# LIX: An Effective Self-applicable Partial Evaluator for Prolog

Stephen-John Craig and Michael Leuschel

Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton, SO17 1BJ, UK
sjc02r,mal@ecs.soton.ac.uk

**Abstract.** This paper presents a self-applicable partial evaluator for a considerable subset of full Prolog. The partial evaluator is shown to achieve non-trivial specialisation and be effectively self-applied. The attempts to self-apply partial evaluators for logic programs have, of yet, not been all that successful. Compared to earlier attempts, our LIX system is practically usable in terms of efficiency and can handle natural logic programming examples with partially static data structures, built-ins, side-effects, and some higher-order and meta-level features such as `call` and `findall`. The LIX system is derived from the development of the LOGEN compiler generator system. It achieves a similar kind of efficiency and specialisation, but can be used for other applications. Notably, we show first attempts at using the system for deforestation and tupling in an offline fashion. We will demonstrate that, contrary to earlier beliefs, declarativeness and the use of the ground representation is not the best way to achieve self-applicable partial evaluators. **Keywords:** Partial Evaluation, Self-application, Logic Programming, Partial Deduction, Deforestation, Tupling.

## 1 Introduction and Summary

*Partial evaluation* has received considerable attention over the past decade both in functional (e.g. [16]), imperative (e.g. [2]) and logic programming (e.g. [9, 18, 25]). In the context of pure logic programs, partial evaluation is often referred to as *partial deduction*, the term partial evaluation being reserved for the treatment of impure logic programs. We will adhere to this convention in this paper.

Guided by the *Futamura projections* (see e.g. [16]) a lot of effort, especially in the functional partial evaluation community, has been put into making systems self-applicable. A partial evaluation or deduction system is called *self-applicable* if it is able to effectively[1] specialise itself. The practical interests of such a capability are manifold. The most well-known are related to the second and third *Futamura projections* [7]. The first Futamura projection consists of specialising an *interpreter* for a particular *object program*, thereby producing a specialised

[1] This implies some efficiency considerations, e.g. the system has to terminate within reasonable time constrains, using an appropriate amount of memory.

version of the interpreter which can be seen as a *compiled* version of the object program. If the partial evaluator is self-applicable then one can specialise the partial evaluator for performing the first Futamura projection, thereby obtaining a *compiler* for the interpreter under consideration. This process is called the second Futamura projection. The third Futamura projection now consists of specialising the partial evaluator to perform the second Futamura projection. By this process we obtain a *compiler generator* (*cogen* for short).

**History of self-application for logic programming** Not surprisingly, writing an effectively self-applicable specialiser is a non-trivial task — the more features one uses in writing the specialiser the more complex the specialisation process becomes, as the specialiser then has to handle these features as well. For a long time it was believed that in order to develop a self-applicable specialiser for logic programs one needed to write a clean, pure and simple specialiser. In practice, this meant using few (or even no) impure features in the implementation of the specialiser. For this the *ground representation* [14] was believed to be key, in which variables of the source program are represented by ground constants within the specialiser. Indeed, the ground representation allows one to freely manipulate the source program to be specialised in a declarative manner. The *non-ground representation*, where source-level variables are represented as variables in the program specialiser, can suffer from semantical problems [23] and requires some non-declarative features (such as `findall/3`) in order to perform the specialisation.

Some early attempts at self-application [6] used the non-ground representation, but the self-applying led to incorrect results as the specialiser did not properly handle the non-declarative constructs that were employed in its implementation[2]. Other specialisers like MIXTUS [27], PADDY [26] and ECCE [22] use the non-ground representation, but none of them are able to effectively specialise themselves.

The ground representation approach towards self-application was pursued in [3], [19], [24], and [4, 12, 13] leading to some self-applicable specialisers:

- SAGE [12], a self-applicable partial evaluator for Gödel. While the speedups obtained by self-application are respectable, the process takes a very long time (several hours) and the obtained specialised specialisers are still extremely slow. This is probably due to the explicit unification algorithm required by the ground representation. To effectively specialise this much more powerful specialisation techniques would be required to obtain reasonably efficient specialisers. Similar performance problems were encountered in the earlier work [3].
- LOGIMIX [16, 24], a self-applicable partial evaluator for a subset of Prolog, including *if-then-else*, side-effects and some built-in's. LOGIMIX uses a meta-interpreter (sometimes called *InstanceDemo*) for the ground representation in which the goals are "lifted" to the non-ground representation for resolution. This avoids the use of an explicit unification algorithm, at the expense

[2] A problem mentioned in [3], see also [19, 24].

of some power [3]. Unfortunately, LOGIMIX gives only modest speedups (when compared to results for functional programming languages, see [24]), but it was probably the first practical self-applicable specialiser for a logic programming language.

