Efficient Specialisation in Prolog Using the Hand-Written Compiler Generator LOGEN

Michael Leuschel

Declarative Systems and Software Engineering, Dept. of Electronics and Computer Science, University of Southampton, Southampton SO17 1BJ, UK
E-MAIL: mal@ecs.soton.ac.uk
WWW: http://www.ecs.soton.ac.uk/~mal

Jesper Jørgensen

Dept. of Mathematics and Physics, Royal Veterinary and Agricultural University, Thorvaldsensvej 40, DK-1871 Frederiksberg C, Denmark
E-MAIL:jesper@dina.kvl.dk

Abstract

The so called "*cogen* approach" to program specialisation, writing a compiler generator instead of a specialiser, has been used with considerable success in partial evaluation of both functional and imperative languages. In earlier work we have shown that this approach is also applicable to partial evaluation of logic programming languages, also called partial deduction.

In this paper we extend upon this by allowing partially instantiated datastructures (via binding types), which are especially important in the context of logic programming. We also extend *cogen* to directly support a large part of Prolog's declarative and non-declarative features and how semi-online specialisation can be efficiently integrated. Benchmarks show that the resulting *cogen* is very efficient, generates very efficient generating extensions (executing up to several orders of magnitude faster than current online systems) which in turn perform very good and non-trivial specialisation, even rivalling existing online systems.

1 Introduction and Overview

Partial evaluation has over the past decade received considerable attention both in functional, imperative and logic programming. In the context of pure logic programs, partial evaluation is sometimes referred to as *partial deduction*, the term partial evaluation being reserved for the treatment of impure logic programs.

Guided by the *Futamura projections* a lot of effort, specially 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. In that case one may, according to the second Futamura projection, obtain *compilers* from interpreters and, according to the third Futamura projection, a *compiler generator* (*cogen* for short).

However 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, because the specialiser then has to handle these features as well. However, the actual creation of the *cogen* according to the third Futamura projection is not of much interest to users since *cogen* can be generated once and for all when a specialiser is given. Therefore, from a user's point of view, whether a *cogen* is produced by self-application or not is of little importance; what is important is that it exists and that it is efficient and produces efficient, non-trivial compilers. This is the background behind the approach to program specialisation called the *cogen approach*: 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. This is not as difficult as one might imagine at first sight: basically the *cogen* turns out to be just a simple extension of a "binding-time analysis" for logic programs (something first discovered for functional languages in [4]).

The most noticeable advantages of the *cogen* approach is that the *cogen* and the compilers it generates can use all features of the implementation language. Therefore, no restrictions due to self-application have to be imposed (the compiler and the compiler generator do not have to be self-applied)! As we will see, this leads to extremely efficient compilers and compiler generators.

Although the Futamura projections focus on how to generate a compiler from an interpreter, the projections of course also apply when we replace the interpreter by some other program. In this case the program produced by the second Futamura projection is not called a compiler, but a *generating extension*. The program produced by the third Futamura projection could rightly be called a *generating extension generator* or gengen, but we will stick to the more conventional *cogen*.

The first *cogen* for logic programming languages was developed in [5]. In this paper we present a much improved and more practical *cogen*. Due to space restrictions we can only give an overview; full details can be found in the technical report [7]. Basically, our main contributions are:

1. a formal specification of the concept of a *binding-type analysis*, allowing the treatment of *partially static* structures, in a (pure) logic programming setting and a description of how to obtain a generic algorithm for *offline*

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

partial deduction from such an analysis.

Basically, binding-types are Hilog types [2] with three pre-defined type constructors: `static`, `dynamic`, and `nonvar`. This is much more refined than the initial approach in [5] which classified arguments as either static or dynamic and which was often too weak for logic programs, where partially instantiated datastructures appear naturally even at runtime.

2. based upon point 1, the description of an efficient, handwritten compiler generator (*cogen*) which generates efficient generating extensions. The crucial idea for simplicity and efficiency of the generating extensions is to incorporate a specific "unfolding" predicate p_u for each predicate p.
3. a way to handle both *extra-logical* features (such as `var/1` or the if-then-else) and *side-effects* (such as `print/1`) within the *cogen*. A refined treatment of the `call/1` predicate has also been developed, allowing improved specialisation of higher-order programs.
4. how to handle negation, disjunction and the if-then-else conditional in the *cogen*.
5. extensive benchmark results showing the efficiency of the *cogen*, the generating extensions but also of the specialised programs.

Compared with [5], the points 3, 4, 5 as well as the partially static structures of point 1 are new, leading to a much more powerful, practical and viable *cogen*.

2 Summary of Benchmark Results

Due to space limiations we cannot delve into the formal and technical details of our new *cogen* system. We therefore just present a summary of experiments we carried out using the system.

A first study of the speed of the *cogen* approach was performed in [5]. However, due to the limitations of the initial *cogen* only very few realistic benchmarks could be run. In particular, most of the benchmarks of the DPPD suite [6] could not be used because they require the treatment of partially instantiated data. The improved *cogen* of this paper can now deal with all the benchmarks in [6]. We thus ran our system on a selection of benchmarks from [6]. To test the ability to specialise non-declarative built-in's we also devised one new non-declarative benchmark: specialising the non-ground unification algorithm with occurs-check from [11].

The implementation of the new *cogen* is actually called LOGEN, runs under Sicstus Prolog and is publicly available. We compare the results of LOGEN with the latest versions of MIXTUS [10] (version 0.3.6) and ECCE [8,3]. (Comparisons of the initial *cogen* with other systems such as LOGIMIX, PADDY, and SP can be found in [5]). All the benchmarks were run under `SICStus Prolog 3.7.1` on a Sun Ultra E450 server with 256Mb RAM operating under `SunOS 5.6`.

