ASAP

IST-2001-38059

Advanced Analysis and Specialization for Pervasive Systems

Specialization for Size Reduction

Deliverable number:	D9
Workpackage:	Resource-Oriented Specialization (WP4)
Preparation date:	1 May 2004
Due date:	1 May 2004
Classification:	Public
Lead participant:	Univ. of Southampton
Partners contributed:	Tech. Univ. of Madrid (UPM), Univ. of Bristol, Univ. of Southampton, Roskilde Univ

Project funded by the European Community under the "Information Society Technologies" (IST) Programme (1998–2002).

Short description:

Very often, programs are built from general purpose components. This results in very large programs: much of the code is not actually required for a particular application and can be eliminated. This deliverable studies and develops several ways to reduce the size of programs so as to enable their application in pervasive systems with limited memory resources. This reduction can be performed both at the source and at the compiled code level, in which case it is also important to take into account the size of the required libraries and of the run-time system.

The deliverable pursues both of these avenues and contains three main parts:

- A new, improved approach to program slicing, leveraging the tools developed in previous work of the ASAP project to reduce the size of the source code of programs.
- An optimising compilation of Prolog to C, with a view to reducing the footprint for running Prolog programs and enabling pervasive applications.
- An innovative application of program analysis and specialization to runtime system libraries which allows an important reduction of code size.

1 Program Slicing

Program slicing is a fundamental operation that has been successfully applied to solve many software engineering tasks, like, e.g., program understanding, maintenance, specialization, debugging, reuse, etc. Slicing was originally introduced by Weiser [Wei84]—in the context of imperative programs—as a debugging technique. Despite its potential applications, we found very few approaches to slicing in logic programming (some notable exceptions are, e.g., [GP95, SD96, SGM02, Vas99, ZCU01]).

Informally, a *program slice* consists of those program statements which are (potentially) related with the values computed at some program point and/or variable, referred to as a *slicing criterion*. Program slices are usually computed from a *program dependence graph* [FOW87] that makes explicit both the data and control dependences for each operation in a program. Program dependences can be traversed backwards and forwards—from the slicing criterion—giving rise to so-called *backward* and *forward* slicing. Additionally, slices can be *static* or *dynamic*, depending on whether a concrete program's input is provided or not. More detailed information on program slicing can be found in [HH01, Tip95].

Recently, Vidal [Vid03] introduced a novel approach to forward slicing of lazy functional logic programs. This work exploits the similarities between slicing and partial evaluation—already noticed in [RT96]—to compute forward slices by a slight modification of an existing partial evaluation scheme [AV02]. The main requirement of [Vid03] is that the underlying partial evaluation algorithm should be—in the terminology of [GS96]—both *monovariant* and *monogenetic* in order to preserve the structure of the original program. Unfortunately, this requirement also restricts the precision of the computed slices.

In this work, we extend the approach of [Vid03] in several ways. First, we adapt it to the logic programming setting. Second, we consider a polyvariant and polygenetic partial evaluation scheme: the *conjunctive* partial deduction algorithm of [DSGJ$^+$99] with control based on characteristic trees [GB91, LMDS98, LdS98]. Therefore, the computed slices are significantly more precise than those of the previous approach. Furthermore, since the basic partial deduction algorithm is kept unmodified, it can easily be implemented on top of an existing partial deduction system (in our case, ECCE [LMDS98]). Finally, we use the redundant argument filtering transformation of [LS96] to slice out unnecessary arguments of predicates (in addition to slicing out entire clauses).

The combination of these two approaches, [Vid03] and [LS96], together with a special-purpose slicing code generator, gives rise to a simple but powerful forward slicing technique. We also pay special attention to using slicing for code size reduction. Indeed, within the present ASAP project we are looking at resource-aware specialization techniques, with the aim of adapting software for pervasive devices with limited resources. We hence also analyze to what extent our approach can be used as an effective code size reduction technique, to reduce the memory footprint of a program.

Our main contributions are the following. We introduce the first, semantics-preserving, forward slicing technique for logic programs that produces executable slices. While traditional approaches in the literature demand different techniques to deal with static and dynamic slicing, our scheme is general enough to produce both static and dynamic slices. In contrast to [Vid03], the restriction to adopt a monovariant/monogenetic partial evaluation algorithm is not needed. Dropping this restriction is important as it allows us to use more powerful specialization schemes and, moreover, we do not need to modify the basic algorithm, thus easing the implementation of a slicing tool (i.e., only the code generation phase should be changed). We illustrate the usefulness of our approach on a series of benchmarks, and analyze its potential as a code-size reduction technique.

This first part of the deliverable attached is an improved and extended version of a paper accepted at a major conference (ESOP'05, part of ETAPS'05).

2 Improved Compilation

Several techniques for implementing Prolog have been devised since the original interpreter developed by Colmerauer and Roussel [Col93], many of them aimed at achieving more speed. An excellent survey of a significant part of this work can be found in [Van94]. The following is a rough classification of implementation techniques for Prolog (which is, in fact, extensible to many other languages):

- Interpreters (such as C-Prolog [Per87] and others), where a slight preprocessing or translation might be done before program execution, but the bulk of the work is done at runtime by the interpreter.

- Compilers to *bytecode* and their interpreters (often called emulators), where the compiler produces relatively low level code in a special-purpose language. Most current emulators for Prolog are based on the Warren Abstract Machine (WAM) [War83, AK91], but other proposals exist [Tay91, KB95].

- Compilers to a lower-level language, often ("native") machine code, which require little or no additional support to be executed. One solution is for the compiler to generate machine code directly. Examples of this are Aquarius [VD92], versions of SICStus Prolog [Swe99] for some architectures, BIM-Prolog [Mar93], and Gnu Prolog [DC01]. Another alternative is to generate code in a (lower-level) language, such as, e.g., C-- [JRR99] or C, for which compilers are readily available; the latter is the approach taken by `wamcc` [CD95].

Each solution has its advantages and disadvantages.

We describe the current status of and provide performance results for a prototype compiler of Prolog to C, termed `ciaocc`. `Ciaocc` is novel in that it is designed to accept different kinds of high-level information, typically obtained via an automatic analysis of the initial Prolog program and expressed in a standardized language of assertions. This information is used to optimize the resulting C code, which is then processed by an off-the-shelf C compiler. The basic translation process essentially mimics the unfolding of a bytecode emulator with respect to the particular bytecode corresponding to the Prolog program. This is facilitated by a flexible design of the instructions and their lower-level components. This approach allows reusing a sizable amount of the machinery of the bytecode emulator: predicates already written in C, data definitions, memory management routines and areas, etc., as well as mixing emulated bytecode with native code in a relatively straightforward way.

We report on the execution improvement of programs compiled by the current version of the system, both with and without analysis information, which can be divided into two different classes:

- Improvements in execution time, of general importance (and in special for systems with reduced resources), and
- Improvements in executable size, which are of concern in compilers for symbolic languages which generate lower-level, general-purpose code. In out case we report how information about type and determinism in the compiled programs affects the size of the executables.

This second part of the deliverable attached is an improvement over the previously submitted one. This version has been recently presented in a major conference on the field (PADL'05), while an earlier version was presented at the CICLOPS workshop.

3 Generation of stripped-down runtime systems

Finally, some effort has been devoted to the generation of runtime systems for which useless code has been removed, using abstract interpretation and program specialization techniques. Program specialization has been successfully applied to user programs, but it has not been directly used for system libraries yet. This approach is specially relevant for pervasive systems, as it allows the use of generic libraries for developing applications on virtual machine runtime-based systems. It presents important differences with respect to current runtime systems for pervasive devices (like Java Micro Edition series of runtime environments), as they have a prefixed set of runtime libraries specific for such systems.

Some results have been obtained, showing that an important reduction in runtime library size has been achieved using a simple abstract domain. It is expected that the application of more complex abstract domains will provide even better results. This work is described in the second part of this deliverable.

References

[AK91] Hassan Ait-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction*. MIT Press, 1991.

[AV02] E. Albert and G. Vidal. The Narrowing-Driven Approach to Functional Logic Program Specialization. *New Generation Computing*, 20(1):3–26, 2002.

[CD95] Philippe Codognet and Daniel Diaz. WAMCC: Compiling Prolog to C. In Leon Sterling, editor, *International Conference on Logic Programming*, pages 317–331. MIT PRess, June 1995.

[Col93] A. Colmerauer. The Birth of Prolog. In *Second History of Programming Languages Conference*, ACM SIGPLAN Notices, pages 37–52, March 1993.

[DC01] D. Diaz and P. Codognet. Design and Implementation of the GNU Prolog System. *Journal of Functional and Logic Programming*, 2001(6), October 2001.

[DSGJ+99] Danny De Schreye, Robert Glück, Jesper Jørgensen, Michael Leuschel, Bern Martens, and Morten Heine Sørensen. Conjunctive partial deduction: Foundations, control, algorithms and experiments. *The Journal of Logic Programming*, 41(2 & 3):231–277, November 1999.

[FOW87] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[GB91] J. Gallagher and M. Bruynooghe. The Derivation of an Algorithm for Program Specialisation. *New Generation Computing*, 9(3-4):305–333, 1991.

[GP95] T. Gyimóthy and J. Paakki. Static Slicing of Logic Programs. In *Proc. of the 2nd Int'l Workshop on Automated and Algorithmic Debugging (AADEBUG'95)*, pages 87–103. IRISA-CNRS, 1995.

[GS96] R. Glück and M.H. Sørensen. A Roadmap to Metacomputation by Supercompilation. pages 137–160. Springer LNCS 1110, February 1996.

[HH01] M. Harman and R. Hierons. An Overview of Program Slicing. *Software Focus*, 2(3):85–92, 2001.

[JRR99] Simon L. Peyton Jones, Norman Ramsey, and Fermin Reig. C--: A Portable Assembly Language that Supports Garbage Collection. In Gopalan Nadathur, editor, *International Conference on Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 1–28. Springer Verlag, September 1999.

[KB95] Andreas Krall and Thomas Berger. The VAM_{AI} - an abstract machine for incremental global dataflow analysis of Prolog. In Maria Garcia de la Banda, Gerda Janssens, and Peter Stuckey, editors, *ICLP'95 Post-Conference Workshop on Abstract Interpretation of Logic Languages*, pages 80–91, Tokyo, 1995. Science University of Tokyo.

[LdS98] M. Leuschel and D. De Schreye. Constrained Partial Deduction and the Preservation of Characteristic Trees. *New Generation Computing*, 16(3):283–342, 1998.

[LMDS98] Michael Leuschel, Bern Martens, and Danny 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.

[LS96] Michael Leuschel and Morten Heine Sørensen. Redundant argument filtering of logic programs. In John Gallagher, editor, *Logic Program Synthesis and Transformation. Proceedings of LOPSTR'96*, LNCS 1207, pages 83–103, Stockholm, Sweden, August 1996. Springer-Verlag.

[Mar93] André Mariën. *Improving the Compilation of Prolog in the Framework of the Warren Abstract Machine*. PhD thesis, Katholieke Universiteit Leuven, September 1993.

[Per87] F. Pereira. *C-Prolog User's Manual, Version 1.5*. University of Edinburgh, 1987.

[RT96] T. Reps and T. Turnidge. Program Specialization via Program Slicing. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle, Germany, February 1996*, pages 409–429. Springer LNCS 1110, 1996.

[SD96] S. Schoenig and M. Ducasse. A Backward Slicing Algorithm for Prolog. In *Proc. of the Int'l Static Analysis Symposium (SAS'96)*, pages 317–331. Springer LNCS 1145, 1996.

[SGM02] G. Szilagyi, T. Gyimothy, and J. Maluszynski. Static and Dynamic Slicing of Constraint Logic Programs. *J. Automated Software Engineering*, 9(1):41–65, 2002.

[Swe99] Swedish Institute for Computer Science, PO Box 1263, S-164 28 Kista, Sweden. *SICStus Prolog 3.8 User's Manual*, 3.8 edition, October 1999. Available from `http://www.sics.se/sicstus/`.

[Tay91] A. Taylor. *High-Performance Prolog Implementation*. PhD thesis, Basser Department of Computer Science, Unversity of Sidney, June 1991.

[Tip95] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[Van94] P. Van Roy. 1983-1993: The Wonder Years of Sequential Prolog Implementation. *Journal of Logic Programming*, 19/20:385–441, 1994.

[VD92] P. Van Roy and A.M. Despain. High-Performace Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer Magazine*, pages 54–68, January 1992.

[Vas99] W. Vasconcelos. A Flexible Framework for Dynamic and Static Slicing of Logic Programs. In *Proc. of the First Int'l Workshop on Practical Aspects of Declarative Languages (PADL'99)*, pages 259–274. Springer LNCS 1551, 1999.

[Vid03] G. Vidal. Forward Slicing of Multi-Paradigm Declarative Programs Based on Partial Evaluation. In *Logic-based Program Synthesis and Transformation (revised and selected papers from the 12th Int'l Workshop LOPSTR 2002)*, pages 219–237. Springer LNCS 2664, 2003.

[War83] D.H.D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.

[Wei84] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[ZCU01] J. Zhao, J. Cheng, and K. Ushijima. A Program Dependence Model for Concurrent Logic Programs and Its Applications. In *Proc. of IEEE Int'l Conf. on Software Maintenance (ICSM'01)*, pages 672–681. IEEE Press, 2001.

Forward Slicing by Conjunctive Partial Deduction and Argument Filtering*

Michael Leuschel[1] and Germán Vidal[2]

[1] School of Electronics and Computer Science, University of Southampton
& Institut für Informatik, Heinrich-Heine Universität Düsseldorf
mal@ecs.soton.ac.uk
[2] DSIC, Technical University of Valencia,
Camino de Vera S/N, E-46022 Valencia, Spain
gvidal@dsic.upv.es

Abstract. Program slicing is a well-known methodology that aims at identifying the program statements that (potentially) affect the values computed at some point of interest. Within imperative programming, this technique has been successfully applied to debugging, specialization, reuse, maintenance, etc. Due to its declarative nature, adapting the slicing notions and techniques to a logic programming setting is not an easy task. In this work, we define the first, semantics-preserving, forward slicing technique for logic programs. Our approach relies on the application of a conjunctive partial deduction algorithm for a precise propagation of information between calls. We do not distinguish between static and dynamic slicing since partial deduction can naturally deal with both static and dynamic data. A slicing tool has been implemented in ECCE, where a post-processing transformation to remove redundant arguments has been added. Experiments conducted on a wide variety of programs are encouraging and demonstrate the usefulness of our approach, both as a classical slicing method and as a technique for code size reduction.

