The Ecce and Logen Partial Evaluators and their Web Interfaces

Michael Leuschel *
University of Düsseldorf, Germany
leuschel@cs.uni-duesseldorf.de

Dan Elphick
University of Southampton, U.K.
dre@ecs.soton.ac.uk

Mauricio Varea
University of Southampton, U.K.
m.varea@ecs.soton.ac.uk

Stephen-John Craig
University of Düsseldorf, Germany
steve.craig@gmail.com

Marc Fontaine
University of Düsseldorf, Germany
fontaine@cs.uni-duesseldorf.de

Abstract

We present ECCE and LOGEN, two partial evaluators for Prolog using the *online* and *offline* approach respectively. We briefly present the foundations of these tools and discuss various applications. We also present new implementations of these tools, carried out in Ciao Prolog. In addition to a command-line interface new user-friendly web interfaces were developed. These enable non-expert users to specialise logic programs using a web browser, without the need for a local installation.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; D.1.6 [*Programming Techniques*]: Logic Programming; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic — *logic programming*; I.2.2 [*Artificial Intelligence*]: Automatic Programming; I.2.3 [*Artificial Intelligence*]: Deduction and Theorem Proving — *logic programming*

Keywords Partial evaluation, partial deduction, Prolog, program transformation, logic programming, web programming

1. Introduction

Partial evaluation [18] is an automatic technique for program optimisation which works by specialising a given source program for a particular application. The main idea is to pre-evaluate the given source program as much as possible, given part of the input values for the program. Partial evaluation thus generalises traditional compiler optimisation techniques such as inlining and constant folding. It can obtain bigger improvements than these techniques, but it is also more difficult to control.

Partial evaluators can be broadly classified into *online* and *offline* systems, depending on how the specialisation process is controlled. An offline system works in two phases: the first phase produces an annotated version of the source program. In the second phase the annotated source program, together with the partial input, can be used to efficiently generate the specialised program. An online system, on the other hand, requires no preliminary annotation phase and takes all its decisions automatically during specialisation.

Both the online and offline approach have advantages and disadvantages when compared against each other. Online systems are easy to use and since all processing takes place in one run, they can use highly flexible strategies for *local* and *global control* that make full use of all available information. Thus online systems can, in principle, achieve better specialisations than (pure) offline systems, where the binding time analysis has to be independent of the actual values of the available input. The drawback of this is the higher complexity of the specialisation. The advantage of offline specialisers is that, once a suitable annotation of the source program is found, programs can be specialised much faster than with online specialisers. On reason is that it is possible to generate a specialised specialiser for a given source program via self-application or by using the cogen approach. The presence of the annotation makes it also more easy to predict the actual outcome of the specialisation (i.e., it can be determined whether a certain computation will be specialised away in all cases).

In this paper we present two specialisers: ECCE, an online specialiser and LOGEN, a specialiser based on offline BTA but extended with some features of online specialisers.

2. The Ecce System

ECCE is an online specialiser for logic programs. Its input are pure Prolog programs (with built-ins, as long as they are used in a declarative manner). The control of ECCE is separated into two components [38]:

- The *global control* (also called control of polyvariance) decides *which* specialised predicates are present in the residual program and ensures that all calls that occur at runtime are an instance of some specialised atom.
- The *local control* (also called unfolding) is responsible for the construction of the residual clauses for each specialised atom.

Some of the particularities of the ECCE system are:

– For the local control ECCE uses determinacy (with lookahead) to decide which atoms should be unfolded and homeomorphic embedding (see, e.g., [23]) to ensure termination (i.e., decide which ones should not be unfolded).
– For the global control ECCE uses both characteristic trees and homeomorphic embedding [32, 33]. The characteristic trees

* All the authors have been partially supported by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-38059 ASAP project.

PEPM '06 January 9–10, Charleston, South Carolina, USA.

[10] are used to capture the specialisation performed for an atom, and are used to control the polyvariance: if two atoms have a different characteristic tree then they are specialised in a different way and combining them would result in a loss of specialisation. The homeomorphic embedding relation is used to ensure that only finitely many atoms are specialised. For generalisation, the most specific generalisation (*msg*) [21] is used.
- ECCE caters for *conjunctive partial deduction* [26, 8]. Essentially, this makes it possible to specialise a conjunction of atoms together rather than in isolation, opening up the door for optimisations such as deforestation (getting rid of intermediate data structures) and tupling (merging multiple visits of the same data structure).
- Redundant argument filtering [35] is another ingredient of ECCE which filters out various useless arguments in the specialised programs.