Given the problem in developing a truly practical self-applicable specialiser for logic programs, the attention shifted to the *cogen approach* [15]: instead of trying to write a partial evaluation system which is neither too inefficient nor too difficult to self-apply, one simply writes a compiler generator directly. Indeed, the actual creation of the cogen according to the third Futamura projection is in general not of much interest to users since the cogen can be generated once and for all when a specialiser is given. This approach was pursued in [17, 21] leading to the LOGEN system, which can produce specialised specialisers much more efficiently than any of the self-applicable systems mentioned above. The resulting specialisers themselves are also much more efficient.

**A new attempt at self-application** In a sense the cogen approach has closed the practical debate on self-application for logic programming languages: one can get most of the benefits of self-application without writing a self-applicable specialiser. Still, there is the question of academic curiosity: is it really impossible to derive the cogen written by hand in [17, 21] by self-application? Also, having a self-applicable specialiser is sometimes more flexible as we may generate different cogen's for different purposes (such as one with debugging enabled). One may produce more or less optimised cogen's by tweaking the specialisation process, and better control the tradeoff between specialisation time and quality of the optimised code. Maybe there are other situations where a self-applicable partial evaluation system is preferrable to a cogen: Glück's specialiser projections [10] and the semantic modifiers of Abramov and Glück [1] may be such a setting.

This paper aims to answer some of these questions. Indeed, after the development of LOGEN we realised that one could translate LOGEN into a classical partial evaluator without too much difficulty. Furthermore, using new annotation facilities developed for the second version of LOGEN [21], one can actually make this partial evaluator (henceforth called LIX) self-applicable. By self-applying LIX we obtain generating extensions via the second Futamura projection which are very similar to the ones produced by LOGEN and the cogen obtained via the third Futamura projection also has lot of similarities to the code of LOGEN. The performance of this self-applicable partial evaluator is (after self-application) on par with LOGEN, and is thus much faster than any of the previous self-applicable logic programming specialisers. In the paper we also show some potential practical applications of this self-applicable specialiser.

The code of the specialiser itself is also surprisingly simple, but uses a few non-declarative features and does not use the ground representation. So, contrary to earlier belief, declarativeness and the ground representation were not the best way to climb the mountain of self-application. Indeed, the use of the non-ground

[3] This idea was first used by Gallagher in [8, 9] and then later in [20] to write a declarative meta-interpreter for integrity checking in databases.

representation makes our partial evaluator much more efficient and avoids all the complications related to specialising an explicit unification algorithm. The only drawback is that to safely deal with the non-ground representation, our partial evaluator needs to use some non-declarative features such as `findall`, and hence also has to be able to specialise them. Fortunately, this turned out to be less of a problem than anticipated.

In summary, Futamura's insight was that a cogen could be derived by a self-applicable specialiser. The insight in [15] was that a cogen is just a simple extension of a binding-time analysis, while our insight is that an effective self-applicable specialiser can be derived by transforming a cogen.

## 2 The Partial Evaluator

LOGEN and LIX are both offline partial evaluators. An offline partial evaluator works on an annotated version of the source program, these annotations are used to guide the specialisation process. There are two kinds of annotations:

- **filter declarations**, indicating whether arguments to predicates are **static** or **dynamic**. This influences the global control.
- **clause annotations**, indicating how every call in the body should be treated during unfolding. These influence the local control.

### 2.1 The Basic Annotations

A common annotation format is used for both the LIX and LOGEN systems. Each call in the program is annotated using `logen/2` and arguments are annotated using filter declarations. The head of a clause is annotated with an identifier. The format of the annotations is demonstrated in the following append example:

```
:- filter append(static,dynamic,dynamic).
logen(app, append([],L,L)).
logen(app, append([H|T], L, [H|T1])) :- logen(unfold, append(T,L,T1).
```

The first argument to append has been marked as **static**, it will be known at specialisation time, and the other arguments have been marked **dynamic**. The recursive call to append is annotated for unfolding, the first argument is known thus guaranteeing termination at specialisation time. Some of the basic annotations are:

- **unfold** for reducible predicates, they will be unravelled during specialisation.
- **memo** for non-reducible predicates, they will be added to the memoisation table and replaced with a generalised residual predicate.
- **call** fully static call will be made during specialisation.
- **rescall** the call will be kept and will appear in the final specialised code.