1 Introduction

Program slicing is a fundamental operation that has been successfully applied to solve many software engineering tasks, like, e.g., program understanding, maintenance, specialization, debugging, reuse, etc. Slicing was originally introduced by Weiser [34]—in the context of imperative programs—as a debugging technique. Despite its potential applications, we found very few approaches to slicing in logic programming (some notable exceptions are, e.g., [10, 28, 29, 31, 35]).

Informally, a *program slice* consists of those program statements which are (potentially) related with the values computed at some program point and/or variable, referred to as a *slicing criterion*. Program slices are usually computed from a *program dependence graph* [5] that makes explicit both the data and control dependences for each operation in a program. Program dependences can

* This work was partially funded by the IST programme of the European Commission, Future and Emerging Technologies under the IST-2001-38059 ASAP project and by the Spanish *Ministerio de Educación y Ciencia* (ref. TIN2004-00231).

be traversed backwards and forwards—from the slicing criterion—giving rise to so-called *backward* and *forward* slicing. Additionally, slices can be *static* or *dynamic*, depending on whether a concrete program's input is provided or not. More detailed information on program slicing can be found in the surveys of Harman and Hierons [12] and Tip [30].

Recently, Vidal [33] introduced a novel approach to forward slicing of lazy functional logic programs. This work exploits the similarities between slicing and partial evaluation—already noticed in [26]—to compute forward slices by a slight modification of an existing partial evaluation scheme [2]. The main requirement of [33] is that the underlying partial evaluation algorithm should be—in the terminology of [27]—both *monovariant* and *monogenetic* in order to preserve the structure of the original program. Unfortunately, this requirement also restricts the precision of the computed slices.

In this work, we extend the approach of [33] in several ways. First, we adapt it to the logic programming setting. Second, we consider a polyvariant and polygenetic partial evaluation scheme: the *conjunctive* partial deduction algorithm of [3] with control based on characteristic trees [9, 19, 20]. Therefore, the computed slices are significantly more precise than those of the previous approach. Basically, conjunctive partial deduction [3] extends the original framework of partial deduction [24] by considering conjunctions of atoms for specialization (hence allowing precise information propagation from one call to another), while a *characteristic tree* [9] is a data structure which encapsulates the evaluation behavior of a goal, i.e., a trace of the unfolding process. This provides a powerful mechanism to guide generalization and polyvariance throughout the transformation process [14, 20]. Also, since each characteristic tree stores the clauses used for the evaluation of a goal, they become very useful to identify the subset of the original program that is reachable from the slicing criterion (i.e., a forward slice). Furthermore, since the basic partial deduction algorithm is kept unmodified, it can easily be implemented on top of an existing partial deduction system (in our case, ECCE [20]). Finally, we use the redundant argument filtering transformation of [22] to slice out unnecessary arguments of predicates (in addition to slicing out entire clauses).

The combination of these two approaches, [33] and [22], together with a special-purpose slicing code generator, gives rise to a simple but powerful forward slicing technique. We also pay special attention to using slicing for code size reduction. Indeed, within the ASAP project [1], we are looking at resource-aware specialization techniques, with the aim of adapting software for pervasive devices with limited resources. We hence also analyze to what extent our approach can be used as an effective code size reduction technique, to reduce the memory footprint of a program.

Our main contributions are the following. We introduce the first, semantics-preserving, forward slicing technique for logic programs that produces executable slices. While traditional approaches in the literature demand different techniques to deal with static and dynamic slicing, our scheme is general enough to produce both static and dynamic slices. In contrast to [33], the restriction to adopt a

monovariant/monogenetic partial evaluation algorithm is not needed. Dropping this restriction is important as it allows us to use more powerful specialization schemes and, moreover, we do not need to modify the basic algorithm, thus easing the implementation of a slicing tool (i.e., only the code generation phase should be changed). We illustrate the usefulness of our approach on a series of benchmarks, and analyze its potential as a code-size reduction technique.

The paper is organized as follows. After introducing some foundations in the next section, Sect. 3 presents our basic approach to the computation of forward slices. Then, Sect. 4 considers the inclusion of a post-processing phase for argument filtering. Section 5 illustrates our technique by means of a detailed example, while Sect. 6 presents an extensive set of benchmarks. Finally, Sect. 7 compares some related works and concludes.

2 Background

In order to keep the paper self-contained, in this section we briefly recall the methodologies involved in our approach to program slicing.

Partial evaluation [13] has been applied to many programming languages, including functional, imperative, object-oriented, logic, and functional logic programming languages. It aims at improving the overall performance of programs by pre-evaluating parts of the program that depend solely on the *static* input.

In the context of logic programming, full input to a program P consists of a goal G and evaluation corresponds to constructing a complete SLDNF-tree for $P \cup \{G\}$. For partial evaluation, the static input takes the form of a goal G' which is more general (i.e., less instantiated) than a typical goal G at runtime. In contrast to other programming languages, one can still execute P for G' and (try to) construct an SLDNF-tree for $P \cup \{G'\}$. However, since G' is not yet fully instantiated, the SLDNF-tree for $P \cup \{G'\}$ is usually infinite and ordinary evaluation will not terminate. A technique which solves this problem is known under the name of *partial deduction* [24]. Its general idea is to construct a finite number of finite, but possibly incomplete[3] SLDNF-trees and to extract from these trees a new program that allows any instance of the goal G' to be executed.

Conjunctive partial deduction (CPD) [3] is an extension of partial deduction that can achieve effects such as *deforestation* and *tupling* [25]. The essence of CPD can be seen in Fig. 1. The so-called global control of CPD generates a set $C = \{C_1, \ldots, C_n\}$ of *conjunctions* whereas the local control generates for each conjunction a possibly incomplete SLDNF-tree τ_i (a process called *unfolding*). The overall goal is to ensure that every leaf conjunction is either an instance of some C_i or can be split up into sub-conjunctions, each of which is an instance of some conjunction in C. This is called the *closedness* condition, and guarantees correctness of the specialized program which is then extracted by:

- generating one specialized predicate per conjunction in C (and inventing a new predicate name for it), and by producing

[3] An SLDNF-tree is *incomplete* if, in addition to success and failure leaves, it also contains leaves where no literal has been selected for a further derivation step.

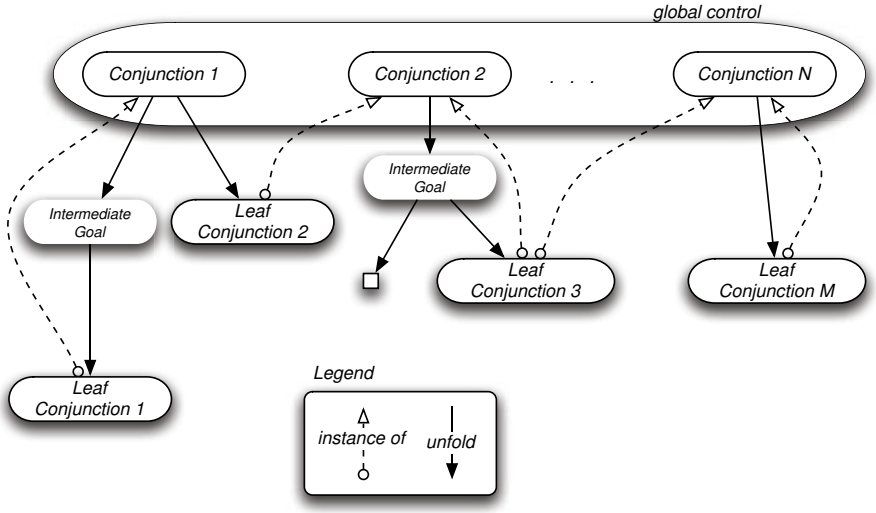

Fig. 1. The Essence of Conjunctive Partial Deduction

– one specialized clause—a *resultant*—per non-failing branch of τ_i.

A single resolution step with a specialized clause now corresponds to performing *all* the resolutions steps (using original program clauses) on the associated branch. Closedness can be ensured by various algorithms [17]. Usually, one starts off with an initial conjunction, unfolds it using some "*unfolding rule*" (a function mapping a program P and a goal G to an SLDNF-tree for $P \cup \{G\}$) and then adds all uncovered[4] leaf conjunctions to C, in turn unfolding them, and so forth. As this process is usually non-terminating, various "*generalization*" operations are applied, which, for example, can replace several conjunctions in C by a single less instantiated one. One useful foundation for the global control is based on so-called *characteristic trees*, used for example by the SP [7] and ECCE [20] specialization systems. We describe them in more detail below, as they turn out to be important for slicing.

Characteristic trees were introduced in partial deduction in order to capture all the relevant aspects of specialization. The following definitions are taken from [20] (which in turn were derived from [9] and the SP system [7]).

Definition 1 (characteristic path). *Let G_0 be a goal, and let P be a normal program whose clauses are numbered. Let $G_0, \ldots, G_n$ be the goals of a finite, possibly incomplete SLDNF-derivation D of $P \cup \{G_0\}$. The characteristic path of the derivation D is the sequence $\langle l_0 : c_0, \ldots, l_{n-1} : c_{n-1} \rangle$, where l_i is the position of the selected literal in G_i, and c_i is defined as follows:*

[4] I.e., those conjunctions which are not an instance of a conjunction in C.

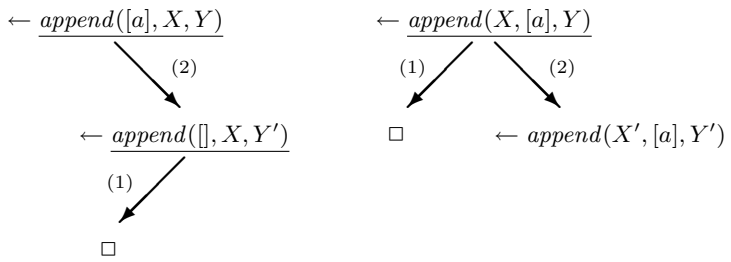

Fig. 2. SLD-trees τ_B and τ_C for Example 1.

- *if the selected literal is an atom, then* c_i *is the number of the clause chosen to resolve with* G_i*;*
- *if the selected literal is* $\neg p(\bar{t})$*, then* c_i *is the predicate* p*.*

Example 1. Let P be the *append* program:

(1) $append([], Z, Z) \leftarrow$
(2) $append([H|X], Y, [H|Z]) \leftarrow append(X, Y, Z)$

Note that we have added clause numbers, which we will henceforth incorporate into illustrations of SLD-trees in order to clarify which clauses have been resolved with. To avoid cluttering the figures we will also drop the substitutions in such figures. For example, the characteristic path of the derivation associated with the only branch of the SLD-tree τ_B in Figure 2 is $\langle 1:2, 1:1 \rangle$.

Note that an SLDNF-derivation D can be either failed, incomplete, successful, or infinite. As we will see below, characteristic paths will only be used to characterize *finite* and *nonfailing* derivations. Once the top-level goal is known, the characteristic path is sufficient to reconstruct all the intermediate goals as well as the final one.[5]

Now that we have characterized derivations, we can characterize goals through the derivations in their associated SLDNF-trees.

Definition 2 (characteristic tree). *Let* G *be a goal,* P *a normal program, and* τ *a finite SLDNF-tree for* $P \cup \{G\}$*. Then the characteristic tree* $\hat{\tau}$ *of* τ *is the set containing the characteristic paths of the nonfailing SLDNF-derivations associated with the branches of* τ*.* $\hat{\tau}$ *is called a characteristic tree if and only if it is the characteristic tree of some finite SLDNF-tree.*

Let U *be an unfolding rule such that* $U(P, G) = \tau$*. Then* $\hat{\tau}$ *is also called the characteristic tree of* G *(in* P*) via* U*. We introduce the notation* $chtree(G, P, U) = \hat{\tau}$*. We also say that* $\hat{\tau}$ *is a characteristic tree of* G *(in* P*) if it is* the *characteristic tree of* G *(in* P*) via some unfolding rule* U*.*

[5] Therefore, using p in the second point of Def. 1 instead of a unique symbol to signal the selection of a negative literal is a matter of convention rather than necessity.

When characteristic trees are used to control CPD, the basic algorithm returns a set of characteristic conjunctions, $\tilde{C}$, that fulfills the conditions for the correctness of the specialization process. A *characteristic conjunction* is a pair $(C, \hat{\tau})$, where C is a conjunction of literals—a goal—and $\hat{\tau} = \mathit{chtree}(C, P, U)$ is a characteristic tree for some program P and unfolding rule U. From this set of characteristic conjunctions, the specialized program is basically obtained by unfolding and renaming.

3 Extracting Executable Forward Slices

In this section, we introduce our approach to the computation of—both static and dynamic—forward slices.

Within imperative programming, the definition of a slicing criterion depends on whether one considers static or dynamic slicing. In the former case, a slicing criterion is traditionally defined as a pair (p, v) where p is a program statement and v is a subset of the program's variables. Then, a forward slice consists of those statements which are dependent on the slicing criterion (i.e., on the values of the variables v that appear in p), a statement being *dependent* on the slicing criterion if the values computed at that statement depend on the values computed at the slicing criterion or if the values computed at the slicing criterion determine if the statement under consideration is executed [30]. As for dynamic slicing, a slicing criterion is often defined as a triple (d, i, v), where d is the input data for the program, i denotes the i-th element of the execution history, and v is a subset of the program's variables.

Adapting these notions to the setting of logic programming is not immediate. There are mainly two aspects that one should take into account:

- The execution of partially instantiated goals—thanks to the use of logic variables—makes it unclear the distinction between static and dynamic slicing.
- The lack of explicit control flow, together with the absence of side effects, makes unnecessary to consider a particular trace of the program's execution for dynamic slicing.