Program	MIXTUS	ECCE		LOGEN	
	with	with	w/o	cogen	genex
ex_depth	200 ms	230 ms	190 ms	1.5 ms	7.2 ms
grammar	220 ms	200 ms	140 ms	6.5 ms	1.1 ms
map.rev	70 ms	60 ms	30 ms	2.7 ms	1.0 ms
map.reduce	30 ms	60 ms	30 ms	"	1.3 ms
match.kmp	50 ms	90 ms	40 ms	1 ms	2.5 ms
model_elim	460 ms	240 ms	170 ms	3 ms	3.1 ms
regexp.r1	60 ms	110 ms	80 ms	1.3 ms	1.4 ms
regexp.r2	240 ms	120 ms	80 ms	"	2.5 ms
regexp.r3	370 ms	160 ms	120 ms	"	10.2 ms
transpose	290 ms	190 ms	150 ms	1.2 ms	1.9 ms
ng_unify	2510 ms	na	na	5.3 ms	3.5 ms

Table 1
Specialisation Times

A summary of all the transformation times can be found in Table 1. The times for MIXTUS contains the time to write the specialised program to file (as we are not the implementors of MIXTUS we were unable to factor this part out), as does the column marked "with" for ECCE. The column marked "w/o" is the pure transformation time of ECCE without measuring the time needed for writing to file. The times for LOGEN exclude writing to file. For LOGEN, the column marked by *cogen* contains the runtimes of the *cogen* to produce the generating extension, whereas the column marked by *genex* contains the times needed by the generating extensions to produce the specialised programs. To be fair, it has to be emphasised that the binding-type analysis was carried out by hand. In a fully automatic system thus, the column with the *cogen* runtimes will have to be increased by the time needed for the binding-type analysis. However, the binding-type analysis and the *cogen* have to be run only *once* for every program and division. Thus, the generating extension produced for $regexp.r1$ was re-used without modification for $regexp.r2$ and $regexp.r2$ while the one produced for *map.rev* was re-used for *map.reduce*. Note that ECCE can only handle declarative programs, and could therefore not be applied on the ng_unify benchmark.

As can be seen in Table 1, LOGEN is by far the fastest specialisation system overall, running up to almost 3 orders of magnitude faster than the existing online systems. And, as can be seen in Table 2, the specialisation performed by the LOGEN system is not very far off the one obtained by MIXTUS and ECCE; some-

Program	Original	mixtus	ecce	logen
ex_depth	1470 ms	680 ms	540 ms	530 ms
	1	2.16	2.72	2.77
grammar	2880 ms	200 ms	300 ms	190 ms
	1	14.40	9.60	15.16
map.rev	230 ms	100 ms	150 ms	120 ms
	1	2.30	1.53	1.92
map.reduce	540 ms	180 ms	150	170 ms
	1	3.00	3.60	3.18
match.kmp	3740 ms	2570 ms	1940 ms	3260 ms
	1	1.46	1.93	1.15
model_elim	1210 ms	340 ms	320 ms	450 ms
	1	3.56	3.78	2.69
regexp.r1	3240 ms	520 ms	760 ms	510 ms
	1	6.23	4.26	6.35
regexp.r2	900 ms	360 ms	350 ms	300 ms
	1	2.50	2.57	3.00
regexp.r3	1850 ms	550 ms	590 ms	1610 ms
	1	3.36	3.14	1.15
transpose	1590 ms	70 ms	70 ms	70 ms
	1	22.71	22.71	22.71
ng_unify	1600 ms	360 ms	na	430 ms
	1	4.44	-	3.72

Table 2
Runtimes and speedups of the specialised programs

times logen even surpasses both of them (for *ex_depth*, *grammar*, *regexp.r*1 and *regexp.r*2)! Being a pure offline system, logen cannot pass the KMP-test, which can be seen in the timings for *match.kmp* in Table 2. (To be able to pass the KMP-test, more sophisticated local control would be required, see [9].) To be fair, both ecce and mixtus are fully automatic systems guaranteeing termination, while for logen further work in the line of [1] will be needed so that the binding-type classifications used in the above benchmarks can be derived automatically (while still ensuring termination). Nonetheless, the

LOGEN system is surprisingly fast and produces surprisingly good specialised programs.

References

[1] M. Bruynooghe, M. Leuschel, and K. Sagonas. A polyvariant binding-time analysis for off-line partial deduction. In C. Hankin, editor, *Proceedings of the European Symposium on Programming (ESOP'98)*, LNCS 1381, pages 27–41. Springer-Verlag, April 1998.

[2] W. Chen, M. Kifer, and D. S. Warren. A first-order semantics of higher-order logic programming constructs. In E. L. Lusk and R. A. Overbeek, editors, *Logic Programming: Proceedings of the North American Conference*, pages 1090–1114. MIT Press, 1989.

[3] 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. To appear.

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

[5] J. Jørgensen and M. Leuschel. Efficiently generating efficient generating extensions in Prolog. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 238–262, Schloß Dagstuhl, 1996. Springer-Verlag.

[6] M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.ecs.soton.ac.uk/~mal`, 1996.

[7] M. Leuschel and J. Jørgensen. Efficient specialisation in Prolog using a hand-written compiler generator. Technical Report DSSE-TR-99-6, Department of Electronics and Computer Science, University of Southampton, September 1999.

[8] 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.

[9] J. Martin and M. Leuschel. Sonic partial deduction. In *Proceedings of the Third International Ershov Conference on Perspectives of System Informatics*, LNCS 1755, Novosibirsk, Russia, 1999. Springer-Verlag. To appear.

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

[11] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.