Various abstract interpretation techniques have been integrated into ECCE (more specific version calculation [37] in [25], and regular types in [27]). It is also possible to develop custom local and global control predicates, as well as override various other components of ECCE. Note that ECCE does not support non-declarative Prolog programs (i.e., Prolog programs which rely on the left-to-right selection rule): while this restricts the applicability of ECCE,[1] it also enables ECCE to perform more powerful optimisations (e.g., replacing infinite by finite failure). However, ECCE is still guaranteed (in default settings) to preserve the order of solutions.

2.1 Applications of ECCE

Evaluations of ECCE's performance on a variety of benchmarks can be found in [33] and in [20, 8] for the conjunctive partial deduction component. Results are compared, amongst others, against MIXTUS [42], SP [10] and PADDY [40].

There have been many applications of ECCE outside of the classical objective of speeding up logic programs. For example, ECCE has been applied to program inversion [14], inductive theorem proving [31] and to solve planning problems of the fluent calculus [22].

Furthermore, ECCE has been used for infinite state model checking in general [34] and for the coverability problem of Petri nets [30, 29] in particular. It has been shown that a special case of the control algorithm of ECCE corresponds to the Karp-Miller procedure for Petri nets. Recently, ECCE has also been applied for verification of Object Petri nets in [9].

In [36] it was shown that by changing ECCE's post-processor it was possible to obtain a precise slicing algorithm for logic programs. Finally, ECCE has been used as a pre-processor for termination analysers [43], with the aim of proving more programs terminating.

2.2 A modular implementation in Ciao Prolog

ECCE was originally implemented in BIM Prolog and later ported to SICStus. The majority of ECCE's evolution has taken place while being written in SICStus Prolog. On the advent of the ASAP EU project we decided twofold: (a) to port the tool another time, to Ciao Prolog, while maintaining compatibility with SICStus Prolog, and (b) to use modularity, in order to obtain a cleaner implementation of ECCE. As a further advantage, Ciao's compiler, `ciaoc`, allows to easily create stand-alone executables and command-line tools. The ECCE executable operates in three different modes:

- in interactive mode,
- in scripted mode for benchmarking, and
- via a command-line interface (CLI).

In the *interactive* mode, users interact with the tool via a text-mode menu. The mode *for benchmarking* was created to automate the interaction with ECCE for specialising a larger number of example files for specific specialisation queries and automatically evaluate the performance improvements. Finally, an extensive CLI was added to ECCE in order to efficiently operate the tool from different front-ends. Figure 1 shows the different switches that can be used in this mode.

2.3 ECCE's Web Interface

Based on the new command-line interface we have developed a web interface for ECCE. The new GUI makes it possible to employ and control ECCE via the Internet using any ordinary web-browser without the need for a local installation. This section gives a brief outline of the web interface and describes the steps that take place during a typical web-ECCE session.

First the user has to provide ECCE with a Prolog source-program. This can be done by uploading a program, by choosing one of the ready-made examples or by typing the code in a small text-area. Limited editing of the current program is also possible in this text-area.

After that, the user can enter the goal-predicate that is required for some of the transformations and toggle several options that control internal settings of ECCE. For almost all applications the default-settings are already the best choice.

When all preparations are done, the user can start one of several source-to-source transformations and analyses by simply pressing the corresponding submit button.

At the moment, there are five different transformations and analyses that can be carried out. These are:

Specialisation Tells ECCE to specialise the source program P for a given goal G (and all its instances).

Slicing Strips out useless code, by analysing what clauses of P are needed to compute the answers of G. It runs the same algorithm as for *specialise*, but uses a different code generator [36].

Most Specific Version Computes the most specific version [37] of the program, by performing a bottom-up abstract interpretation, keeping a single success pattern per predicate and using the *msg* to combine multiple success patterns. This can also be run automatically as a post-processing, as described in [25].

Redundant Argument Filtering This allows to run the first algorithm presented in [35], without having to specialise the program (by default the algorithm is run automatically after specialisation, together with *inverse redundant argument filtering*).

Inverse Redundant Argument Filtering This runs the second algorithm from [35], weeding out further redundant arguments.

The GUI displays the result of the transformation in a small text-area underneath the source-program. For slicing operations, the output is mapped back to the original prolog-program and unused code is greyed out. All output prolog-programs can be downloaded from the GUI for further processing by the user.