Therefore, we define a *slicing criterion* simply as a goal.[6] Typically, the goal will appear in the code of the source program. However, we lift this requirement for simplicity since it does affect to the forthcoming developments. A forward slice should thus contain a *subset* of the original program with those clauses that are reachable from the slicing criterion. Similarly to [28], the notion of "subset" is formalized in terms of an abstraction relation, to allow arguments to be removed, or rather replaced by a special term:

Definition 3 (term abstraction). *Let $\top_t$ be the empty term (i.e., an unnamed existentially quantified variable, like the anonymous variable of Prolog). A term t is an abstraction of term t', in symbols $t \succeq t'$, iff $t = \top_t$ or $t = t'$.*

[6] If we fix an entry point to the program and restrict ourselves to a particular evaluation strategy (as in Prolog), one can still consider a concrete trace of the program. In this case, however, a standard tracer would suffice to identify the interesting goal.

Definition 4 (literal abstraction). *An atom $p(t_1, \ldots, t_n)$ is an abstraction of atom $q(t'_1, \ldots, t'_m)$, in symbols $p(t_1, \ldots, t_n) \succeq q(t'_1, \ldots, t'_m)$, iff $p = q$, $n = m$, and $t_i \succeq t'_i$ for all $i = 1, \ldots, n$. A negative literal $\neg P$ is an abstraction of a negative literal $\neg Q$ iff $P \succeq Q$.*

Definition 5 (clause abstraction). *A clause c is an abstraction of a clause $c' = L'_0 \leftarrow L'_1, \ldots, L'_n$, in symbols $c \succeq c'$, iff $c = L_0 \leftarrow L_1, \ldots, L_n$ and $L_i \succeq L'_i$ for all $i \in \{1, \ldots, n\}$.*

Definition 6 (program abstraction). *A normal program[7] $P = (c_1, \ldots, c_n)$ is an abstraction of normal program $P' = (c'_0, \ldots, c'_m)$, in symbols $P \succeq P'$, iff $n \leq m$ and there exists a subsequence $(s_1, \ldots, s_n)$ of $(1, \ldots, m)$ such that $c_i \succeq c'_{s_i}$ for all $i \in \{1, \ldots, n\}$.*

Informally, a program P is an abstraction of program P' if it can be obtained from P' by clause deletion and by replacing some predicate arguments by the empty term $\top_t$. In the following, P is a *slice* of program P' iff $P \succeq P'$. Trivially, program slices are normal programs.

Now, we can formally define the notion of *correct* slice:

Definition 7 (correct slice). *Let P be a program and G a slicing criterion. A program P' is a correct slice of P w.r.t. G iff P' is a slice of P (i.e., $P' \succeq P$) and the following conditions hold:*

- *$P \cup \{G\}$ has an SLDNF-refutation with computed answer θ if and only if $P' \cup \{G\}$ does, and*
- *$P \cup \{G\}$ has a finitely failed SLDNF-tree if and only if $P' \cup \{G\}$ does.*

Traditional approaches to program slicing rely on the construction of some data structure which reflects the data and control dependences in a program (like, e.g., the *program dependence graphs* of [5, 15]). The key contribution of this paper is to show that CPD can actually play such a role.

Roughly speaking, our slicing technique proceeds as follows. Firstly, given a program P and a goal G, a CPD algorithm based on characteristic trees is applied. The use of characteristic trees is relevant in our context since they record the clauses used during the unfolding of each conjunction. The complete algorithm outputs a so-called *global tree*—where each node is a characteristic conjunction—which represents an abstraction of the execution of the considered goal. In fact, this global tree contains information which is similar to that in a program dependence graph (e.g., dependences among predicate calls). In standard conjuntive partial deduction, the characteristic conjunctions, $\tilde{C}$, in the computed global tree are unfolded—following the associated characteristic trees—to produce a correct specialization of the original program (after renaming). In order to compute a forward slice, only the code generation phase of the CPD algorithm should be changed: now, we use the characteristic tree of each conjunction in $\tilde{C}$ to determine which clauses of the original program have been used and, thus, should appear in the slice.

[7] We consider that programs are *sequences* of clauses in order to enforce the preservation of the syntax of the original program.

Given a characteristic path δ, we define $cl(\delta)$ as the set of clause numbers in this path, i.e., $cl(\delta) = \{c \mid \langle l : c\rangle$ appears in δ and c is a clause number$\}$. Program slices are then obtained from a set of characteristic trees as follows:

Definition 8 (forward slicing). *Let P be a normal program and G be a slicing criterion. Let $\tilde{C}$ be the output of the CPD algorithm (a set of characteristic conjunctions) and T be the characteristic trees in $\tilde{C}$. A forward slice of P w.r.t. G, denoted by $slice_T(P)$, contains those clauses of P that appear in some characteristic path of T. Formally, $slice_T(P) = \cup_{\hat{\tau}\in T} \{cl(\delta) \mid \delta \in \hat{\tau}\}$.*

The correctness of the forward slicing method is stated as follows:

Theorem 1. *Let P be a normal program and G be a slicing criterion. Let P' be a forward slice according to Def. 8. Then, P' is a correct slice of P w.r.t. G.*

Proof. First, since P' is obtained from P only by clause deletion, P' is trivially a slice of P (i.e., $P' \succeq P$).

Now, the proof proceeds as follows. Let P be a program and G a goal. Assume that a CPD of P w.r.t. G returns the specialized program P'' and that the associated slice for P w.r.t. G is P'. Then, we show that P'' can also be obtained by CPD of the slice, P', w.r.t. G. Then, correctness of the slice follows straightforwardly by the correctness of CPD. Let us now formalize this proof scheme.

According to Def. 8, we consider a set of characteristic conjunctions, $\tilde{C}$, as the output of the CPD algorithm, where T is the set of characteristic trees in $\tilde{C}$. Assume that P'' is the specialized program produced by the code generator of the standard CPD algorithm. By the correctness of this algorithm [3], $P \cup \{G\}$ has an SLDNF-refutation with computed answer θ if and only if $P'' \cup \{G\}$ does, and $P \cup \{G\}$ has a finitely failed SLDNF-tree if and only if $P'' \cup \{G\}$ does.

Let $P' = slice_T(P)$ be the associated forward slice of P w.r.t. G. Now, we only need to show that P'' can also be obtained by CPD of the slice P' w.r.t. G. Actually, this is an immediate consequence of Def. 2 by considering the same unfolding rule that was used in the CPD algorithm. Therefore, by the correctness of this algorithm [3], $P' \cup \{G\}$ has an SLDNF-refutation with computed answer θ if and only if $P'' \cup \{G\}$ does, and $P' \cup \{G\}$ has a finitely failed SLDNF-tree if and only if $P'' \cup \{G\}$ does.

Trivially, we have that $P \cup \{G\}$ has an SLDNF-refutation with computed answer θ if and only if $P' \cup \{G\}$ does, and $P \cup \{G\}$ has a finitely failed SLDNF-tree if and only if $P' \cup \{G\}$ does, which proves the claim.

Our slicing technique produces *correct* forward slices and, moreover, is more flexible than previous approaches in the literature. In particular, in can be used to perform both dynamic and static forward slicing with a modest implementation effort, since only the code generation phase of the CPD algorithm should be changed. Furthermore, it also subsumes *quasi* static slicing, an attempt to define a hybrid method ranging between static and dynamic slicing [32] (which becomes useful when only the value of some parameters is known).

4 Improving Forward Slices by Argument Filtering

The method of Def. 8 has been fully implemented in ECCE, an off-the-shelf partial evaluator for logic programs based on CPD and characteristic trees. In practice, however, we found that computed slices often contain redundant arguments that are not relevant for the execution of the slicing criterion. In order to further refine the computed slices and be able to slice out unnecessary arguments of predicates, we use the redundant argument filtering transformations (RAF) of [22].

RAF is a technique developed in [22] which detects certain redundant arguments (finding all redundant arguments is undecidable in general [22]). Basically, it detects those arguments which are existential and which can thus be safely removed. RAF is very useful when performed after CPD. Redundant arguments also arise when one re-uses generic predicates for more specific purposes. For instance, let us define a `member/2` predicate by re-using a generic `delete/3` predicate:

```
member(X,L) :- delete(X,L,DL).
delete(X,[X|T],T).     delete(X,[Y|T],[Y|DT]) :- delete(X,T,DT).
```

Here, the third argument of `delete` is redundant and will be removed by the partial evaluator ECCE if RAF is enabled:

```
member(X,L) :- delete(X,L).
delete(X,[X|T]).       delete(X,[Y|T]) :- delete(X,T).
```

The ECCE system also contains the reverse argument filtering (FAR) of [22] ("reverse" because the safety conditions are reversed w.r.t. RAF). While RAF detects existential arguments (which might return a computed answer binding), FAR detects arguments which can be non-existential and non-ground but whose value is never used (and for which no computed answer binding will be returned). Consider, e.g., the following program:

```
p(X) :-  q(f(X)).     q(Z).
```

Here, the argument of `q(f(X))` is not a variable but the value is never used. The ECCE system will remove this argument if FAR is enabled:

```
p(X) :-  q.     q.
```

The elimination of redundant arguments turns out to be quite useful to remove unnecessary arguments from program slices (see next section). Only one extension is necessary in our context: while redundant arguments are deleted in [22], we replace them by the special symbol $\top_t$ so that the filtered program is still a slice—an abstraction—of the original program. The correctness of the extended slicing algorithm then follows from Theorem 1 and the results in [22].

```
int(cst(X),_,_,X).
int(var(X),Vars,Vals,R) :- lookup(X,Vars,Vals,R).
int(plus(X,Y),Vars,Vals,Res) :-
    int(X,Vars,Vals,RX), int(Y,Vars,Vals,RY), Res is RX+RY.
int(minus(X,Y),Vars,Vals,Res) :-
    int(X,Vars,Vals,RX), int(Y,Vars,Vals,RY), Res is RX-RY.
int(fun(X),Vars,Vals,Res) :-
  def0(X,Def), int(Def,Vars,Vals,Res).
int(fun(X,Arg),Vars,Vals,Res) :-
    def1(X,Var,Def), int(Arg,Vars,Vals,ResArg),
    int(Def,[Var|Vars],[ResArg|Vals],Res).
def0(one,cst(1)).
def0(rec,fun(rec)).
def1(inc,xx,plus(var(xx),cst(1))).
def1(rec,xx,fun(rec,var(xx))).
lookup(X,[X|_],[Val|_],Val).
lookup(X,[Y|T],[_|ValT],Res) :- X \= Y, lookup(X,T,ValT,Res).
```

Fig. 3. A simple functional interpreter

5 Forward Slicing in Practice

In this section, we illustrate our approach to the computation of forward slices through some selected examples. Consider the program in Fig. 3 which defines an interpreter for a simple language with constants, variables, and some predefined functions. First, we consider the following slicing criterion:

```
slice1(X) :- int(minus(cst(4),plus(fun(one),cst(2))),[xx],[11],X).
```

The slice computed by ECCE w.r.t. this slicing criterion is as follows:

```
int(cst(X),_,_,X).

int(plus(X,Y),Vars,Vals,Res) :-
   int(X,Vars,Vals,RX), int(Y,Vars,Vals,RY), Res is RX+RY.
int(minus(X,Y),Vars,Vals,Res) :-
   int(X,Vars,Vals,RX), int(Y,Vars,Vals,RY), Res is RX-RY.
int(fun(X),Vars,Vals,Res) :- def0(X,Def), int(Def,Vars,Vals,Res).

def0(one,cst(1)).

slice1(X) :- int(minus(cst(4),plus(fun(one),cst(2))),[xx],[11],X).
```

Here, some predicates have been completely removed from the slice (e.g., `def1` or `lookup`), even though they are reachable in the predicate dependency graph. Furthermore, unused clauses are also removed, cutting down further the size of the slice. By applying the argument filtering post-processing, we get[8]

[8] For clarity, in the examples we use "$*$" to denote the empty term $\top_t$. In practice, empty terms can be replaced by any term since they play no role in the computation.

```
int(cst(X),*,*,X).

int(plus(X,Y),*,*,Res) :- int(X,*,*,RX), int(Y,*,*,RY), Res is RX+RY.
int(minus(X,Y),*,*,Res) :- int(X,*,*,RX), int(Y,*,*,RY), Res is RX-RY.
int(fun(X),*,*,Res) :- def0(X,Def), int(Def,*,*,Res).

def0(one,cst(1)).

slice1(X) :- int(minus(cst(4),plus(fun(one),cst(2))),*,*,X).
```

The resulting slice is executable and will produce the same result as the original program:

```
| ?- slice1(X).
X = 1 ?
yes
```

Note that this example could have been tackled by a *dynamic* slicing method, as a fully specified query was provided as the slicing criterion. It would be interesting to know how a dynamic slicer would compare against our technique, and whether we have lost any precision. In order to test this, we have implemented a simple dynamic slicer in SICStus Prolog using profiled code and extracting the used clauses using the `profile_data/4` built-in. The so extracted slice corresponds exactly to our result (without the argument filtering; see Fig. 4), and hence no precision has been lost in this example.

In general, not only the code size of the slice is smaller but also the runtime can be improved. Thus, our forward slicing algorithm can be seen as a—rather conservative—partial evaluation method that guarantees that code size does not increase. For instance, it can be useful for resource aware specialization, when the (potential) code explosion of typical partial evaluators is unacceptable.

Our slicing tool can also be useful for program debugging. In particular, it can help the programmer to locate the source of an incorrect answer (or an unexpected loop; finite failure is preserved in Def. 7) since it identifies the clauses that could affect the computed answer, thus easing the correction of the program. Consider, e.g., that the definition of function `plus` contains a bug:

```
int(plus(X,Y),Vars,Vals,Res) :-
    int(X,Vars,Vals,RX), int(Y,Vars,Vals,RY), Res is RX-RY.
```

i.e., the programmer wrote `RX-RY` instead of `RX+RY`. Given the following goal:

```
slice2(X) :- int(plus(cst(1),cst(2)),[x],[1],X).
```

the execution returns the—incorrect—computed answer `X = -1`. By computing a forward slice w.r.t. `slice2(X)`, we get (after argument filtering) the following:

```
int(cst(X),*,*,X).
int(plus(X,Y),*,*,Res) :- int(X,*,*,RX), int(Y,*,*,RY), Res is RX-RY.
slice2(X) :- int(plus(cst(1),cst(2)),*,*,X).
```

Fig. 4. The Dynamic Slicer using Profiled Code (for comparing precision with our approach)

This slice contains only 3 clauses and, thus, the user can easily detect that the definition of `plus` is wrong.