### 2.2 The Source Code

We now present the main body of the LIX partial evaluator[4]. An atom $A$ is specialised by calling `lix(`$A$`,Res)`. The `memo/2` and `memo_table/2` predicates return in their second argument a call to a new specialised predicate where static arguments have been removed and dynamic ones generalised. Generalisation and filtering are performed by `generalise_and_filter/3`. It returns in its second argument the generalised call (to be unfolded) and in its third argument the call to the specialised predicate. It uses the annotations defined by `filter/2` to perform its task. The predicate `gensym/2` is used to create unique names for the specialised predicates. The predicate `unfold/2` computes the bodies of specialised predicates. A call annotated as `memo` is replaced by a call to the specialised version. If it does not already exist it is created by `memo/2`. A call annotated as `unfold` is further unfolded; a call annotated as **call** is completely evaluated; finally, a call annotated as `rescall` is added to the residual code without modification (for built-ins that cannot be evaluated or code that is defined elsewhere). All clauses defining the new predicate are collected using `findall/3` and pretty printed.

Note the use of the global side effect, `assert(memo_table(GCall, RCall))`, to maintain the list of previously specialised calls. The `univ` operator `=..` can be used either to decompose a term into a list containing its functor and arguments or else construct a term from such a list. For example the term `f(X,Y)` can be deconstructed into `[f,X,Y]`.

To save space the definition of `pretty_print_clauses/1` is not given.

```
:- dynamic memo_table/2,flag/2.
lix(CallToSpecialise, ResidualCall) :-
        print(':- dynamic flag/2, memo_table/2.\n'),
        print(':- use_module(library(lists)).\n'),
        memo(CallToSpecialise, ResidualCall).
memo(Call, Residual) :-
        (   memo_table(Call, Residual) ->  true
        ;   generalise_and_filter(Call, GenCall, ResidualPred),
            assert(memo_table(GenCall,ResidualPred)),
            findall((ResidualPred:-Body), unfold(GenCall,Body), Clauses),
            format('/*~k=~k*/~n', [ResidualPred,GenCall]),
            pretty_print_clauses(Clauses), memo_table(Call, Residual)
        ).
unfold(Head, Residual) :- ann_clause(_, Head, Body),pe(Body, Residual).
pe(true, true).
pe((A,B), (ResA,ResB)) :- pe(A, ResA), pe(B, ResB).
pe(logen(call,Call), true) :- call(Call).
pe(logen(rescall,Call), Call).
pe(logen(memo,Call), Residual) :- memo(Call, Residual).
pe(logen(unfold,Call), Residual) :- unfold(Call, Residual).
generalise_and_filter(Call, GenCall, ResidualPred) :-
```

[4] The LIX system can be downloaded from:
`http://www.ecs.soton.ac.uk/~sjc02r/lix/lix.html`.

```
        filter(Call, Filter), Call=..[Head|Args],
        gen_filter(Filter, Args, GenArgs, ResArgs), GenCall=..[Head|GenArgs],
        gensym(Head, ResHead), ResidualPred =..[ResHead|ResArgs].
gen_filter([], [], [], []).
gen_filter([static|A], [B|C], [B|D], E) :- gen_filter(A, C, D, E).
gen_filter([dynamic|A], [_|B], [C|D], [C|E]) :- gen_filter(A, B, D, E).
/* code for unique symbol generation, using dynamic flag/2 */
oldvalue(Sym, Value) :- flag(gensym(Sym), Value), !.
oldvalue(_, 0).
set_flag(Sym, Value) :-
   nonvar(Sym), retract(flag(Sym,_)), !, asserta(flag(Sym,Value)).
set_flag(Sym, Value) :- nonvar(Sym), asserta(flag(Sym,Value)).
gensym(Head, ResidualHead) :-
   var(ResidualHead), atom(Head), oldvalue(Head, OldVal),
   NewVal is OldVal+1, set_flag(gensym(Head), NewVal),
   name(A__, "__"), string_concat(Head, A__, Head__),
   string_concat(Head__, NewVal, ResidualHead).
append([], A, A).
append([A|B], C, [A|D]) :-  append(B, C, D).
string_concat(A, B, C) :- name(A, D), name(B, E),
                          append(D, E, F), name(C, F).
/* Clause Database: automatically created from annotated file */
ann_clause(1, app([],A,A), true).
ann_clause(2, app([A|B],C,[A|D]), logen(memo,app(B,C,D))).
filter(app(_,_,_), [dynamic,static,dynamic]).
```