An interesting feature of the GUI is the possibility to produce a visualisation of the specialisation tree (see Fig. 3). This tree is a graphical representation of the derivation of the specialised program. Green edges labelled with unf(L,N) show the unfolding that has been done; where L is the literal number selected and N is the number of the clause resolved with. Blue edges denote the descendency relationship at the global control level. Solid rectangles containing a definition with a double equality are global tree nodes for which code has been generate. Gray nodes are global tree nodes

[1] For non-declarative programs MIXTUS [42] or LOGEN are better alternatives.

```
% ./ecce --help
ECCE: The online partial evaluator for pure Prolog
       (c) Michael Leuschel 1995-2005
USAGE:
ecce [OPTIONS] FILE
 OPTIONS:
   -pe "GOAL"    partially evaluate FILE for GOAL
   -slice "GOAL" slice FILE for GOAL
   -msv          run MSV Analysis on FILE
   -raf GOAL     run RAF Argument Filtering on FILE for GOAL
   -far          run FAR Argument Filtering on FILE
   -o FILE       write specialised program to FILE
   -dot FILE     write specialization graph to dot FILE (before post-processing)
   -v            verbose mode
   -t            perform self-test
   -d            print debugging information
   -i            stay in interactive mode after performing OPTIONS
   -config OPT   change default control settings, OPT=classic,fast,mixtus,minimal,classic-fast,deforest
   -pp OPT       change default postprocessor settings, OPT=off,max
```

Figure 1. ECCE's Command-line Interface

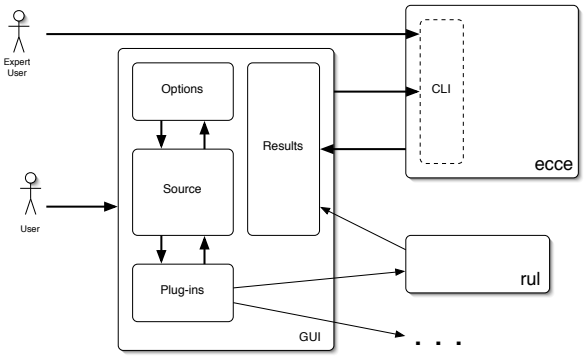

Figure 2. Overall Architecture

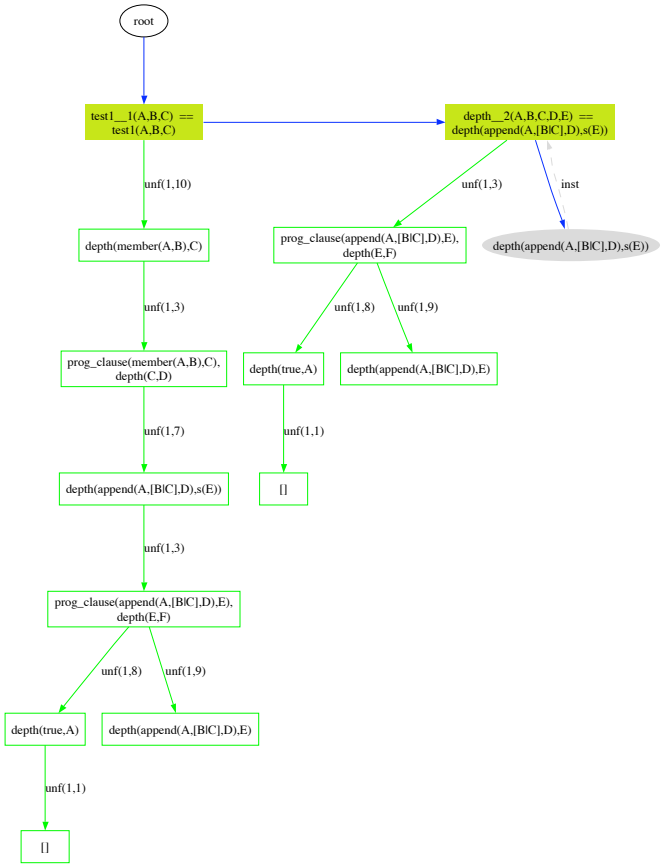

Figure 3. Specialisation Tree

for which no code was generated, either because the node was abstracted or because it is an instance of another node. After a specialisation has been computed, clicking on a button opens a window with the specialisation tree.

A major concern of any web interface is security. During specialisation ECCE evaluates predicates of the Prolog code submitted by the user. To ensure that malicious Prolog code cannot corrupt the web server, ECCE contains a whitelist of built-in predicates and the conditions, under which it is safe to evaluate them. In addition the computation time and the memory usage of individual specialisation processes is limited to avoid overloading the web-server. (for complex specialisation tasks users should download and install a local copy of ECCE.)

The implementation of the web interface is based on the scripting-language PHP. It doesn't rely on any particular features of the web browser and should be compatible with any reasonable browser. To view specialisation trees a svg-plugin is needed.