The previous two slices can be extracted by a dynamic slicing technique, since they do not involve a non-terminating goal. Now, we consider the following slicing criterion:

```
slice3(X) :- int(fun(rec),[aa,bb,cc,dd],[0,1,2,3],X).
```

Despite the fact that this goal has an infinite search space, our slicing tool returns the following slice (after argument filtering):

```
int(fun(X),*,*,*) :- def0(X,Def), int(Def,*,*,*).
def0(rec,fun(rec)).
slice3(X) :- int(fun(rec),*,*,*).
```

From this slice, the clauses which are responsible of the infinite computation can easily be identified.

6 Experimental Results

In this section, we show a summary of the experiments conducted on an extensive set of benchmarks. We used SICStus Prolog 3.11.1 (powerpc-darwin-7.2.0) and Ciao-Prolog 1.11 #221, running on a Powerbook G4, 1GHz, 1GByte of RAM. The operating system was Mac OS 10.3. We also ran some experiments with SWI Prolog 5.2.0. The runtime was obtained by special purpose benchmarker files (generated automatically be ECCE) which execute the original and specialized programs without loop overhead. The code size was obtained by using the `fcompile` command of SICStus Prolog and then measuring the size of the compiled `*.ql` files. The total speedups were obtained by the formula $\frac{n}{\sum_{i=1}^{n} \frac{spec_i}{orig_i}}$ where n is the number of benchmarks, and $spec_i$ and $orig_i$ are the absolute execution times of the specialized/sliced and original programs respectively.[9] The total code size reduction was obtained by the formula $1 - \frac{\sum_{i=1}^{n} specsz_i}{\sum_{i=1}^{n} origsz_i}$ where n is the number of benchmarks, and $specsz_i$ and $origsz_i$ are the code sizes of the specialized/sliced and original programs respectively.

DPPD. We first compared the slicing tool with the default conjunctive specialization of ECCE on the DPPD library of specialization benchmarks [16]. In a sense these are not typical slicing scenarios, but nonetheless give an indication of the behavior of the slicing algorithm. The experiments also allow us to evaluate to what extent our technique is useful as an alternative way to specialize programs, especially for code size reduction. Finally, the use of the DPPD library allows comparison with other reference implementations (see, e.g., [3, 14] for comparisons with MIXTUS, SP and PADDY).

Note that some of the benchmark programs contain negation and built-ins. To handle negation, we actually had to extend our algorithm to generate failing declarations (`p(_,...,_) :- fail.`) for those predicates that are selected negatively but for which no clauses otherwise appear in the slice. From the perspective of the SLDNF semantics this is not required, but it is important to avoid generating runtime exceptions (calling undefined predicates). Finally, to handle the built-ins nothing had to be added on top of ECCE's treatment of built-ins.[10]

[9] Observe that this is different from the average of the speedups (which has the disadvantage that big slowdowns are not penalized sufficiently).

[10] Even higher-order predicates such as `call/1` in map.rev and map.reduce are not a problem as long as at specialization time they are sufficiently instantiated. ECCE will print a warning if the original program has to be kept (e.g., because of something like `call(X)` in the specialized program) and in that case no clause can be sliced away (as we have no information about `X`).

Table 1. Speedups obtained by Specialization and by Slicing

Prolog System	SWI-Prolog		SICStus		Ciao	
Technique	Specialized	Sliced	Specialized	Sliced.	Specialized	Sliced
TOTAL	2.43	1.04	2.74	1.04	2.62	1.05
Average	5.23	1.07	6.27	1.09	11.26	1.09

Table 1 (summary of Table 4 in Appendix A) shows the speedup of the ECCE default specialization and of our slicing algorithm. Timings for SWI Prolog, SICStus Prolog, and Ciao Prolog are shown. It can be seen that the average speedup of slicing is just 4%. This shows how efficient modern Prolog implementations are, and that little overhead has to be paid for adding extra clauses to a program. (One can also see that SICStus runs roughly 4 times faster than SWI.) There are, however, a few benchmarks where slicing does lead to a significant speedup (e.g., for model-elim), probably due to the fact that the removal of clauses also removes unnecessary backtracking. Anyway, the main purpose of slicing is not speedup, but reducing code size. In this case, slicing has managed an overall code size reduction of 26.2% whereas the standard specialization has increased the code size by 56%. As can be seen in Table 7 of Appendix A, in the worst case, the specialization has increased the code size by 493.5% (whereas slicing never increases the code size). On the other hand, there are cases where specialization achieves much smaller code size than slicing, e.g., for ssuply where the specializer has managed to (almost) fully unfold the specialized query, thus producing basically a series of facts.

Slicing-Specific Benchmarks. Let us now turn our attention to four, more slicing-specific experiments. Table 2 contains the results of these experiments. The `inter_medium` benchmark is the simple interpreter of Sect. 5. The `ctl_trace` benchmark is the CTL model checker from [21], extended to compute witness traces. It is sliced for a particular system and temporal logic formula to model check. The `lambdaint` benchmark is an interpreter for a simple functional language taken from [18]. It is sliced for a particular functional program (computing the Fibonacci numbers). Finally, `matlab` is an interpreter for a subset of the Matlab language (the code can be found in Appendix B). The overall results are very good: the code size is reduced by 60.5% and runtime decreased by 16%.

Comparing the Influence of Local and Global Control. In Table 3, we compare the influence of the partial deduction control. Here, "Full slicing" is the standard CPD that we have used so far; "Simple Std. PD" is a standard (non-conjunctive) partial deduction with relatively simple control; and "Naïve PD" is very simple standard partial deduction in the style of [33], i.e., with a one-step unfolding and very simple generalization (although it is still more precise than [33] as it can produce some polyvariance), where we have turned the redundant argument filtering off.

Table 2. Slicing Specific Benchmarks

	Slicing Time	Runtime		Size		
Benchmark	ms	Original ms	Sliced speedup	Original Bytes	Sliced Bytes	Reduction %
inter_medium	20	117	1.06	4798	1578	67.1%
lambdaint	390	177	1.29	7389	4769	35.5%
ctl_trace	1940	427	1.35	8053	4214	47.7%
matlab	2390	1020	1.02	27496	8303	69.8%
Total			1.16			60.5%

Table 3. Various Slicing Approaches

	Full Slicing		Simple Std. PD		Naïve PD	
Benchmark	Time (ms)	Reduction	Time (ms)	Reduction	Time (ms)	Reduction
inter_medium	20	67.1%	50	67.1%	20	41.8%
lambdaint	390	35.5%	880	9.0%	30	9.0%
ctl_trace	1940	47.7%	140	47.7%	40	1.3%
matlab	2390	69.8%	1170	69.8%	200	19.3%
Total	4740	60.5%	2240	56.4%	290	17.0%

Table 6 in Appendix A shows the clear difference between our slicing approach and one using a naïve PD on the DPPD benchmarks used earlier: our approach manages a code size reduction of 26% whereas the naïve PD approach manages just 9.4%. The table also shows that the filtering had an impact on three benchmarks, but overall the impact of the filtering is quite small. This is somewhat surprising, and may be due to the nature of the benchmarks. However, it may also mean that in the future we have to look at more powerful filtering approaches.

7 Discussion, Related and Future Work

In this work, we have introduced the first, semantics-preserving, forward slicing technique for logic programs. Traditional approaches to program slicing rely on the construction of some data structure to store the data and control dependences in a program. The key contribution of this paper has been to show that CPD can actually play such a role. The main advantages of this approach are the following: there is no need to distinguish between static and dynamic slicing and, furthermore, a slicing tool can be fully implemented with a modest implementation effort, since only the final code generation phase should be changed (i.e., the core algorithm of the partial deduction system remains untouched). A slicing tool has been fully implemented in ECCE, where a post-processing transformation to remove redundant arguments has been added. Our experiments demonstrate the usefulness of our approach, both as a classical slicing method as well as a technique for code size reduction.

As mentioned before, we are not aware of any other approach to forward slicing of logic programs. Previous approaches have only considered *backward* slicing. For instance, Schoening and Ducassé [28] defined the first backward slicing algorithm for Prolog which produces executable programs. Vasconcelos [31] introduced a flexible framework to compute both static and dynamic backward slices. Similar techniques have also been defined for constraint logic programs [29] and concurrent logic programs [35]. Within imperative programming, Field, Ramalingam, and Tip [6] introduced a *constrained* slicing scheme in which source programs are translated to an intermediate graph representation. Similarly to our approach, constrained slicing generalizes the traditional notions of static and dynamic slicing since arbitrary constraints on the input data can be made.

The closest approaches are those of [33] and [22]. Vidal [33] introduced a forward slicing method for lazy functional logic programs that exploits the similarities between slicing and partial evaluation. However, only a restrictive form of partial evaluation—i.e., monovariant and monogenetic partial evaluation—is allowed, which also restricts the precision of the computed slices. Our new approach differs from that of [33] in several aspects: we consider logic programs; we use a polyvariant and polygenetic partial evaluation scheme and, therefore, the computed slices are significantly more precise; and, moreover, since the basic partial deduction algorithm is kept unmodified, it can easily be implemented on top of an existing partial deduction system. On the other hand, Leuschel and Sørensen [22] introduced the concept of *correct erasure* in order to detect and remove redundant arguments from logic programs. They present a constructive algorithm for computing correct erasures which can be used to perform a simple form of slicing. In our approach, we use this algorithm as a post-processing phase to slice out unnecessary arguments of predicates in the computed slices. The combination of these two approaches, [33] and [22], together with a special-purpose slicing code generator, form the basis of a powerful forward slicing technique.

Since our work constitutes a first step towards the development of a forward slicing technique for logic programs, there are many interesting topics for future work. For instance, our slicing technique is *syntax-preserving*, i.e., the computed slice is a fragment of the original program (which is useful for debugging purposes). In contrast, *amorphous* slicing [11] exploits different program transformations in order to simplify the program while preserving its semantics w.r.t. the slicing criterion. Thus, it would be interesting to extend our technique along this line in order to obtain smaller program slices. Another possibility for future work involves the computation of *backward* slices (a harder topic). In this case, the information gathered by characteristic trees is not enough and some extension is needed.

One should also investigate to what extent abstract interpretation can be used to complement our slicing technique. On its own, abstract interpretation will probably lack the precise propagation of concrete values, hence making it less suitable for dynamic slicing. However, for static slicing it may be able to remove certain clauses that a partial deduction approach cannot remove (see, e.g., [4, 8] where useless clauses are removed to complement partial deduction)

and one should investigate this possibility further. One could also investigate better global control, adapted for slicing (to avoid wasted specialisation effort in case added polyvariance does not increase the precision of the slice). Finally, we can use our slicing technique as a starting point for resource aware specialization, i.e., finding a good tradeoff between code size and execution speed.

Acknowledgements. We would like to thank the anonymous referees of ESOP'05 for their valuable feedback. We also would like to thank Dan Elphick, John Gallagher, Germán Puebla, and all the other partners of the ASAP project for their considerable help and input.

References

1. Advanced Specialization and Analysis for Pervasive Computing. EU IST FET Programme Project Number IST-2001-38059. `http://www.asap.ecs.soton.ac.uk/`
2. E. Albert and G. Vidal. The Narrowing-Driven Approach to Functional Logic Program Specialization. *New Generation Computing*, 2(1):3–26, 2002.
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, 1999.
4. D. A. de Waal and J. Gallagher. The applicability of logic program analysis and transformation to theorem proving. In A. Bundy, editor, *Automated Deduction—CADE-12*, pages 207–221. Springer-Verlag, 1994.
5. J. Ferrante, K. Ottenstein, and J. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.
6. J. Field, G. Ramalingam, and F. Tip. Parametric Program Slicing In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 379-392, ACM Press, 1995.
7. J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.
8. J. Gallagher and D. A. de Waal. Deletion of redundant unary type predicates from logic programs. In K.-K. Lau and T. Clement, editors, Logic Program Synthesis and Transformation. *Proc. of LOPSTR'92*, pages 151–167, Manchester, UK, 1992.
9. J. Gallagher and M. Bruynooghe. The Derivation of an Algorithm for Program Specialisation. *New Generation Computing*, 9(3-4):305–333, 1991.
10. T. Gyimóthy and J. Paakki. Static Slicing of Logic Programs. In *Proc. of the 2nd Int'l Workshop on Automated and Algorithmic Debugging (AADEBUG'95)*, pages 87–103. IRISA-CNRS, 1995.
11. M. Harman and S. Danicic. Amorphous Program Slicing. In *Proc. of the 5th Int'l Workshop on Program Comprehension*. IEEE Computer Society Press, 1997.
12. M. Harman and R. Hierons. An Overview of Program Slicing. *Software Focus*, 2(3):85–92, 2001.
13. N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
14. J. Jørgensen, M. Leuschel, and B. Martens. Conjunctive partial deduction in practice. In J. Gallagher, editor, *Logic Program Synthesis and Transformation. Proceedings of LOPSTR'96*, LNCS 1207, pages 59–82. Springer-Verlag, 1996.