### 2.3 Deriving LIX from LOGEN

The LIX partial evaluator was created by transforming the LOGEN compiler generator. The basic insight was that it is possible to create a classical partial evaluator that when specialised would produce similar generating extensions. Let us compare a small extract of code from both LOGEN and LIX, dealing with the **call** and **rescall** annotations:

| LOGEN | LIX |
|---|---|
| `body(logen(call,Call),Call,true).` | `pe(logen(call,Call), true) :- call(Call).` |
| `body(logen(rescall,Call),true,Call).` | `pe(logen(rescall,Call), Call) :- true.` |

The `body` predicate is explained in detail in [21]. Basically, the first argument is an annotated call, the second argument is the code that will appear in the generating extension and the third argument denotes the specialised code. We can see that the middle argument from `body/3` in LOGEN has been transformed into a call in the LIX version. This call is annotated as **residual** for self-application, and will hence appear in the generating extension produced by self-application. A more detailed comparison of the generating extensions and the produced cogen can be found in Section 5.

### 2.4 Specialised Code

To specialise code we use the `lix/2` entry point. Calling `lix(app(A,[b],C),Res)` specialises the append predicate to append `[b]` to the end of a list:

```
app__1([], [b]).
app__1([A|B], [A|C]) :- app__1(B, C).
```

The generation of the above code took 0.318 ms[5]. This is a very simple example to demonstrate the partial evaluator. The specialisation of a non-trivial Vanilla debugging interpreter and other examples can be found on the `lix` homepage[6].

## 3 Towards Self-Application

We have presented the main body of the code for the LIX system. For a partial evaluator to be self-applicable it must be able to effectively handle all of the features it uses. The system we have presented so far uses a few non-declarative features and does not use the ground representation. In this section will shall introduce the required extension to make LIX self-applicable.

### 3.1 The nonvar Binding-Type

We now present a new feature derived from LOGEN which is useful when specialising interpreters. This annotation will be the key for effective self-application.

In addition to marking arguments to predicates as **static** or **dynamic**, it is also possible to use the binding-type **nonvar**. This means that this argument is not a free variable and will have at least a top-level function symbol, but it is not necessarily ground. For example `f(X)`, `f(a)` and `f` are all nonvar but the variable `X` is not. During generalisation, the top level function symbol is kept but all its sub-arguments are replaced by fresh variables. For filtering, every sub-argument becomes a new argument of the residual predicate.

A small example will help to illustrate this annotation:

```
:- filter p(nonvar).
p(f(X)) :- p(g(a)).              p(g(X)) :- p(h(X)).
p(h(a)).                         p(h(X)) :- p(f(X)).
```

If we mark no calls as unfoldable, we get the following specialised program for the call `p(f(Z))`:

```
%%% entry point:  p(f(Z)) :- p__0(Z)
p__0(B) :-   p__1(a).            p__1(B) :-   p__2(B).
p__2(a).                         p__2(B) :-  p__0(B).
```

If we mark everything except the last call as unfoldable we obtain:

```
p__0(B).
p__0(B) :- p__0(a).
```

[5] Benchmarks performed using SICStus Prolog 3.10.1 for Linux on a Pentium 2.4GHz with 512MB RAM.

[6] `http://www.ecs.soton.ac.uk/~sjc02r/lix/lix.html`

The `gen_filter/2` predicate in the LIX source code is extended to handle the **nonvar** annotation:

```
gen_filter([nonvar|A], [B|C], [D|E], F) :-
        B=..[G|H], length(H, I), length(J, I),
        D=..[G|J], gen_filter(A, C, E, K), append(J, K, F).
```

### 3.2 Treatment of findall

In LIX `findall` is used to collect the clauses when unfolding a call; hence we have to be able to treat this feature during specialisation.