The web interface is modular and can easily be ported, e.g. for a local installation. In addition, the interface has scope for extending ECCE's functionality by appending extra modules that also perform source-to-source transformations. We call these modules *plug-ins*, and the system already has incorporated several, e.g. the Regular Unary Logic (RUL) analysis (with and without magic-set transformation) of [11], Non-Determinisitc Finite Tree Automaton (NFTA) of [12] (again with and without magic-set transformation), and CiaoPP with default Analysis and Specialisation options [41].

3. The Logen System

Let us now turn our attention to the second tool: LOGEN, a partial evaluator developed using the cogen approach.

With the help of a self-applicable partial evaluator, one can construct a *compiler generator* (*cogen*) using the third Futamura projection (see e.g. [18]). A *cogen* is a program that given an annotated source program produces a specialised specialiser for that program. If the source program is an interpreter, this specialiser can be viewed as a compiler, hence the name "compiler generator."

However, obtaining an efficient cogen by self-application is a difficult task, This has led several researchers to write the *cogen* by hand [16, 17, 3, 1, 13, 44], rather than trying to obtain it by self-application. In the setting of logic programming this idea has led to the LOGEN system [19, 28].

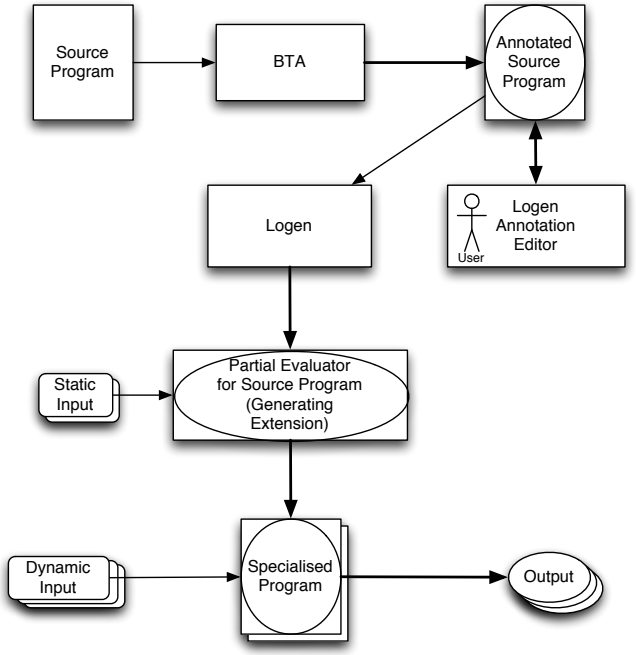

Figure 4. Illustrating the LOGEN system and the *cogen* approach

Figure 4 (from [24]) highlights the way the LOGEN system works. Typically, a user would proceed as follows (more details can be found in [24] from which part of the material is taken):

- First the source program is annotated using a binding-time analysis (BTA), which produces an annotated source program. The annotation essentially marks program points and predicate arguments as static or dynamic. This annotated source program can be further inspected, which allows the user to edit and manually refine the annotations to get better specialisation.
- Second, the LOGEN compiler generator is run on the annotated source program and produces a specialised specialiser for the source program, also called a *generating extension*.
- This generating extension can now be used to specialise the source program for some static input. Note that the same generating extension can be used for different static inputs (i.e., there is no need to regenerate the generating extension unless the annotated source program changes).
- As soon as the remainder of the input is known, the specialised program can be run and will produce the same output as the original source program. Note again, that the same specialised program can be run for different dynamic inputs; the specialised program only has to be regenerated if the static input changes (or the original program itself changes).

The particularities of the LOGEN system are:

- Efficient specialisation through the use of specialised specialisers.
- Support for partially static data, with user-definable binding-types. For example, the following defines a list of variable bindings, with statically known variable names but unknown (at specialisation time) variable values:

```
:- type env ---> [] ;
        [ bind(static,dynamic) | type(env)].
```

- Support for almost full Prolog, e.g., the if-then-else, negation, disjunction, findall, the cut, built-ins, modules, higher-order calls,...
- An automatic BTA [6] which ensures termination of the unfolding process. The BTA uses the binary clause semantics to identify potential infinite loops in the specialised specialiser. In addition, LOGEN contains two simple BTAs which annotate everything as static and unfold/call or dynamic and memo/rescall respectively. These can be used in circumstances where the automatic BTA cannot be used, e.g., in cases where the source program is large and the automatic BTA is too expensive. The automatic BTA can also be used just to propagate the static/dynamic information without trying to infer terminating clause annotations.
- A watchdog mode which detects non-terminating annotations as well as support for online unfold annotations and online binding-types using online control techniques such as the homeomorphic embedding relation.
- Extensions for Constraint Logic Programming [5], as well as support for co-routining (`when` declarations) [4] are available.
- A self-tuning resource aware specialisation method that adapts the annotations for a particular architecture and set of representative sample queries [7].