15. D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimization. In *Proc. of the 8th Symp. on the Principles of Programming Languages (POPL'81), SIGPLAN Notices*, pages 207–218, 1981.
16. M. Leuschel. The DPPD Library of Benchmarks. Accessible from URL: `http://www.ecs.soton.ac.uk/~mal/systems/dppd.html`
17. M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.
18. 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*, pages 341–376. Springer LNCS 3049, 2004.
19. M. Leuschel and D. De Schreye. Constrained Partial Deduction and the Preservation of Characteristic Trees. *New Generation Computing*, 16(3):283–342, 1998.
20. 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.
21. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi, editor, Logic-Based Program Synthesis and Transformation. *Proceedings of LOPSTR'99*, LNCS 1817, pages 63–82, Venice, Italy, 2000.
22. M. Leuschel and M. H. Sørensen. Redundant argument filtering of logic programs. In J. Gallagher, editor, *Logic Program Synthesis and Transformation. Proceedings of LOPSTR'96*, LNCS 1207, pages 83–103. Springer-Verlag, 1996.
23. M. Leuschel and G. Vidal. Forward Slicing by Conjunctive Partial Deduction and Argument Filtering. Technical Report, DSSE, University of Southamtpon, December 2004.
24. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3& 4):217–242, 1991.
25. A. Pettorossi and M. Proietti. Transformation of Logic Programs: Foundations and Techniques. *The Journal of Logic Programming*, 19,20:261–320, 1994.
26. T. Reps and T. Turnidge. Program Specialization via Program Slicing. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle*, pages 409–429. Springer LNCS 1110, 1996.
27. R. Glück and M.H. Sørensen A Roadmap to Metacomputation by Supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle*, pages 137–160. Springer LNCS 1110, 1996.
28. S. Schoenig and M. Ducassé. A Backward Slicing Algorithm for Prolog. In *Proc. of the Int'l Static Analysis Symposium (SAS'96)*, pages 317–331. Springer LNCS 1145, 1996.
29. G. Szilagyi, T. Gyimothy, and J. Maluszynski. Static and Dynamic Slicing of Constraint Logic Programs. *J. Automated Software Engineering*, 9(1):41–65, 2002.
30. F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.
31. W. Vasconcelos. A Flexible Framework for Dynamic and Static Slicing of Logic Programs. In *Proc. of the First Int'l Workshop on Practical Aspects of Declarative Languages (PADL'99)*, pages 259–274. Springer LNCS 1551, 1999.
32. G. Venkatesh. The Semantic Approach to Program Slicing. *SIGPLAN Notices*, 26(6):107–119, 1991. *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'91).*

33. G. Vidal. Forward slicing of multi-paradigm declarative programs based on partial evaluation. In M. Leuschel, editor, *Proc. of Logic-based Program Synthesis and Transformation (LOPSTR'02)*, LNCS 2664, pages 219–237. Springer-Verlag, 2003.
34. M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.
35. J. Zhao, J. Cheng, and K. Ushijima. A Program Dependence Model for Concurrent Logic Programs and Its Applications. In *Proc. of IEEE Int'l Conf. on Software Maintenance (ICSM'01)*, pages 672–681. IEEE Press, 2001.

A Additional Tables

Table 4. Speedups obtained by Specialization and by Slicing

	SWI-Prolog			SICStus			Ciao		
Benchmark	Orig. ms	Spec. speed	Sliced speed	Orig. ms	Spec. speed	Sliced speed	Orig. ms	Spec. speed	Sliced speed
advisor	697	4.02	0.99	250	3.95	1.03	247	2.96	1.03
applast	503	3.36	0.99	140	5.25	0.98	140	3.50	1.05
contains.kmp	757	8.73	1.00	277	9.22	1.01	250	7.50	0.84
depth.lam	337	6.73	1.01	117	8.75	1.00	123	6.17	1.06
doubleapp	353	1.66	0.98	63	1.46	0.86	83	1.47	1.09
ex_depth	190	3.80	1.00	67	3.33	1.05	63	2.71	1.00
flip	150	1.61	1.05	30	1.13	0.75	43	1.30	1.00
groundunify.simple	543	7.09	1.22	193	9.67	1.38	200	4.62	1.36
groundunify.complex	360	4.15	0.97	123	3.70	0.97	140	3.23	1.05
imperative-solve	1160	2.32	1.02	430	2.87	1.02	13133	66.78	1.02
liftsolve.app	1710	34.20	1.00	620	37.20	1.04	593	22.25	0.99
liftsolve.lmkng	160	1.17	1.02	60	1.29	1.06	57	1.13	1.13
map.reduce	940	3.71	0.99	337	6.31	0.98	5520	92.00	1.02
map.rev	393	2.41	1.00	140	4.67	1.00	2013	43.14	1.05
matchapp	323	1.94	0.99	130	3.55	0.95	127	2.53	1.03
match.kmp	487	1.85	0.97	160	2.82	1.00	173	2.08	0.98
model_elim	1120	3.82	2.40	443	7.82	2.96	440	6.29	2.49
regexp.r1	2010	4.71	1.02	623	5.67	1.04	600	4.39	1.00
regexp.r2	583	3.30	1.02	183	3.67	1.04	183	3.06	0.98
regexp.r3	1173	3.29	1.02	363	3.52	1.03	360	3.09	1.03
relative	2923	15.95	1.00	730	21.90	1.02	827	16.53	0.98
remove	467	1.43	1.02	157	1.62	1.00	197	1.59	1.09
remove2	667	1.33	1.00	193	1.57	1.02	227	1.36	1.05
rev	1280	0.87	1.00	220	0.88	0.97	247	0.93	0.95
rev_acc_type	3643	1.00	1.00	523	1.12	1.01	627	1.03	0.99
rotateprune	993	1.01	0.99	297	1.13	1.00	317	1.06	0.99
ssuply	1087	18.11	1.34	333	20.00	1.49	343	14.71	1.41
transpose	763	7.63	1.00	170	4.64	1.00	213	4.57	1.00
vanilla.doubleapp	877	3.42	1.01	247	6.73	1.01	273	4.56	1.00
TOTAL		2.43	1.04		2.74	1.04		2.62	1.05
Average		5.23	1.07		6.27	1.09		11.26	1.09

Table 5. Compiled Code Size in SICStus

Benchmark	Original	Specialized		Sliced	
	Bytes	Bytes	% of original	Bytes	% of original
advisor	3850	1936	50.3%	2320	60.3%
applast	942	844	89.6%	942	100.0%
doubleapp	656	1264	192.7%	656	100.0%
ex_depth	2295	1596	69.5%	1193	52.0%
flip	774	1628	210.3%	774	100.0%
groundunify.complex	5881	34901	593.5%	5881	100.0%
groundunify.simple	5881	894	15.2%	2537	43.1%
imperative-solve	7514	17417	231.8%	5424	72.2%
liftsolve.app	2919	1723	59.0%	2381	81.6%
liftsolve.lmkng	2919	3795	130.0%	1502	51.5%
map.reduce	2385	1489	62.4%	1459	61.2%
map.rev	2385	1266	53.1%	1088	45.6%
match.kmp	824	1772	215.0%	824	100.0%
matchapp	678	1587	234.1%	678	100.0%
model_elim	3703	1615	43.6%	2199	59.4%
regexp.r1	1313	1621	123.5%	1204	91.7%
regexp.r2	1313	2486	189.3%	1204	91.7%
regexp.r3	1313	1989	151.5%	1204	91.7%
relative	2113	3947	186.8%	2113	100.0%
remove	965	4222	437.5%	965	100.0%
remove2	1244	2429	195.3%	1244	100.0%
rev	790	943	119.4%	790	100.0%
rev_acc_type	781	945	121.0%	783	100.3%
rotateprune	1622	2635	162.5%	1622	100.0%
ssuply	5391	768	14.2%	3583	66.5%
transpose	1091	2182	200.0%	1091	100.0%
vanilla.doubleapp	1896	1069	56.4%	1185	62.5%
TOTAL	Bytes		156.0%		73.8%

Table 6. Code Size Reduction: Various Slicing approaches on DPPD

Benchmark	Full Slicing	Full Slicing wo RAF/FAR	Naïve PD
advisor	39.7%	39.7%	32.8%
applast	0.0%	0.0%	0.0%
doubleapp	0.0%	0.0%	0.0%
ex_depth	48.0%	48.0%	0.0%
flip	0.0%	0.0%	0.0%
groundunify.complex	0.0%	0.0%	0.0%
groundunify.simple	56.9%	**55.7**%	0.0%
imperative-solve	27.8%	27.8%	0.0%
liftsolve.app	18.4%	18.4%	18.4%
liftsolve.lmkng	48.5%	48.5%	48.5%
map.reduce	38.8%	38.8%	38.8%
map.rev	54.4%	54.4%	54.4%
match.kmp	0.0%	0.0%	0.0%
matchapp	0.0%	0.0%	0.0%
model_elim	40.6%	**38.4**%	5.2%
regexp.r1	8.3%	8.3%	0.0%
regexp.r2	8.3%	8.3%	0.0%
regexp.r3	8.3%	8.3%	0.0%
relative	0.0%	0.0%	0.0%
remove	0.0%	0.0%	0.0%
remove2	0.0%	0.0%	0.0%
rev	0.0%	0.0%	0.0%
rev_acc_type	-0.3%	-0.3%	-0.3%
rotateprune	0.0%	0.0%	0.0%
ssuply	33.5%	**32.0**%	6.7%
transpose	0.0%	0.0%	0.0%
vanilla.doubleapp	37.5%	37.5%	0.0%
TOTAL	26.2%	25.8%	9.4%

B The Matlab Slicing Benchmark

In this appendix we present the full code for the largest of our slicing benchmarks: an interpreter for a subset of the Matlab language. The interpreter was written by Daniel Elphick.

The specialisation/slicing query is the call `run1(_)`, whereas the runtime queries are`run1([])`, `run1([1])`.

```
% specialisation goals : run1([X]), run12([X]), run2([X]), run22([X])
% run1 is factorial using a while loop
% run2 is factorial using recursion
% run12 and run22 use the parser but should produce exactly the same results

% to run try run12([const(5)])
% should eventually print 120

eval_mfile(Text, Params, Nargout) :- load, parse(Text, MFile), !,
eval_matlab(MFile, Params, Nargout).

% NB: The following line should point to the absolute path of matlab_parser.pl
load :- use_module('~/prolog/cogen2/examples/matlab/matlab_parser.pl').

eval_matlab([F|Funcs], Pars, 1) :-
   eval_function(F, Pars, 1, [R], [F|Funcs]).

eval_function(function(_, Rets, Pars, Vars, Comms), Params, Nargout, Values, Funcs) :-
  bind_undef(env([], Funcs), Vars, NEnv3),
  bind_params(NEnv3, NEnv, Pars, Params),
  eval_commands(Comms, NEnv, NEnv2),
  return_values(Nargout, Rets, Values, NEnv2).

bind_undef(Env, [], Env).
bind_undef(Env, [Var|Vars], NEnv) :-
  store_var(Env, Var, undef, Env2),
  bind_undef(Env2, Vars, NEnv).

bind_params(Vars, Vars, _, []).
bind_params(Vars, NVars, [Var|Pars], [Value|Values]) :-
store_var(Vars, Var, Value, Vars2),
bind_params(Vars2, NVars, Pars, Values).

return_values(0, _, [], _).
return_values(N, [Var|Rets], [V|Values], Env) :- M is N - 1,
   return_values(M, Rets, Values, Env), lookup_var(Var, Env, V).

eval_commands([], Env, Env).
eval_commands([C|Program], Env, NEnv) :-
  eval_comm(C, Env, N), eval_commands(Program, N, NEnv).

eval_comm(exp(E,false), Env, NEnv) :-
```

```
  eval_exp(E, NE, Env, NEnv), print_exp('ans', NE).
eval_comm(exp(E,true), Env, NEnv) :- eval_exp(E, _NE, Env, NEnv).

eval_comm(assign(var(V), E, false), Env, NEnv) :-
  eval_exp(E, NE, Env, N1), store_var(N1, V, NE, NEnv), print_exp(V, NE).
eval_comm(assign(var(V), E, true), Env, NEnv) :-
  eval_exp(E, NE, Env, N1), store_var(N1, V, NE, NEnv).
eval_comm(while(Exp, Commands), Env, NEnv) :-
  env_copy(Env, NEnv), eval_while(Exp, Commands, Env, NEnv).

eval_comm(for(Var, Range, Commands), Env, NEnv) :-
  eval_exp(Range, R, Env, N), eval_for(Var, R, Commands, N, NEnv).

eval_comm(if(Exp, Commands, _), Env, NEnv) :-
  eval_exp(Exp, N, Env, Env2), eval_if(N,Commands,Env,Env2,NEnv).

eval_if(N,Commands,_Env,Env2,NEnv) :- non_zero(N),
  eval_commands(Commands, Env2, NEnv).
eval_if(N,Commands,Env,_Env2,NEnv) :- is_zero(N),
  eval_commands(Commands, Env, NEnv).

eval_while(Exp, Commands, Env, NEnv) :-
  eval_exp(Exp, N, Env, Env2), non_zero(N),
  eval_commands(Commands, Env2, Env3),
  env_copy(Env3, NEnv), eval_while(Exp, Commands, Env3, NEnv).

eval_while(_, _, Env, Env).

eval_for(var(Var), matrix([[]]), _, Env, NEnv) :- store(Env, Var, [], NEnv).
eval_for(var(Var), R, Commands, Env, NEnv) :-
   get_elements(R, R1), eval_for2(Var, R1, Commands, Env, NEnv).

get_elements(const(X), [const(X)]).
get_elements(matrix([X]), X).

eval_for2(_, [], _, Env, Env).
eval_for2(Var, [R|Range], Commands, Env, NEnv) :-
  store_var(Env, Var, R, N),
  eval_commands(Commands, N, N2),
  eval_for2(Var, Range, Commands, N2, NEnv).

env_copy(env(V, F), env(NV, F)) :- var_copy(V, NV).

var_copy([], []).
var_copy([X/_|T], [X/_|T1]) :- var_copy(T,T1).

eval_exp(const(C), const(C), Env, Env).
eval_exp(var(Var), Value, Env, Env) :- lookup_var(Var, Env, Value).
eval_exp(bin_op(Op, Exp1, Exp2), Res, Env, NEnv) :-
  eval_exp(Exp1, R1, Env, E1), eval_exp(Exp2, R2, E1, NEnv),
```

```
  apply_op(Op, R1, R2, Res).

eval_exp(func_call(Func, Params), Value, Env, NEnv) :-
  eval_exps(Params, NParams, Env, NEnv),
  lookup_func(Func, Env, Function),
  get_funcs(Env, Funcs),
  eval_function(Function, NParams, 1, [Value], Funcs).

eval_exp(matrix(Rows), Result, Env, Env) :-
eval_rows(Rows, NewRows, Env), convert(NewRows, Result).

eval_exp(colon(S, E), Result, Env, Env) :- eval_exp(S, const(S1), Env, Env),
  eval_exp(E, const(E1), Env, Env), expand_colon(S1, 1, E1, Result).  /* , !.  */
eval_exp(colon(S, I, E), Result, Env, Env) :-
  eval_exp(S, const(S1), Env, Env), eval_exp(I, const(I1), Env, Env),
  eval_exp(E, const(E1), Env, Env),
  expand_colon(S1, I1, E1, Result).  /*  , ! .*/

expand_colon(_, 0, _, matrix([[]])).
expand_colon(S, I, E, matrix([P])) :- I > 0, arith_prog_incr(S, I, E, P).
expand_colon(S, I, E, matrix([P])) :- I < 0, arith_prog_decr(S, I, E, P).

arith_prog_incr(S, _, E, []) :- S > E.
arith_prog_incr(S, I, E, [const(S)|T]) :- S1 is S + I, arith_prog_incr(S1, I, E, T).

arith_prog_decr(S, _, E, []) :- S < E.
% I is negative so addition will result in decreasing sequence
arith_prog_decr(S, I, E, [const(S)|T]) :- S1 is S + I, arith_prog_decr(S1, I, E, T).

convert([[const(R)]], const(R)).
convert(Rows, matrix(Rows)).

eval_rows([], [], _).
eval_rows([R|Rows], [N|NRows], Env) :-
eval_exps(R, N, Env, _), eval_rows(Rows, NRows, Env).

get_funcs(env(_, Funcs), Funcs).

eval_exps([], [], E, E).
eval_exps([P|Pars], [NP|NPars], Env, NEnv) :-
  eval_exp(P, NP, Env, Env2), eval_exps(Pars, NPars, Env2, NEnv).

add(const(C1), const(C2), const(C3)) :- C3 is C1 + C2.
minus(const(C1), const(C2), const(C3)) :- C3 is C1 - C2.
mldivide(const(C1), const(C2), const(C3)) :- C3 is C2 / C1.
mrdivide(const(C1), const(C2), const(C3)) :- C3 is C1 / C2.
mtimes(const(C1), const(C2), const(C3)) :- C3 is C1 * C2.
gt(const(E1), const(E2), const(1)) :- E1 > E2.
gt(const(E1), const(E2), const(0)) :- E1 =< E2.
lt(const(E1), const(E2), const(1)) :- E1 < E2.
```

```
lt(const(E1), const(E2), const(0)) :- E1 >= E2.
ge(const(E1), const(E2), const(1)) :- E1 >= E2.
ge(const(E1), const(E2), const(0)) :- E1 < E2.
le(const(E1), const(E2), const(1)) :- E1 =< E2.
le(const(E1), const(E2), const(0)) :- E1 > E2.
eq(const(E1), const(E2), const(1)) :- E1 == E2.
eq(const(E1), const(E2), const(0)) :- E1 \== E2.
ne(const(E1), const(E2), const(1)) :- E1 \== E2.
ne(const(E1), const(E2), const(0)) :- E1 == E2.

is_zero(const(0)).
non_zero(const(N)) :- N \== 0.

apply_op('+', E1, E2, R) :- add(E1, E2, R).
apply_op('-', E1, E2, R) :- minus(E1, E2, R).
apply_op('/', E1, E2, R) :- mrdivide(E1, E2, R).
apply_op('\\', E1, E2, R) :- mldivide(E1, E2, R).
apply_op('*', E1, E2, R) :- mtimes(E1, E2, R).
apply_op('>', E1, E2, R) :- gt(E1, E2, R).
apply_op('<', E1, E2, R) :- lt(E1, E2, R).
apply_op('>=', E1, E2, R) :- ge(E1, E2, R).
apply_op('<=', E1, E2, R) :- le(E1, E2, R).
apply_op('==', E1, E2, R) :- eq(E1, E2, R).
apply_op('~=', E1, E2, R) :- ne(E1, E2, R).

print_exp(V, NE) :- write(V), write(' ='), nl, print_value(NE).

print_element(const(NE)) :- write('    '), write(NE).
print_value(const(NE)) :- write('    '), write(NE), nl,nl.
print_value(undef) :- write('    undefined!'), nl,nl.
print_value(matrix([[]])) :- write('    []'), nl.
print_value(matrix(NE)) :- print_rows(NE), nl.
print_rows([]).
print_rows([R|Rows]) :- print_row(R), nl, print_rows(Rows).
print_row([]).
print_row([C|Cols]) :- print_element(C), print_row(Cols).

store_var(env(Vars, Funcs), Key, Value, env(NVars, Funcs)) :-
store(Vars, Key, Value, NVars).

lookup_var(Key, env(Vars, _Funcs), Value) :- lookup(Key, Vars, Value).
lookup_func(Key, env(_, Funcs), Func) :- lookupf(Key, Funcs, Func).

func_matches(Key, function(Key, _, _, _)).
lookupf(Key, [F|_x], F) :- func_matches(Key, F).
lookupf(Key, [_|T], Value) :- lookupf(Key, T, Value).

store([], Key, Value, [Key/Value]).
store([Key/_Value2|T], Key, Value, [Key/Value|T]).
store([Key2/Value2|T], Key, Value, [Key2/Value2|BT]) :-
```

```
  Key\==Key2, store(T, Key, Value, BT).

lookup(Key, [Key/Value|_], Value).
lookup(Key, [Key2/_|T], Value) :- Key \== Key2, lookup(Key, T, Value).

run1(Pars) :- eval_matlab([
function('fact', ['y'], ['x'], ['x', 'y'],
     [assign(var('y'), const(1), true),
  while(bin_op('>', var('x'), const(0)),
    [assign(var('y'), bin_op('*', var('y'), var('x')), true),
 assign(var('x'), bin_op('-', var('x'), const(1)), true)])])],Pars, 1).

run12(Pars) :- eval_mfile("function y = fact(x)
y = 1;
while x > 0
  y = y * x;
  x = x - 1;
end", Pars, 1).

run2(Pars) :- eval_matlab([
function('fact', ['y'], ['x'], ['x', 'y'],
     [if(bin_op('>', var('x'), const(1)),
 [assign(var('y'), bin_op('*', var('x'),
   func_call('fact', [bin_op('-', var('x'), const(1))])), true)],
 [assign(var('y'), const(1), true)])])],
Pars, 1).

run22(Pars) :- eval_mfile("function y = fact(x)
if x > 1
  y = x * fact(x - 1);
else
  y = 1;
end", Pars, 1).

run3(Pars) :- eval_mfile("function y = fact(x)
y = 1;
for n = 1:x
  y = y * n;
end", Pars, 1).
```