Handling `findall` is actually not much different from handling negation in [21]. There is a static version (`findall`), in which the call is executed at specialisation time, and a dynamic version (`resfindall`), where it is executed at runtime. In both cases, the second argument must be annotated. For `resfindall`, much like `resnot` in [21], the annotated argument should be deterministic and should not fail (which can be ensured by wrapping the argument into a `hide_nf` annotation, see [21]). Also, if a `findall` is marked as static then the call should be sufficiently instantiated to fully determine the list of solutions. The following code is used in the subsequent examples:

```
:- filter all_p(static,dynamic).
all_p(X,Y) :- findall(X,p(X),Y).
:- filter p(static).
p(a). p(b).
```

If the `findall` is marked as residual and we `memo p(X)` inside it then the specialised program for `all_p(a,Y)` is:

```
all_p__0(A) :- findall(a,p__1,A).
p__1.
```

If we mark `p(X)` as `unfold` we get:

```
all_p__0(A) :- findall(a,true,A).
```

For self-application, only `resfindall` is actually required. The `pe/2` predicate is extended as follows:

```
pe(resfindall(Vars,G2,Sols), findall(Vars,VS2,Sols)) :-
     pe(G2,VS2).
```

### 3.3 Treatment of if

In the LIX code an `if-then-else` is used in `memo/2`. In this case the `if` is dynamic, the body of the conditional will be computed, along with those of the branches and an `if` statement will be constructed in the residual code. LIX is also extended to handle a static `if` which is performed at specialisation time.

```
pe(resif(A,B,C), (D->E;F)) :- pe(A, D), pe(B, E), pe(C, F).
pe(if(A,B,C), D) :- (pe(A, _) -> pe(B, D) ; pe(C, D)).
```

### 3.4 Handling the cut

This is actually very easy to do, as with careful annotation the cut can be treated as a normal built-in call. The cut must be annotated using **call**, where it is performed at specialisation time, or **rescall**, where it is included in the residual code. It is up to the annotator to ensure that this is sound, i.e. LIX assumes that:

- if a cut marked **call** is reached during specialisation then the calls to the left of the cut will never fail at runtime.
- if a cut is marked as **rescall** within a predicate $p$, then no calls to $p$ are unfolded.

These conditions are sufficient to handle the cut in a sound, but still useful manner.

## 4 Self-Application

Using the features introduced in Section 3 and the basic annotations from Section 2.1, LIX can be successfully annotated for self-application. Self-application allows us to achieve the Futamura projections mentioned in the introduction.

### 4.1 Generating Extensions

In Section 2.4 we specialised `app/3` for the call `app(A,[b],C)`. If a partial evaluator is fully self-applicable then it can specialise itself for performing a particular specialisation, producing a *generating extension*. This process is the second Futamura projection. When specialising an interpreter the generating extension is a compiler.

A generating extension for the append predicate can be created by calling `lix(lix(app(A,B,C),R),R1)`, creating a specialised specialiser for append.

```
/*Generated by Lix*/
:- dynamic flag/2, memo_table/2.
/* oldvalue__1(_5557,_5586) = oldvalue(_5557,_5586) */
oldvalue__1(A, B) :- flag(gensym(A), B), !.
oldvalue__1(_, 0).

/* set_flag__1(_7128,_7153) = set_flag(gensym(_7128),_7153) */
set_flag__1(A, B) :- retract(flag(gensym(A),_)), !,
                     asserta(flag(gensym(A),B)).
set_flag__1(A, B) :- asserta(flag(gensym(A),B)).

/* gensym__1(_4392) = gensym(app,_4392) */
gensym__1(A) :- var(A), oldvalue__1(app, B),
                C is B+1,set_flag__1(app, C),
                name(C, D), name(A, [97,112,112,95,95|D]).
/* Printing and Flatten Clauses removed to save space */
```

```
/* unfold__1(_6925,_6927,_6929,_6956) = unfold(app(_6925,_6927,_6929),_6956) */
unfold__1([], A, A, true).
unfold__1([A|B], C, [A|D], E) :- memo__1(B, C, D, E).

/* memo__1(_2453,_2455,_2457,_2484) = memo(app(_2453,_2455,_2457),_2484) */
memo__1(A, B, C, D) :-
        (   memo_table(app(A,B,C), D) -> true
        ;   gensym__1(E), F=..[E,G,H],
            assert(memo_table(app(G,B,H),F)),
            findall((F:-I), unfold__1(G,B,H,I), J),
            format('/*~k=~k*/~n', [F,app(G,B,H)]),
            pretty_print_clauses__1(J),
            memo_table(app(A,B,C), D)
        ).
/* lix__1(_1288,_1290,_1292,_1319) = lix(app(_1288,_1290,_1292),_1319) */
lix__1(A, B, C, D) :- memo__1(A, B, C, D).
```

This is almost entirely equivalent to the proposed specialised unfolders in [17, 21]. It is actually slightly better as it will do flow analysis and only generate unfolders for those predicates that are reachable from the query to be specialised. Note the `gensym/2` predicate is specialised to produce only symbols of the form `app__N`. Generation of the above took 3.3 ms.