3.1 Applications of LOGEN

Various experimental evaluations of the LOGEN system have been reported; see for example [28]. In essence, specialisation without the binding-time analysis turns out to be very fast (several orders of magnitude faster than online specialisers). The quality of the specialisation of course depends highly on the annotation, but the experiments show that with the proper annotations specialisation can be on par with online specialisers.

LOGEN has been applied to numerous interpreters. [24] contains a systematic study on how to apply LOGEN to interpreters in general, and how to achieve "Jones optimality" in particular.

In [2] LOGEN has been succesfully applied to a flexible meta-interpreter for access control checks on deductive databases. By specialisation of this interpreter one obtains flexible access control with virtually zero overhead.

In [45] LOGEN has been used towards the goal of provably correct compilation by specialising denotational semantics language specifications expressed in Horn logic and definite clause grammars. In particular, the semantics for the SCR specification language was expressed, and LOGEN generated target code in a provably correct manner.

LOGEN has also been used as a preprocessor to achieve efficient model checking for domain specific specification languages. For example, in [34] a CTL model checker was specialised for particular systems and specific temporal logic formulas. In [30, 29] LOGEN was used to precompile a Petri net interpreter for later infinite model checking. In [9] an interpreter for Object Petri nets was specialised together with the CTL model checker to achieve efficient verification. LOGEN is also currently being applied to improve the efficiency of the expressive pointcut language for aspect-orientation from [39]. Recently, in the ASAP project (EU FET IST-2001-38059) LOGEN was used to specialise an emulator (written by Kim Henriksen and John Gallagher) for the machine language of PIC processors, for analysis purposes [15].

3.2 Re-implementation in Ciao

The very first version of LOGEN [19] was implemented in BIM Prolog. LOGEN was then ported to SICStus Prolog. Within the ASAP EU project LOGEN was also re-implemented in Ciao Prolog,

with similar motivations as for ECCE. Of crucial importance was the fact that with Ciao Prolog one can include the compiler in runtime systems[2] enables a user to produce generating extensions which are fully stand-alone, without any further need for the LOGEN system. Indeed, this is one of the often cited advantages of the COGEN approach, which has thus come true in this new version.

```
% logen --help
Usage: logen [Options] File.pl ["Atom."]
  Possible Options are:
      --help: Prints this message
      -w: watch mode (supervise specialization)
      -W: watch non interactive; halt at first alarm
      -c: run cogen only (not the .gx file)
      -s: run silently
      --safe: run gx in safe sandbox mode
      -v: print debugging messages
      -vv: print more debugging messages
      -vvv: print even more debugging messages
      --compile_gx: compile the gx file using ciaoc
      -m: display memo table
      -g: display gx file
      --logen_dir ARG1: path to logen files
      -o ARG1: GX filename
      --spec_file ARG1: Spec filename
      --ciao_path ARG1: Ciao binary directory
      --simple_bta ARG1 ARG2 ARG3: Run simple bta
      -d: debug mode for GX file
      -d2: even more debugging messages in GX file
      --single_process: run logen in single process
```

Figure 5. LOGEN's Command-line Interface

3.3 LOGEN's Web Interface

Previous versions of LOGEN had various graphical front ends, implemented in Tcl/Tk, XEmacs, and Python/Tk respectively. Distribution and installation of these front ends was not always unproblematic, and we have now moved to a more elegant web interface, providing a user-friendly interface for running LOGEN and editing the annotations. A main advantage of the web interface over the earlier interfaces is its portability and the ability to run LOGEN remotely without local installation (even though we also cater for local execution).

The web interface provides a way to annotate programs and then specialise them. Colour coding is used to provide visual feedback about the annotations (e.g., static parts are marked green, dynamic parts red). The web interface also ensures that the source file and the annotation file are kept in sync: if the user changes the source file the changes will be detected and only the changed parts have to be re-annotated.

Security: Security was an important aspect in the design process of the web interface, as allowing untrusted users access to a LOGEN system installed on a web server could open up security holes: in particular we need to prevent the execution of unsafe Prolog code submitted by users (e.g., `system('rm *.*')`). To do so, we have extended LOGEN to produce *safe* generating extensions. In such a *safe* generating extension, every *call* to a Prolog predicate, is first checked against a whitelist of safe predicates. Trying to call an unsafe predicate will cause the process to be aborted and trigger a warning message.

Our whitelist also allows using the `call` predicate and solution aggregation predicates such as `findall` and `bagof`, but only if the predicate argument is itself in the whitelist.