C Comparing Different Prologs

Table 7. Comparing the runtime of different Prologs

	SWI/SICS		Ciao/SICS		SWI/Ciao	
Benchmark	Original	Spec	Original	Spec	Original	Spec
advisor	2.79	2.74	0.99	1.32	2.82	2.08
applast	3.60	5.63	1.00	1.50	3.60	3.75
contains.kmp	2.73	2.89	0.90	1.11	3.03	2.60
depth.lam	2.89	3.75	1.06	1.50	2.73	2.50
doubleapp	5.58	4.92	1.32	1.31	4.24	3.76
ex_depth	2.85	2.50	0.95	1.17	3.00	2.14
flip	5.00	3.50	1.44	1.25	3.46	2.80
groundunify.simple	2.81	3.83	1.03	2.17	2.72	1.77
groundunify.complex	2.92	2.60	1.14	1.30	2.57	2.00
imperative-solve	2.70	3.33	30.54	1.31	0.09	2.54
liftsolve.app	2.76	3.00	0.96	1.60	2.88	1.88
liftsolve.lmkng	2.67	2.93	0.94	1.07	2.82	2.73
map.reduce	2.79	4.75	16.40	1.13	0.17	4.22
map.rev	2.81	5.44	14.38	1.56	0.20	3.50
matchapp	2.49	4.55	0.97	1.36	2.55	3.33
match.kmp	3.04	4.65	1.08	1.47	2.81	3.16
model_elim	2.53	5.18	0.99	1.24	2.55	4.19
regexp.r1	3.22	3.88	0.96	1.24	3.35	3.12
regexp.r2	3.18	3.53	1.00	1.20	3.18	2.94
regexp.r3	3.23	3.45	0.99	1.13	3.26	3.06
relative	4.00	5.50	1.13	1.50	3.54	3.67
remove	2.98	3.38	1.26	1.28	2.37	2.65
remove2	3.45	4.05	1.17	1.35	2.94	3.00
rev	5.82	5.85	1.12	1.07	5.19	5.49
rev_acc_type	6.96	7.79	1.20	1.31	5.81	5.96
rotateprune	3.35	3.75	1.07	1.14	3.14	3.29
ssuply	3.26	3.60	1.03	1.40	3.17	2.57
transpose	4.49	2.73	1.25	1.27	3.58	2.14
vanilla.doubleapp	3.55	7.00	1.11	1.64	3.21	4.28
AVERAGE	3.46	4.16	3.08	1.34	2.93	3.14

Improved Compilation of Prolog to C Using Moded Types and Determinism Information*

J. Morales[1], M. Carro[1], and M. Hermenegildo[1,2]

[1] C. S. School, Technical U. of Madrid,
jfran@clip.dia.fi.upm.es and mcarro@fi.upm.es
[2] Depts. of Comp. Sci. and Elec. and Comp. Eng., U. of New Mexico (UNM)
herme@fi.upm.es and herme@unm.edu

Abstract. We describe the current status of and provide performance results for a prototype compiler of Prolog to C, `ciaocc`. `ciaocc` is novel in that it is designed to accept different kinds of high-level information, typically obtained via an automatic analysis of the initial Prolog program and expressed in a standardized language of assertions. This information is used to optimize the resulting C code, which is then processed by an off-the-shelf C compiler. The basic translation process essentially mimics the unfolding of a bytecode emulator with respect to the particular bytecode corresponding to the Prolog program. This is facilitated by a flexible design of the instructions and their lower-level components. This approach allows reusing a sizable amount of the machinery of the bytecode emulator: predicates already written in C, data definitions, memory management routines and areas, etc., as well as mixing emulated bytecode with native code in a relatively straightforward way. We report on the performance of programs compiled by the current version of the system, both with and without analysis information.
Keywords: Prolog, C, optimizing compilation, global analysis.

1 Introduction

Several techniques for implementing Prolog have been devised since the original interpreter developed by Colmerauer and Roussel [1], many of them aimed at achieving more speed. An excellent survey of a significant part of this work can be found in [2]. The following is a rough classification of implementation techniques for Prolog (which is, in fact, extensible to many other languages):

- Interpreters (such as C-Prolog [3] and others), where a slight preprocessing or translation might be done before program execution, but the bulk of the work is done at runtime by the interpreter.

* This work is partially supported by Spanish MCYT Project TIC 2002-0055 *CUBICO*, and EU Projects IST-2001-34717 *Amos* and IST-2001-38059 *ASAP*, and by the Prince of Asturias Chair in Information Science and Technology at the University of New Mexico. J. Morales is also supported by an MCYT fellowship co-financed by the European Social Fund.

- Compilers to *bytecode* and their interpreters (often called emulators), where the compiler produces relatively low level code in a special-purpose language. Most current emulators for Prolog are based on the Warren Abstract Machine (WAM) [4, 5], but other proposals exist [6, 7].
- Compilers to a lower-level language, often ("native") machine code, which require little or no additional support to be executed. One solution is for the compiler to generate machine code directly. Examples of this are Aquarius [8], versions of SICStus Prolog [9] for some architectures, BIM-Prolog [10], and Gnu Prolog [11]. Another alternative is to generate code in a (lower-level) language, such as, e.g., C-- [12] or C, for which compilers are readily available; the latter is the approach taken by `wamcc` [13].

Each solution has its advantages and disadvantages:

Executable performance vs. executable size and compilation speed: Compilation to lower-level code can achieve faster programs by eliminating interpretation overhead and performing lower-level optimizations. This difference gets larger as more sophisticated forms of code analysis are performed as part of the compilation process. Interpreters in turn have potentially smaller load/compilation times and are often a good solution due to their simplicity when speed is not a priority. Emulators occupy an intermediate point in complexity and cost. Highly optimized emulators [9, 14–17] offer very good performance and reduced program size which may be a crucial issue for very large programs and symbolic data sets.

Portability: Interpreters offer portability since executing the same Prolog code in different architectures boils down (in principle) to recompiling the interpreter. Emulators usually retain the portability of interpreters, by recompiling the emulator (bytecode is usually architecture-independent), unless they are written in machine code.[3] Compilers to native code require architecture-dependent back-ends which typically make porting and maintaining them a non-trivial task. Developing these back-ends can be simplified by using an intermediate RTL-level code [11], although different translations of this code are needed for different architectures.

Opportunities for optimizations: Code optimization can be applied at the Prolog level [18, 19], to WAM code [20], to lower-level code [21], and/or to native code [8, 22]. At a higher level it is typically possible to perform more global and structural optimizations, which are then implicitly carried over onto lower levels. Lower-level optimizations can be introduced as the native code level is approached; performing these low-level optimizations is one of the motivations for compiling to machine code. However, recent performance evaluations show that well-tuned emulators can beat, at least in some cases, Prolog compilers which generate machine code directly but which do not perform extensive optimization [11]. Translating to a low-level language such as C is interesting because it makes portability easier, as C compilers exist for most architectures and C

[3] This is the case for the Quintus emulator, although it is coded in a generic RTL language ("PROGOL") to simplify ports.

is low-level enough as to express a large class of optimizations which cannot be captured solely by means of Prolog-to-Prolog transformations.

Given all the considerations above, it is safe to say that different approaches are useful in different situations and perhaps even for different parts of the same program. The emulator approach can be very useful during development, and in any case for non-performance bound portions of large symbolic data sets and programs. On the other hand, in order to generate the highest performance code it seems appropriate to perform optimizations at all levels and to eventually translate to machine code. The selection of a language such as C as an intermediate target can offer a good compromise between opportunity for optimization, portability for native code, and interoperability in multi-language applications.

In `ciaocc` we have taken precisely such an approach: we implemented a compilation from Prolog to native code via an intermediate translation to C which optionally uses high-level information to generate optimized C code. Our starting point is the standard version of Ciao Prolog [17], essentially an emulator-based system of competitive performance. Its abstract machine is an evolution of the &-Prolog abstract machine [23], itself a separate branch from early versions (0.5–0.7) of the SICStus Prolog abstract machine.

`ciaocc` adopts the same scheme for memory areas, data tagging, etc. as the original emulator. This facilitates mixing emulated and native code (as done also by SICStus) and has also the important practical advantage that many complex and already existing fragments of C code present in the components of the emulator (builtins, low-level file and stream management, memory management and garbage collection routines, etc.) can be reused by the new compiler. This is important because our intention is not to develop a prototype but a full compiler that can be put into everyday use and developing all those parts again would be unrealistic.