The generating extension for append can be used to specialise the append predicate for different sets of static data. Calling the generating extension with `lix__1(A,[b],C,R)` creates the same specialised version of the append predicate as in seciton 2.4:

```
app__1([], [b]).
app__1([A|B], [A|C]) :- app__1(B, C).
```

However using the generating extension is faster, for this small example 0.212 ms instead of 0.318 ms. Using a larger benchmark, unfolding (as opposed to memoising) the append predicate for a 10,000 item list produces more dramatic results. To generate the same code the generating extension takes 40 ms compared to 990 ms for LIX. The overhead of creating the generating extension for the larger benchmark is only 10 ms. Generating extensions can be very efficient when a program is to be specialised multiple times with different static data.

### 4.2 Lix Compiler Generator

The third Futamura projection is realised by specialising the partial evaluator to perform the second Futamura projection. By this process we obtain a *compiler generator* (*cogen* for short), a program that transforms interpreters into compilers. By specialising LIX to create generating extensions we create LIX-COGEN, a self-applied compiler generator. This can be achieved with the query `lix(lix(lix(Call,R),R1),R2)`. An extract from the produced code is now given:

```
/*unfold__13(Annotation, Generated Code, Specialisation Time) */
unfold__13(true, true, true).
unfold__13((A,B), (C,D), (E,F)) :-
        unfold__13(A, C, E),
        unfold__13(B, D, F).
unfold__13(logen(call,A), true, call(A)).
unfold__13(logen(rescall,A), A, true).
...
```

This has basically re-generated the 3-level cogen described in [17, 21]. In the **rescall** annotation for example, the call ($A$) will become part of the residual program, and nothing ($true$) is performed at specialisation time.

This code extract demonstrates the importance of the **nonvar** annotation. The annoated version of the original `unfold/2` is now shown.

```
:- filter unfold(nonvar,dynamic):unfold.
logen(unfold, unfold(X,Code)) :-
        logen(unfold, ann_clause(_,X,B)),
        logen(unfold, pe(B,Code)).
```

Without the **nonvar** annotation the first argument would be annotated **dynamic** as the arguments to the call being unfold may not be known at specialisation time. This would produce a single generic unfolder predicate much like the original `lix`. The **nonvar** annotation is needed to generate the specialised unfolders.

The generated LIX-COGEN will transform an annotated program directly into a generating extension, like the one found in section 4.1. However LIX-COGEN is faster: to create the same generating extension from an input program of $1,000$ predicates LIX-COGEN takes only 3.9 s compared to 100.9 s for LIX.

## 5 Comparison

**Logen** The LOGEN system is an offline partial evaluation system using the cogen approach. Instead of using self-application to achieve the third Futamura projection, the LOGEN compiler generator is hand written. LIX was derived from LOGEN by rewriting it into a classical partial evaluation system. Using the second Futamura projection and self-applying LIX produces almost identical generating extensions to those produced by LOGEN (and both systems can in principle treat full Prolog). Apart from the predicate names the specialised unfolders generated by the two systems are the same:

| LOGEN Generating Extension | LIX-COGEN Generating Extension |
|---|---|
| `app__u([],A,A,true).` | `unfold__1([], A, A, true).` |
| `app__u([A\|B],C,[A\|D],E) :-` | `unfold__1([A\|B], C, [A\|D], E) :-` |
| `  app__m(B,C,D,E).` | `  memo__1(B, C, D, E).` |
| `...` | `...` |

While LOGEN is a hand written compiler generator, LIX must be self-applied to produce the same result as in Section 4.2. If we compare the LOGEN source code to the output in Section 4.2 we find very similar clauses in the form of `body/3` (note however, that the order of the last two arguments is reversed).

```
body(true,true,true).
body((G,GS),(G1,GS1),(V,VS)) :-
  body(G,G1,V),
  body(GS,GS1,VS).
body(logen(call,Call),Call,true).
body(logen(rescall,Call),true,Call).
```

LOGEN

```
unfold__13(true, true, true).
unfold__13((A,B), (C,D), (E,F)) :-
      unfold__13(A, C, E),
      unfold__13(B, D, F).
unfold__13(logen(call,A), true, call(A)).
unfold__13(logen(rescall,A), A, true).
```

LIX-COGEN

Unlike LIX, LOGEN does not perform flow analysis. It produces unfolders for all predicates in the program, regardless of whether or not they are reachable.