[2] Academic licenses of SICStus Prolog do not allow the distribution of the Prolog compiler in packaged runtime systems. Unfortunately, the automatic BTA still requires SICStus Prolog since we do not have a robust convex hull solver in Ciao Prolog yet. This is packaged up as a separate binary, which does not require access to the Prolog compiler.

Besides arbitrary code execution, the interface also has to handle accidental or deliberate denial of service (DOS) attacks due to infinite unfolding or memoization. As a counter measure, spawned processes are automatically killed when a time limit is exceeded, but currently there is a risk that too many simultaneous connections could overload the server and cause even legitimate processes to time out without completing.

The interface can also be used offline on a local machine as long as it has a web server. For this usage safe mode can be disabled. (But it is advised to carefully restrict access to trusted users only through server mechanisms in this case.) The specialisation timeout can be extended in the case of more CPU-intensive code.

Displaying and Editing Annotations: As already shown in Figure 4, the annotation system maintains two files: a source file (Figure 7 shows append) and an annotation file (Figure 8). Our interface preserves the formatting of the source code and overlays the annotations using colours and underlining to distinguish the different types. Converting the source and annotation file to HTML is a two-stage process shown in Figure 6. First an XML representation (Figure 9) is created using a Prolog program that matches the annotations to the source code. This process keeps the structure and comments from the original source code. The XML is then processed by an XSL transform which creates HTML code containing style sheet information and Javascript hooks as shown in Figure 10. Having two stages simplifies each stage and allows a separation of skills, where a Prolog programmer can create simple structured XML in the first stage, while web programmers can control the second without intimate knowledge of Prolog.

Figure 6. Converting source and annotation files to HTML

```
append([],Y,Y).
append([X|Xs],Y,[X|Zs]) :-
    append(Xs,Y,Zs).
```

Figure 7. Prolog source for `append.pl`

```
logen(append/3,append([],A,A)).
logen(append/3,append([B|C],A,[B|D])) :-
    logen(memo,append(C,A,D)).
:- filter
    append(dynamic,dynamic,dynamic).
```

Figure 8. Annotation file for `append.pl`

Javascript is used extensively so that programs can be annotated without repeated loading of new pages from the server. Instead clicking on an annotated predicate brings up a menu of possible annotations and selecting one changes the HTML code locally on the server; the changes are only sent to the server when the user has finished. The annotations are sent to the server as a flat list of the form:

```
[unfold, unfold, memo, call]
```

These annotations are then mapped back onto the source using a Prolog executable, `annotate`, resulting in an annotated program. Since the filters are more free-form, they are edited and sent back as text from the client (even though a limited form of editing, such as changing static arguments to dynamic or vice-versa can be performed directly using the mouse). The server passes this text to

```
<?xml version="1.0" encoding="UTF-8"?>
<article>
<source><head>append</head>([],Y,Y).
<head>append</head>([X|Xs],Y,[X|Zs]) :-
    <memo>append</memo>(Xs,Y,Zs).
</source>
<filters>
:- filter
    <filter>append</filter>(dynamic,dynamic,dynamic).
</filters>
</article>
```

Figure 9. XML produced by the annotation reader. (Some tags used only for syntax highlighting have been removed to aid clarity)

```
<pre class="source">
<span class="head">append</span>([],Y,Y).
<span class="head">append</span>([X|Xs],Y,[X|Zs]) :-
    <span class="memo" id="ann16"
       onclick="return dropdownmenu(this, event)"
       onmouseover="mouseoverAnn('ann16')"
       onmouseout="mouseoutAnn('ann16')"
    >append</span>(Xs,Y,Zs).
</pre>
...
<pre>
:- filter
    <span class="filter">append</span>(dynamic,
            dynamic, dynamic).
</pre>
```

Figure 10. HTML produced from `append.pl`

`annotate`, which parses and appends the result to the annotation file.

4. Conclusion

In conclusion, the ECCE system provides advanced specialisation for pure Prolog programs. Being an online specialiser, its usage is simple and accessible to non-expert users. The development of the web interface provides an even simpler way of using the system, requiring no local installation and providing easy access to various options of the system. The ECCE system has been applied to numerous case studies, and has found many applications outside of its original goal area of specialising programs, in particular for the analysis and verification of systems expressed as Prolog rules.

Since the appearance of the first version of LOGEN in 1996 in [19] the system has now achieved a certain maturity and has been applied to numerous case studies. It is now in a state where it can be used by programmers who are not researchers in partial evaluation. Indeed, the first version of LOGEN was efficient, but only supported a small subset of Prolog and all annotations had to be provided by hand. LOGEN now supports almost full Prolog, provides a command-line interface, has considerable support for annotating source code and especially the web interface has lowered the entry threshold for using the system. We hope that the LOGEN system will thus prove to be valuable to researchers and programmers alike.