A practical advantage is the availability of high-quality C compilers for most architectures. `ciaocc` differs from other systems which compile Prolog to C in that that the translation includes a scheme to optionally optimize the code using higher-level information available at compile-time regarding determinacy, types, instantiation modes, etc. of the source program.

Maintainability and portability lead us also not to adopt other approaches such as compiling to C--. The goal of C-- is to achieve portable high performance without relinquishing control over low-level details, which is of course very desirable. However, the associated tools do not seem to be presently mature enough as to be used for a compiler in production status within the near future, and not even to be used as base for a research prototype in their present stage. Future portability will also depend on the existence of back-ends for a range of architectures. We, however, are quite confident that the backend which now generates C code could be adapted to generate C-- (or other low-level languages) without too many problems.

The high-level information, which is assumed expressed by means of the powerful and well-defined assertion language of [24], is inferred by automatic global analysis tools. In our system we take advantage of the availability of

relatively mature tools for this purpose within the Ciao environment, and, in particular the preprocessor, CiaoPP [25]. Alternatively, such assertions can also be simply provided by the programmer.

Our approach is thus different from, for example, `wamcc`, which also generated C, but which did not use extensive analysis information and used low-level tricks which in practice tied it to a particular C compiler, `gcc`. Aquarius [8] and Parma [22] used analysis information at several compilation stages, but they generated directly machine code, and it has proved difficult to port and maintain them. Notwithstanding, they were landmark contributions that proved the power of using global information in a Prolog compiler.

A drawback of putting more burden on the compiler is that compile times and compiler complexity grow, specially in the global analysis phase. While this can turn out to be a problem in extreme cases, incremental analysis in combination with a suitable module system [26] can result in very reasonable analysis times in practice.[4] Moreover, global analysis is not mandatory in `ciaocc` and can be reserved for the phase of generating the final, "production" executable. We expect that, as the system matures, `ciaocc` itself (now in a prototype stage) will not be slower than a Prolog-to-bytecode compiler.

2 The Basic Compilation Scheme

The compilation process starts with a preprocessing phase which normalizes clauses (i.e., aliasing and structure unification is removed from the head), and expands disjunctions, negations and if-then-else constructs. It also unfolds calls to `is/2` when possible into calls to simpler arithmetic predicates, replaces the cut by calls to the lower-level predicates `metachoice/1` (which stores in its argument the address of the current choicepoint) and `metacut/1` (which performs a cut to the choicepoint whose address is passed in its argument), and performs a simple, local analysis which gathers information about the type and freeness state of variables.[5] Having this analysis in the compiler (in addition to the analyses performed by the preprocessor) improves the code even if no external information is available. The compiler then translates this normalized version of Prolog to WAM-based instructions (at this point the same ones used by the Ciao emulator), and then it splits these WAM instructions into an intermediate low level code and performs the final translation to C.

Typing WAM Instructions: WAM instructions dealing with data are handled internally using an enriched representation which encodes the possible instantiation state of their arguments.

[4] See [25] and its references for reports on analysis times of CiaoPP.

[5] In general, the types used throughout the paper are *instantiation types*, i.e., they have mode information built in (see [24] for a more complete discussion of this issue). *Freeness of variables* distinguishes between free variables and the *top* type, "term", which includes any term.

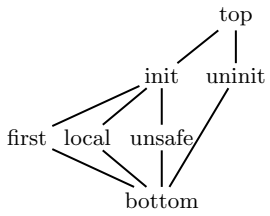

Fig. 1. Lattice of WAM types.

This allows using original type information, and also generating and propagating lower-level information regarding the type (i.e., from the point of view of the tags of the abstract machine) and instantiation/initialization state of the variables (which is not seen at a higher level). Unification instructions are represented as $\langle TypeX,\ X\rangle = \langle TypeY,\ Y\rangle$, where $TypeX$ and $TypeY$ refer to the classification of WAM-level types (see Figure 1), and X and Y refer to variables, which may be later stored as WAM X or Y registers or directly passed on as C function arguments. *init* and *uninit* correspond to initialized (i.e., free) and uninitialized variable cells. *First*, *local*, and *unsafe* classify the status of the variables according to where they appear in a clause.

Table 1 summarizes the aforementioned representation for some selected cases. The registers taken as arguments are the temporary registers *x(I)*, the stack variables *y(I)*, and the register for structure arguments *n(I)*. The last one can be seen as the second argument, implicit in the *unify_** WAM instructions. A number of other temporal registers are available, and used, for example, to hold intermediate results from expression evaluation. **_constant*, **_nil*, **_list* and **_structure* instructions are represented similarly. Only *x(·)* variables are created in an uninitialized state, and they are initialized on demand (in particular, when calling another predicate which may overwrite the registers and in the points where garbage collection can start). This representation is more uniform than the traditional WAM instructions, and as more information is known about the variables, the associated (low level) types can be refined and more specific code generated. Using a richer lattice and initial information (Section 3), a more descriptive intermediate code can be generated and used in the back-end.

put_variable(I,J)	⟨uninit,I⟩ = ⟨uninit,J⟩
put_value(I,J)	⟨init,I⟩ = ⟨uninit,J⟩
get_variable(I,J)	⟨uninit,I⟩ = ⟨init,J⟩
get_value(I,J)	⟨init,I⟩ = ⟨init,J⟩

unify_variable(I[, J])	`if` (initialized(J)) `then`
	⟨uninit,I⟩ = ⟨init,J⟩
	`else`
	⟨uninit,I⟩ = ⟨uninit,J⟩

unify_value(I[, J])	`if` (initialized(J)) `then`
	⟨init,I⟩ = ⟨init,J⟩
	`else`
	⟨init,I⟩ = ⟨uninit,J⟩

Table 1. Representation of some WAM unification instructions with types.

```
       while (code != NULL)
          code = ((Continuation (*)(State *))code)(state);
```

```
Continuation foo(State *state) {
   ...
   state->cont = &foo_cont;
   return &bar;
}
```

```
Continuation foo_cont(State *state) {
   ...
   return state->cont;
}
```

Fig. 2. The C execution loop and blocks scheme.

Generation of the Intermediate Low Level Language: WAM-like control and data instructions (Table 2) are then split into simpler ones (Table 3) (of a level similar to that of the BAM [27]) which are more suitable for optimizations, and which simplify the final code generation. The *Type* argument in the unification instructions reflects the type of the their arguments: for example, in the instruction *bind*, *Type* is used to specify if the arguments contain a variable or not. For the unification of structures, write and read modes are avoided by using a two-stream scheme [2] which is implicit in the unification instructions in Table 1 and later translated into the required series of assignments and jump instructions (*jump*, *cjump*) in Table 2. The WAM instructions *switch_on_term*, *switch_on_cons* and *switch_on_functor* are also included, although the C back-end does not exploit them fully at the moment, resorting to a linear search in some cases. A more efficient indexing mechanism will be implemented in the near future.

Builtins return an exit state which is used to decide whether to backtrack or not. Determinism information, if available, is passed on through this stage and used when compiling with optimizations (see Section 3).

Compilation to C: The final C code conceptually corresponds to an unfolding of the emulator loop with respect to the particular sequence(s) of WAM instructions corresponding to the Prolog program. Each basic block of bytecode (i.e., each sequence beginning in a label and ending in an instruction involving a possibly non-local jump) is translated to a separate C function, which receives (a pointer to) the state of the abstract machine as input argument, and returns a pointer to the continuation. This approach, chosen on purpose, does not build functions which are too large for the C compiler to handle. For example, the code corresponding to a head unification is a basic block, since it is guaranteed that the labels corresponding to the two-stream algorithm will have local scope. A failure during unification is implemented by (conditionally) jumping to a special label, *fail*, which actually implements an exit protocol similar to that generated by the general C translation. Figure 2 shows schematic versions of the execution loop and templates of the functions that code blocks are compiled into.

This scheme does not require machine-dependent options of the C compiler or extensions to ANSI C. One of the goals of our system –to study the impact of optimizations based on high-level information on the program– can be achieved with the proposed compilation scheme, and, as mentioned before, we give portability and code cleanliness a high priority. The option of producing more efficient but non-portable code can always be added at a later stage.

Choice, stack and heap management instructions	
no_choice	Mark that there is no alternative
push_choice(Arity)	Create a choicepoint
recover_choice(Arity)	Restore the state stored in a choicepoint
last_choice(Arity)	Restore state and discard latest choice point
complete_choice(Arity)	Complete the choice point
cut_choice(Chp)	Cut to a given choice point
push_frame	Allocate a frame on top of the stack
complete_frame(FrameSize)	Complete the stack frame
modify_frame(NewSize)	Change the size of the frame
pop_frame	Deallocate the last frame
recover_frame	Recover after returning from a call
ensure_heap(Amount, Arity)	Ensure that enough heap is allocated.
Unification	
load(X, Type)	Load X with a term
trail_if_conditional(A)	Trail if A is a conditional variable
bind(TypeX, X, TypeY, Y)	Bind X and Y
read(Type, X)	Begin read of the structure arguments of X
deref(X, Y)	Dereference X into Y
move(X, Y)	Copy X to Y
globalize_if_unsafe(X, Y)	Copy (safely) X to stack variable Y
globalize_to_arg(X, Y)	Copy (safely) X to structure argument Y
jump(Label)	Jump to *Label*
cjump(Cond, Label)	Jump to *Label* if *Cond* is true
not(Cond)	Negate the *Cond* condition
test(Type, X)	True if X matches *Type*
equal(X, Y)	True if X and Y are equal
Indexing	
switch_on_type(X, Var, Str, List, Cons)	Jump to the label that matches the type of X
switch_on_functor(X, Table, Else)	
switch_on_cons(X, Table, Else)	

Table 2. Control and data instructions.

An Example — the `fact/2` *Predicate:* We will illustrate briefly the different compilation stages using the well-known factorial program (Figure 3). We have chosen it due to its simplicity, even if the performance gain is not very high in this case. The normalized code is shown in Figure 4, and the WAM code corresponding to the recursive clause is listed in the leftmost column of Table 3, while the internal representation of this code appears in the middle column of the same table. Variables are annotated using information which can be deduced from local clause inspection.

This WAM-like representation is translated to the low-level code as shown in Figure 5 (ignore, for the moment, the framed instructions; they will be discussed in Section 3). This code is what is finally translated to C.

For reference, executing `fact(100, N)` 20000 times took 0.65 seconds running emulated bytecode, and 0.63 seconds running the code compiled to C (a speedup of 1.03). This did not use external information, used the emulator data

structures to store Prolog terms, and performed runtime checks to verify that the arguments are of the right type, even when this is not strictly necessary. Since the loop in Figure 2 is a bit more costly (by a few assembler instructions) than the WAM emulator loop, the speedup brought about by the C translation alone is, in many cases, not as relevant as one may think at first.

```
fact(0, 1).
fact(X, Y) :-
    X > 0,
    X0 is X - 1,
    fact(X0, Y0),
    Y is X * Y0.
```

Fig. 3. Factorial, initial code.

```
fact(A, B) :-
    0 = A,
    1 = B.
```

```
fact(A, B) :-
    A > 0,
    builtin__sub1_1(A, C),
    fact(C, D),
    builtin__times_2(A, D, B).
```

Fig. 4. Factorial, after normalizing.

WAM code	Without Types/Modes	With Types/Modes
put_constant(0,2)	0 = ⟨uninit,x(2)⟩	0 = ⟨uninit,x(2)⟩
builtin_2(37,0,2)	⟨init,x(0)⟩ > ⟨int(0),x(2)⟩	⟨int,x(0)⟩ > ⟨int(0),x(2)⟩
allocate	builtin__push_frame	builtin__push_frame
get_y_variable(0,1)	⟨uninit,y(0)⟩ = ⟨init,x(1)⟩	⟨uninit,y(0)⟩ = ⟨var,x(1)⟩
get_y_variable(2,0)	⟨uninit,y(2)⟩ = ⟨init,x(0)⟩	⟨uninit,y(2)⟩ = ⟨int,x(0)⟩
init([1])	⟨uninit,y(1)⟩ = ⟨uninit,y(1)⟩	⟨uninit,y(1)⟩ = ⟨uninit,y(1)⟩
true(3)	builtin__complete_frame(3)	builtin__complete_frame(3)
function_1(2,0,0)	builtin__sub1_1(⟨init,x(0)⟩, ⟨uninit,x(0)⟩)	builtin__sub1_1(⟨int,x(0)⟩, ⟨uninit,x(0)⟩)
put_y_value(1,1)	⟨init,y(1)⟩ = ⟨uninit,x(1)⟩	⟨var,y(1)⟩ = ⟨uninit,x(1)⟩
call(fac/2,3)	builtin__modify_frame(3) fact(⟨init,x(0)⟩, ⟨init,x(1)⟩)	builtin__modify_frame(3) fact(⟨init,x(0)⟩, ⟨var,x(1)⟩)
put_y_value(2,0)	⟨init,y(2)⟩ = ⟨uninit,x(0)⟩	⟨int,y(2)⟩ = ⟨uninit,x(0)⟩
put_y_value(2,1)	⟨init,y(1)⟩ = ⟨uninit,x(1)⟩	⟨number,y(1)⟩ = ⟨uninit,x(1)⟩
function_2(9,0,0,1)	builtin__times_2(⟨init,x(0)⟩, ⟨init,x(1)⟩,⟨uninit,x(0)⟩)	builtin__times_2(⟨int,x(0)⟩, ⟨number,x(1)⟩, ⟨uninit,x(0)⟩)
get_y_value(0,0)	⟨init,y(0)⟩ = ⟨init,x(0)⟩	⟨var,y(0)⟩ = ⟨init,x(0)⟩
deallocate	builtin__pop_frame	builtin__pop_frame
execute(true/0)	builtin__proceed	builtin__proceed

Table 3. WAM code and internal representation without and with external types information. Underlined instruction changed due to additional information.

