow_{p/n}$.*

The proof proceeds by linking the transformation rules (a) – (i) to unfolding steps and the transformation rule (j) to a a memoisations step of the specialisation process.

## 5 Conclusion

We have shown how one can reconstruct the Lloyd-Topor transformation in a systematic way by specialising an interpreter for extended logic programs which uses coroutining.

In addition to providing an insight about this classic transformation, this provided an interesting test case for partial evaluation; stepping up from vanilla on to more challenging and practically useful interpreters. We have also obtained a command-line compiler for extended programs. An interesting direction for future work would be to build upon our work and develop an interpreter for ever more powerful languages, e.g., constructing a compiler for Gödel [2, 5], Curry [3] or Escher. It is also interesting to see whether other transformations (e.g., [15, 16]) can be obtained in a similar fashion by partial evaluation.

## References

1. K. R. Apt and F. Turini. *Meta-logics and Logic Programming.* MIT Press, 1995.
2. A. F. Bowers, P. M. Hill, and F. Ibañez. Resolution for logic programming with universal quantifiers. In H. G. Glaser, P. H. Hartel, and H. Kuchen, editors, *Proc. PLILP'97*, LNCS 1292, pages 63–77. Springer-Verlag, 1997.
3. M. Hanus. The integration of functions into logic programming. *The Journal of Logic Programming*, 19 & 20:583–628, May 1994.
4. P. Hill and J. Gallagher. Meta-programming in logic programming. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 421–497. Oxford University Press, 1998.
5. P. Hill and J. W. Lloyd. *The Gödel Programming Language.* MIT Press, 1994.
6. N. D. Jones. Partial evaluation, self-application and types. In M. S. Paterson, editor, *Automata, Languages and Programming*, LNCS 443, pages 639–659. Springer-Verlag, 1990.
7. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.
8. M. Leuschel. Homeomorphic embedding for online termination of symbolic methods. In T. Æ. Mogensen, D. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation*, LNCS 2?56, pages 379–403. Springer-Verlag, 2002.
9. M. Leuschel, S. Craig, M. Bruynooghe, and W. Vanhoof. Specializing interpreters using offline partial deduction. In M. Bruynooghe and K.-K. Lau, editors, *Program Development in Computational Logic*, LNCS 3049, pages 341–376. Springer-Verlag, 2004.
10. M. Leuschel, J. Jørgensen, W. Vanhoof, and M. Bruynooghe. Offline specialisation in Prolog using a hand-written compiler generator. *Theory and Practice of Logic Programming*, 4(1):139–191, 2004.
11. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.
12. J. W. Lloyd and R. W. Topor. Making Prolog more expressive. *Journal of Logic Programming*, 1(3):225–240, 1984.
13. H. Makholm. On Jones-optimal specialization for strongly typed languages. In W. Taha, editor, *Semantics, Applications, and Implementation of Program Generation*, LNCS 1924, pages 129–148. Springer-Verlag, 2000.
14. B. Martens. *On the Semantics of Meta-Programming and the Control of Partial Deduction in Logic Programming.* PhD thesis, K.U. Leuven, February 1994.
15. J. Moreno-Navarro and S. Muñoz. On the practical use of negation in a Prolog compiler. In V. Santos and E. Pontelli, editors, *Proceedings PADL'2000*, LNCS 1753, pages 124–140. ACM and ALP, Springer-Verlag, 2000.
16. S. Muñoz Hernándex, J. Mariño, and J. J. Moreno-Navarro. Constructive intensional negation. In Y. Kameyama and P. J. Stuckey, editors, *Proceedings FLOPS'04*, LNCS 2998. Springer-Verlag, April 2004.
17. W. Vanhoof and B. Martens. To parse or not to parse. In N. Fuchs, editor, *Proceedings LOPSTR'97*, LNCS 1463, pages 322–342, Leuven, Belgium, July 1997.
18. Q. Wang, G. Gupta, and M. Leuschel. Towards provably correct code generation via horn logical continuation semantics. In M. V. Hermenegildo and D. Cabeza, editors, *Proceedings PADL'05*, LNCS 3350, pages 98–112. Springer, 2005.