More information about the tools can be found at the main web sites for our tools:

- `http://stups.cs.uni-duesseldorf.de/~pe/ecce`
- `http://stups.cs.uni-duesseldorf.de/~pe/weblogen`

The choice on which tool to use depends on the particular application. ECCE is a fully automatic online specialiser. It is hence easier to use by inexperienced users, but more difficult to tweak in case specialisation does not proceed as desired. ECCE uses more refined control techniques and can perform conjunctive partial deduction, which LOGEN cannot. Because of the additional annotation phase, LOGEN is more difficult to master by inexperienced users, but the outcome of the specialisation is easier to tweak and predict. The specialisation phase of LOGEN is very efficient, with very little overhead compared to ordinary evaluation. LOGEN can deal with almost full Prolog, whereas ECCE only deals with declarative Prolog programs. The latter restricts the range of programs to which ECCE can be applied, but it also allows ECCE to perform more powerful optimisations.

Acknowledgements

We wish to thank all the people who have been involved in the development of the ECCE and LOGEN tools and their mathematical foundations. In particular, we wish to thank Danny De Schreye, Jesper Jørgensen and Bern Martens for their initial work on ECCE and LOGEN. Furthermore, we are grateful for the contributions of Morten-Heine Sørensen as well as Laksono Adhianto, André de Waal, John Gallagher, Robert Glück, Stefan Gruner, Helko Lehmann, Germán Puebla, and Germán Vidal. We also would like to thank the partners of the ASAP project, in particular the people from UPM for their support and improvements to Ciao Prolog, and Kim Henriksen and Andrew Moss for their feedback on LOGEN.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[2] S. Barker, M. Leuschel, and M. Varea. Efficient and flexible access control via logic program specialisation. In *Proceedings PEPM '04*, pages 190–199, New York, NY, USA, 2004. ACM Press.

[3] L. Birkedal and M. Welinder. Hand-writing program generator generators. In M. Hermenegildo and J. Penjam, editors, *Proceedings PLILP'91*, LNCS 844, pages 198–214, Madrid, Spain, 1994. Springer-Verlag.

[4] S. Craig. *Practicable Prolog Specialisation*. PhD thesis, University of Southampton, U.K., June 2005.

[5] S. Craig and M. Leuschel. A compiler generator for constraint logic programs. In M. Broy and A. V. Zamulin, editors, *Proceedings of PSI'03*, LNCS 2890, pages 148–161. Springer-Verlag, 2003.

[6] S.-J. Craig, J. Gallagher, M. Leuschel, and K. S. Henriksen. Fully automatic binding-time analysis for Prolog. In S. Etalle, editor, *Proceedings LOPSTR 2004*, LNCS 3573, pages 53–68. Springer-Verlag, August 2004.

[7] S.-J. Craig and M. Leuschel. Self-tuning resource aware specialisation for Prolog. In *Proceedings PPDP '05*, pages 23–34, New York, NY, USA, 2005. ACM Press.

[8] 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, November 1999.

[9] B. Farwer and M. Leuschel. Model checking object Petri nets in Prolog. In *Proceedings PPDP '04*, pages 20–31, New York, NY, USA, 2004. ACM Press.

[10] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3 & 4):305–333, 1991.

[11] J. Gallagher and D. A. de Waal. Fast and precise regular approximations of logic programs. In P. Van Hentenryck, editor, *Proceedings ICLP'94*, pages 599–613. The MIT Press, 1994.

[12] J. P. Gallagher and K. S. Henriksen. Abstract domains based on regular types. In B. Demoen and V. Lifschitz, editors, *Proceedings ICLP 2004*, LNCS 3132, pages 27 – 42, January 2004.

[13] 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.

[14] R. Glück and M. Leuschel. Abstraction-based partial deduction for solving inverse problems – a transformational approach to software verification. In *Proceedings of the Third International Ershov Conference on Perspectives of System Informatics*, LNCS 1755, pages 93–100, Novosibirsk, Russia, 1999. Springer-Verlag.

[15] K. S. Henriksen and J. P. Gallagher. Analysis and specialisation of a PIC processor. In *Proceedings of the IEEE International Conference on Systems, Man & Cybernetics (2)*, pages 1131–1135, The Hague, The Netherlands, 2004.

[16] C. K. Holst. Syntactic currying: yet another approach to partial evaluation. Technical report, DIKU, Department of Computer Science, University of Copenhagen, 1989.