**Logimix and Sage** Comparisons of the initial *cogen* with other systems such as LOGIMIX, PADDY, and SP can be found in [17]. In essence, LOGEN was was 50 times faster than LOGMIX at producing the generating extensions (0.02 s instead of 1.10 s or 0.02 s instead of 0.98 s) and the specialisation times were about 2 times faster. It is likely that a similar relationship holds between LIX and LOGIMIX given that LIX and LOGEN have similar performance. Unfortunately LOGIMIX no longer runs on current versions of SICStus Prolog and we were thus unable to compare LIX and LOGIMIX directly. Similarly, Gödel no longer runs on current versions of SICStus Prolog, and hence we could not produce any timings for SAGE. However, timings from [12] indicate that the use of the ground representation means that SAGE is far too slow to be practical. Indeed, generating the compiler generator took about 100 hours and creating a generating extension for the examples in [12] took at least 7.9 hours. The speedups from using the generating extension instead of the partial evaluator range from 2.7 to 3.6 but the execution times for the generating extensions still ranged from 113 s to 447 s.

**Multi-level Languages** Our annotation scheme (for both LIX and LOGEN) can be viewed as a two-level language. Contrary to MetaML [28] our annotations are not part of the programming language itself (as we treat classical Prolog). It would be interesting to investigate to what extent one could extend our scheme for multiple levels of specialisation [11].

## 6 New Applications

Apart from the academic satisfaction of building a self-applicable specialiser, we think that there will be practical applications as well. We elaborate on a few in this section.

**Several Versions of the Cogen** In the development of new annotation and specialisation techniques it is often useful to have a debugging specialisation environment without incurring any additonal overhead when it is not required. Using LIX we can produce a debugging or non-debugging specialiser from the same base code, the overhead of debugging being specialised away when it is not required. By augmenting LIX with extra options we can produce several versions of the cogen depending on the requirements:

– a debugging cogen, useful if the specialisation does not work as expected
– a profiling cogen
– a simple cogen, whose generating extensions produce no code but which can be fed into termination analysers or abstract interpreters to obtain information to check the annotations.

We could also play with the annotations of LIX to produce more or less aggressive specialisers, depending on the desired tradeoff between specialisation time, size of the specialised code and the generating extensions, and quality of the specialised code. This would be more flexible and maintainable than re-writing LOGEN to accomodate various tradeoffs.

**Extensions for Deforestation/Tupling** LIX is more flexible than LOGEN: we do not have to know beforehand which predicates are susceptible to being unfolded or memoised. Hence, LIX can handle a potentially unbounded number of predicates. Using this allows LIX to perform a simple form of conjunctive partial deduction [5].

For example, the following is the well known double append example where conjunctive partial deduction can remove the unnecessary intermediate datastructure `XY` (this is *deforestation*):

```
doubleapp(X,Y,Z,XYZ) :- append(X,Y,XY), append(XY,Z,XYZ).
append([],L,L).
append([H|X],Y,[H|Z]) :- append(X,Y,Z).
```

When annotating this example for LIX we can now simply annotate a conjunction as `memo` (which is not allowed in LOGEN):

```
ann_clause(1,doubleapp(A,B,C,D), (memo((append(A,B,E),append(E,C,D))))).
```

Running LIX on this will produce a result where the intermediate datastructure has been removed (after post-processing, as in [5]):

```
doubleapp(A,B,C,D) :- doubleapp__0(A,B,C,D).
append__2([],B,B).
append__2([C|D],E,[C|F]) :- append__2(D,E,F).
conj__1([],[],B,B).
conj__1([],[C|D],E,[C|F]) :- append__2(D,E,F).
conj__1([G|H],I,J,[G|K]) :- conj__1(H,I,J,K).
doubleapp__0(B,C,D,E) :- conj__1(B,C,D,E).
```

For this example to work in LOGEN we would need to declare every possible conjunction skeleton beforehand, as a specialised unfolder predicate has to be generated for every such conjunction. LIX is more flexible in that respect, as it can unfold a conjunction even if it has not been declared before.

We have also managed to deal with the rotate-prune example from [5], but more research will be needed into the extent that the extra flexibility of LIX can be used to do deforestation or tupling in practice. It should be possible, for example, to find out whether there is a bounded number of conjunction skeletons simply by self-application.

## 7 Conclusions and Future Work

We have presented an implemented, effective and surprisingly simple, self-applicable partial evaluation system for Prolog and have demonstrated that the ground representation is not required for a partial evaluation system to be self-applicable. The LIX system can be used for the specialisation of non-trivial interpreters, and we hope to extend the system to use more sophisticated binding types developed for LOGEN.