[17] C. K. Holst and J. Launchbury. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming, Skye, Scotland*, pages 210–218. Glasgow University, 1991.

[18] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[19] 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.

[20] J. Jørgensen, M. Leuschel, and B. Martens. Conjunctive partial deduction in practice. In J. Gallagher, editor, *Proceedings LOPSTR'96*, LNCS 1207, pages 59–82, Stockholm, Sweden, August 1996. Springer-Verlag.

[21] J.-L. Lassez, M. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan-Kaufmann, 1988.

[22] H. Lehmann and M. Leuschel. Solving planning problems by partial deduction. In M. Parigot and A. Voronkov, editors, *Proceedings LPAR'2000*, LNAI 1955, pages 451–468, Reunion Island, France, 2000. Springer-Verlag.

[23] 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 - Essays dedicated to Neil Jones*, LNCS 2566, pages 379–403. Springer-Verlag, 2002.

[24] 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.

[25] M. Leuschel and D. De Schreye. Logic program specialisation: How to be more specific. In H. Kuchen and S. Swierstra, editors, *Proceedings PLILP'96*, LNCS 1140, pages 137–151, Aachen, Germany, September 1996. Springer-Verlag.

[26] M. Leuschel, D. De Schreye, and A. de Waal. A conceptual embedding of folding into partial deduction: Towards a maximal integration. In M. Maher, editor, *Proceedings JICSLP'96*, pages 319–332, Bonn, Germany, September 1996. MIT Press.

[27] M. Leuschel and S. Gruner. Abstract partial deduction using regular types and its application to model checking. In A. Pettorossi, editor, *Proceedings LOPSTR'2001*, LNCS 2372, pages 91–110, Paphos, Cyprus, 2001. Springer-Verlag.

[28] 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.

[29] M. Leuschel and H. Lehmann. Coverability of reset Petri nets and other well-structured transition systems by partial deduction. In J. Lloyd, editor, *Proceedings of the International Conference on Computational Logic (CL'2000)*, LNAI 1861, pages 101–115, London, UK, 2000. Springer-Verlag.

[30] M. Leuschel and H. Lehmann. Solving coverability problems of Petri nets by partial deduction. In M. Gabbrielli and F. Pfenning, editors, *Proceedings PPDP'2000*, pages 268–279, Montreal, Canada, 2000. ACM Press.

[31] M. Leuschel and H. Lehmann. Inductive theorem proving by program specialisation: Generating proofs for Isabelle using Ecce (invited talk). In M. Bruynooghe, editor, *Proceedings LOPSTR'03*, LNCS 3018. Springer-Verlag, 2004.

[32] M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar*, LNCS 1110, pages 263–283, Schloß Dagstuhl, 1996. Springer-Verlag.

[33] 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.

[34] M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi, editor, *Proceedings LOPSTR'99*, LNCS 1817, pages 63–82, Venice, Italy, 2000.

[35] M. Leuschel and M. H. Sørensen. Redundant argument filtering of logic programs. In J. Gallagher, editor, *Proceedings LOPSTR'96*, LNCS 1207, pages 83–103, Stockholm, Sweden, August 1996. Springer-Verlag.

[36] M. Leuschel and G. Vidal. Forward slicing by conjunctive partial deduction and argument filtering. In M. Sagiv, editor, *Proceedings ESOP 2005*, LNCS 3444, pages 61–76. Springer-Verlag, April 2005.

[37] K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. *Annals of Mathematics and Artificial Intelligence*, 1:303–338, 1990.

[38] B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling, editor, *Proceedings ICLP'95*, pages 597–613, Kanagawa, Japan, June 1995. MIT Press.

[39] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In A. P. Black, editor, *European Conference on Object-Oriented Programming (ECOOP)*, LNCS 3586, pages 214–240. Springer-Verlag, 2005.

[40] S. Prestwich. Online partial deduction of large programs. In *Proceedings PEPM'93*, pages 111–118. ACM Press, 1993.

[41] G. Puebla, E. Albert, and M. Hermenegildo. A generic framework for the analysis and specialization of logic programs. In A. Serebrenik and S. Munoz-Hernandez, editors, *Proceedings of the 15th Workshop on Logic-based methods in Programming Environments*, Sitges, Spain, October 2005.

[42] D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.

[43] L. Tamary and M. Codish. Abstract partial evaluation for termination analysis. In *7th International Workshop on Termination (WST 2004)*, June 2004.

[44] P. Thiemann. Cogen in six lines. In *International Conference on Functional Programming*, pages 180–189. ACM Press, 1996.

[45] 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.