While LIX and LOGEN essentially perform the same task, there are some situations where a self-applicable partial evaluation system is preferrable. LIX can potentially produce better generating extensions, using specialised versions of `gensym` and performing some of the generalisation and filtering beforehand. We have shown the potential for the use of LIX in deforestation, and in producing multiple cogens from the same code. Tweaking the annotation of LIX allows the cogen generation to be controlled. The overhead of a debugging cogen can be removed or a more aggressive specialiser can be generated.

At present the annotations for LIX and LOGEN are placed by hand. We are still working on a fully automatic binding time analysis (bta). The automatic bta will be used with a graphical interface allowing the user to tweak the annotations.

## References


1. S. M. Abramov and R. Glück. Semantics modifiers: an approach to non-standard semantics of programming languages. In M. Sato and Y. Toyama, editors, *Third Fuji International Symposium on Functional and Logic Programming*, page to appear. World Scientific, 1998.
2. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).
3. A. Bondorf, F. Frauendorf, and M. Richter. An experiment in automatic self-applicable partial evaluation of Prolog. Technical Report 335, Lehrstuhl Informatik V, University of Dortmund, 1990.
4. A. F. Bowers and C. A. Gurr. Towards fast and declarative meta-programming. In K. R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 137–166. MIT Press, 1995.
5. D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms and experiments. *The Journal of Logic Programming*, 41(2 & 3):231–277, 1999.
6. H. Fujita and K. Furukawa. A self-applicable partial evaluator and its use in incremental compilation. *New Generation Computing*, 6(2 & 3):91–118, 1988.
7. Y. Futamura. Partial evaluation of a computation process — an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
8. J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.
9. J. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93*, pages 88–98. ACM Press, 1993.
10. R. Glück. On the generation of specialisers. *Journal of Functional Programming*, 4(4):499–514, 1994.

11. R. Glück and J. Jørgensen. Efficient multi-level generating extensions for program specialization. In S. Swierstra and M. Hermenegildo, editors, *Proceedings PLILP'95*, LNCS 982, pages 259–278, Utrecht, The Netherlands, September 1995. Springer-Verlag.
12. C. A. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel*. PhD thesis, Department of Computer Science, University of Bristol, January 1994.
13. C. A. Gurr. Specialising the ground representation in the logic programming language Gödel. In Y. Deville, editor, *Proceedings LOPSTR'93*, Workshops in Computing, pages 124–140, Louvain-La-Neuve, Belgium, 1994. Springer-Verlag.
14. 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 Science Publications, Oxford University Press, 1998.
15. C. K. Holst. Syntactic currying: yet another approach to partial evaluation. Technical report, DIKU, Department of Computer Science, University of Copenhagen, 1989.
16. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
17. J. Jørgensen and M. Leuschel. Efficiently generating efficient generating extensions in Prolog. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar*, LNCS 1110, pages 238–262, Schloß Dagstuhl, 1996. Springer-Verlag.
18. J. Komorowski. An introduction to partial deduction. In A. Pettorossi, editor, *Proceedings Meta'92*, LNCS 649, pages 49–69. Springer-Verlag, 1992.
19. M. Leuschel. Partial evaluation of the "real thing". In L. Fribourg and F. Turini, editors, *Proceedings of LOPSTR'94 and META'94*, LNCS 883, pages 122–137, Pisa, Italy, June 1994. Springer-Verlag.
20. M. Leuschel and D. De Schreye. Towards creating specialised integrity checks through partial evaluation of meta-interpreters. In *Proceedings PEPM'95*, pages 253–263, La Jolla, California, June 1995. ACM Press.
21. 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.
22. M. Leuschel, B. Martens, and D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, January 1998.
23. B. Martens and D. De Schreye. Why untyped non-ground meta-programming is not (much of) a problem. *The Journal of Logic Programming*, 22(1):47–99, 1995.
24. T. Mogensen and A. Bondorf. Logimix: A self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement, editors, *Proceedings of LOPSTR'92*, pages 214–227. Springer-Verlag, 1992.
25. A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming*, 19& 20:261–320, May 1994.
26. S. Prestwich. The PADDY partial deduction system. Technical Report ECRC-92-6, ECRC, Munich, Germany, 1992.
27. D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.
28. W. Taha and T. Sheard. Metaml and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248:211–242, 2000.