3 Improving Code Generation

In order to improve the generated code using global information, the compiler can take into account types, modes, determinism and non-failure properties [25] coded as assertions [24] — a few such assertions can be seen in the example which appears later in this section. Automatization of the compilation process is achieved by using the CiaoPP analysis tool in connection with `ciaocc`. CiaoPP implements several powerful analysis (for modes, types, and determinacy, besides other relevant properties) which are able to generate (or check) these assertions. The program information that CiaoPP is currently able to infer automatically is actually enough for our purposes (with the single exception stated in Section 4).

```
fact(x(0), x(1)) :-
  push_choice(2)
  ensure_heap(callpad,2)
  deref(x(0),x(0))
   cjump(not(test(var,x(0))),V3)
   load(temp2,int(0))
   bind(var,x(0),nonvar,temp2)
   jump(V4)
 V3:
  cjump(not(test(int(0),x(0))),fail)
 V4:
  deref(x(1),x(1))
   cjump(not(test(var,x(1))),V5)
  load(temp2,int(1))
  bind(var,x(1),nonvar,temp2)
   jump(V6)
 V5:
   cjump(not(test(int(1),x(1))),fail)
 V6:
  complete_choice(2)
;
```

```
last_choice(2)
load(x(2),int(0))
>(x(0),x(2))
push_frame
move(x(1),y(0))
move(x(0),y(2))
init(y(1))
complete_frame(3)
builtin__sub1(x(0), x(0))
move(y(1),x(1))
modify_frame(3)
fact(x(0), x(1))
recover_frame
move(y(2),x(0))
move(y(1),x(1))
builtin__times(x(0), x(1), x(0))
deref(y(0),temp)
deref(x(0),x(0))
=(temp,x(0))
pop_frame
```

Fig. 5. Low level code for the `fact/2` example (see also Section 3).

The generation of low-level code using additional type information makes use of a lattice of moded types obtained by extending the *init* element in the lattice in Figure 1 with the type domain in Figure 6. `str(N/A)` corresponds to (and expands to) each of the structures whose name and arity are known at compile time. This information enriches the *Type* parameter of the low-level code. Information about the determinacy / number of solutions of each call is carried over into this stage and used to optimize the C code.

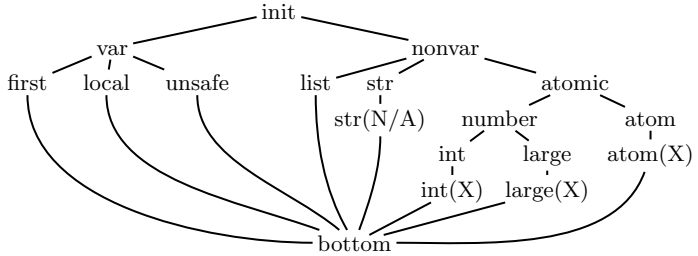

Fig. 6. Extended *init* subdomain.

In general, information about types and determinism makes it possible to avoid some runtime tests. The standard WAM compilation also performs some optimizations (e.g., classification of variables and indexing on the first argument), but they are based on a per-clause (per-predicate, in the case of indexing) analysis, and in general it does not propagate the deduced information (e.g. from

arithmetic builtins). A number of further optimizations can be done by using type, mode, and determinism information:

Unify Instructions: Calls to the general *unify* builtin are replaced by the more specialized *bind* instruction if one or both arguments are known to store variables. When arguments are known to be constants, a simple comparison instruction is emitted instead.

Two-Stream Unification: Unifying a register with a structure/constant requires some tests to determine the unification mode (read or write). An additional test is required to compare the register value with the structure/constant. These tests can often be removed at compile-time if enough information is known about the variable.

Indexing: Index trees are generated by selecting literals (mostly builtins and unifications), which give type/mode information, to construct a decision tree on the types of the first argument.[6] When type information is available, the search can be optimized by removing some of the tests in the nodes.

Avoiding Unnecessary Variable Safety Tests: Another optimization performed in the low level code using type information is the replacement of globalizing instructions for unsafe variables by explicit dereferences. When the type of a variable is nonvar, its globalization is equivalent to a dereference, which is faster.

Uninitialized Output Arguments: When possible, letting the called predicate fill in the contents of output arguments in pre-established registers avoids allocation, initialization, and binding of free variables, which is slower.

Selecting Optimized Predicate Versions: Calls to predicates can also be optimized in the presence of type information. Specialized predicate versions (in the sense of low level optimizations) can be generated and selected using call patterns deduced from the type information. The current implementation does not generate specialized versions of user predicates, since this can already be done extensively by CiaoPP [18]. However it does optimize calls to internal *builtin* predicates written in C (such as, e.g., arithmetic builtins), which results in relevant speedups in many cases.

Determinism: These optimizations are based on two types of analysis. The first one uses information regarding the number of solutions for a predicate call to deduce, for each such call, if there is a known and fixed fail continuation. Then, instructions to manage choicepoints are inserted. The resulting code is then re-analyzed to remove these instructions when possible or to replace them by simpler ones (e.g., to restore a choice point state without untrailing, if it is known at compile time that the execution will not trail any value since the choice point was created). The latter can take advantage of additional information regarding register, heap, and trail usage of each predicate.[7] In addition, the C back-end can

[6] This is the WAM definition, which can of course be extended to other arguments.

[7] This is currently known only for internal predicates written in C, and which are available by default in the system, but the scheme is general and can be extended to Prolog predicates.

generate different argument passing schemes based on determinism information: predicates with zero or one solution can be translated to a function returning a boolean, and predicates with exactly one solution to a function returning `void`. This requires a somewhat different translation to C (which we do not have space to describe in full) and which takes into account this possibility by bypassing the emulator loop, in several senses similarly to what is presented in [28].

An Example — the `fact/2` Predicate with program information: Let us assume that it has been inferred that `fact/2` (Figure 3) is always called with its first argument instantiated to an integer and with a free variable in its second argument. This information is written in the assertion language for example as:[8]

```
:- true pred fact(X, Y) :  int * var => int * int.
```

which reflects the types and modes of the calls and successes of the predicate. That information is also propagated through the normalized predicate producing the annotated program shown in Figure 7, where program-point information is also shown.

```
fact(A, B) :-
    true(int(A)),
    0 = A,
    true(var(B)),
    1 = B.
```

```
fact(A, B) :-
    true(int(A)),
    A > 0,
    true(int(A)), true(var(C)),
    builtin__sub1_1(A, C),
    true(any(C)), true(var(D)),
    fact(C, D),
    true(int(A)), true(int(D)),
    true(var(B)),
    builtin__times_2(A, D, B).
```

Fig. 7. Annotated factorial (using type information).

The WAM code generated for this example is shown in the rightmost column of Table 3. Underlined instructions were made more specific due to improved information — but note that the representation is homogeneous with respect to the "no information" case. The impact of type information in the generation of low-level code can be seen in Figure 5. Instructions inside the dashed boxes are removed when type information is available, and the (arithmetic) builtins enclosed in rectangles are replaced by calls to specialized versions which work with integers and which do not perform type/mode testing. The optimized `fact/2` program took 0.54 seconds with the same call as in Section 2: a 20% speedup with respect to the bytecode version and a 16% speedup over the compilation to C without type information.

[8] The `true` prefix implies that this information is to be *trusted and used*, rather than to be *checked* by the compiler. Indeed, we require the stated properties to be correct, and `ciaocc` does not check them: this is a task delegated to CiaoPP. Wrong *true* assertions can, therefore, lead to incorrect compilation. However, the assertions generated by CiaoPP are guaranteed correct by the analysis process.

Program	Bytecode (Std. Ciao)	Non opt. C	Opt1. C	Opt2. C
`queens11 (1)`	691	391 (1.76)	208 (3.32)	166 (4.16)
`crypt (1000)`	1525	976 (1.56)	598 (2.55)	597 (2.55)
`primes (10000)`	896	697 (1.28)	403 (2.22)	402 (2.22)
`tak (1000)`	9836	5625 (1.74)	5285 (1.86)	771 (12.75)
`deriv (10000)`	125	83 (1.50)	82 (1.52)	72 (1.74)
`poly (100)`	439	251 (1.74)	199 (2.20)	177 (2.48)
`qsort (10000)`	521	319 (1.63)	378 (1.37)	259 (2.01)
`exp (10)`	494	508 (0.97)	469 (1.05)	459 (1.07)
`fib (1000)`	263	245 (1.07)	234 (1.12)	250 (1.05)
`knights (1)`	621	441 (1.46)	390 (1.59)	356 (1.74)
Average Speedup		(1.46 – 1.43)	(1.88 – 1.77)	(3.18 – 2.34)

Table 4. Bytecode emulation vs. unoptimized, optimized (types), and optimized (types and determinism) compilation to C. *Arithmetic – Geometric* means are shown.

4 Performance Measurements

We have evaluated the performance of a set of benchmarks executed by emulated bytecode, translation to C, and by other programming systems. The benchmarks, while representing interesting cases, are not real-life programs, and some of them have been executed up to 10.000 times in order to obtain reasonable and stable execution times. Since parts of the compiler are still in an experimental state, we have not been able to use larger benchmarks yet. All the measurements have been performed on a Pentium 4 Xeon @ 2.0GHz with 1Gb of RAM, running Linux with a 2.4 kernel and using `gcc` 3.2 as C compiler. A short description of the benchmarks follows:

crypt: Cryptoarithmetic puzzle involving multiplication.
primes: Sieve of Erathostenes (with N = 98).
tak: Takeuchi function with arguments `tak(18, 12, 6, X)`.
deriv: Symbolic derivation of polynomials.
poly: Symbolically raise `1+x+y+z` to the 10^{th} power.
qsort: QuickSort of a list of 50 elements.
exp: 13^{7111} using both a linear- and a logarithmic-time algorithm.
fib: F_{1000} using a simply recursive predicate.
knight: Chess knight tour in a 5×5 board.

A summary of the results appears in Table 4. The figures between parentheses in the first column is the number of repetitions of each benchmark. The second column contains the execution times of programs run by the Ciao bytecode emulator. The third column corresponds to programs compiled to C without compile-time information. The fourth and fifth columns correspond, respectively, to the execution times when compiling to C with type and type+determinism information. The numbers between parentheses are the speedups relative to the bytecode version. All times are in milliseconds. Arithmetic and geometric means are also shown in order to diminish the influence of exceptional cases.

Table 5 shows the execution times for the same benchmarks in five well-known Prolog compilers: GNU Prolog 1.2.16, `wamcc` 2.23, SICStus 3.8.6, SWI-Prolog 5.2.7, and Yap 4.5.0. The aim is not really to compare directly with them,

Program	GProlog	WAMCC	SICStus	SWI	Yap	Mercury	Opt2. C / Mercury
queens11 (1)	809	378	572	5869	362	106	1.57
crypt (1000)	1258	966	1517	8740	1252	160	3.73
primes (10000)	1102	730	797	7259	1233	336	1.20
tak (1000)	11955	7362	6869	74750	8135	482	1.60
deriv (10000)	108	126	121	339	100	72	1.00
poly (100)	440	448	420	1999	424	84	2.11
qsort (10000)	618	522	523	2619	354	129	2.01
exp (10)	—	—	415	—	340	—	—
fib (1000)	—	—	285	—	454	—	—
knights (1)	911	545	631	2800	596	135	2.63
						Average	1.98 – 1.82

Table 5. Speed of other Prolog systems and Mercury

because a different underlying technology and external information is being used, but rather to establish that our baseline, the speed of the bytecode system (Ciao), is similar and quite close, in particular, to that of SICStus. In principle, comparable optimizations could be made in these systems. The cells marked with "—" correspond to cases where the benchmark could not be executed (in GNU Prolog, `wamcc`, and SWI, due to lack of multi-precision arithmetic).

We also include the performance results for Mercury [29] (version 0.11.0). Strictly speaking the Mercury compiler is not a Prolog compiler: the source language is substantially different from Prolog. But Mercury has enough similarities to be relevant and its performance represents an upper reference line, given that the language was restricted in several ways to allow the compiler, which generates C code with different degrees of "purity", to achieve very high performance by using extensive optimizations. Also, the language design requires the necessary information to perform these optimizations to be included by the programmer as part of the source. Instead, the approach that we use in Ciao is to infer automatically the information and not restricting the language.

Going back to Table 4, while some performance gains are obtained in the *naive* translation to C, these are not very significant, and there is even one program which shows a slowdown. We have tracked this down to be due to a combination of several factors:

- The simple compilation scheme generates clean, portable, "trick-free" C (some compiler dependent extensions would speed up the programs). The execution profile is very near to what the emulator would do.
- As noted in Section 2, the C compiler makes the fetch/switch loop of the emulator a bit cheaper than the C execution loop. We have identified this as a cause of the poor speedup of programs where recursive calls dominate the execution (e.g., `factorial`). We want, of course, to improve this point in the future.
- The increment in size of the program (to be discussed later — see Table 6) may also cause more cache misses. We also want to investigate this point in more detail.

As expected, the performance obtained when using compile-time information is much better. The best speedups are obtained in benchmarks using arithmetic builtins, for which the compiler can use optimized versions where several checks have been removed. In some of these cases the functions which implement arithmetic operations are simple enough as to be inlined by the C compiler — an added benefit which comes for free from compiling to an intermediate language (C, in this case) and using tools designed for it. This is, for example, the case of `queens`, in which it is known that all the numbers involved are integers. Besides the information deduced by the analyzer, hand-written annotations stating that the integers involved fit into a machine word, and thus there is no need for infinite precision arithmetic, have been manually added.[9]

Determinism information often (but not always) improves the execution. The Takeuchi function (`tak`) is an extreme case, where savings in choicepoint generation affect execution time. While the performance obtained is still almost a factor of 2 from that of Mercury, the results are encouraging since we are dealing with a more complex source language (which preserves full unification, logical variables, cuts, `call`/1, database, etc.), we are using a portable approach (compilation to standard C), and we have not yet applied all possible optimizations.

A relevant point is to what extent a sophisticated analysis tool is useful in practical situations. The degree of optimization chosen can increase the time spent in the compilation, and this might preclude its everyday use. We have measured (informally) the speed of our tools in comparison with the standard Ciao Prolog compiler (which generates bytecode), and found that the compilation to C takes about three times more than the compilation to bytecode. A considerable amount of time is used in I/O, which is being performed directly from Prolog, and which can be optimized if necessary. Due to a well-developed machinery (which can notwithstanding be improved in a future by, e.g, compiling CiaoPP itself to C), the global analysis necessary for examples is really fast and never exceeded twice the time of the compilation to C. Thus we think that the use of global analysis to obtain the information we need for `ciaocc` is a practical option already in its current state.

Table 6 compares object size (in bytes) of the bytecode and the different schemes of compilation to C and using the same compiler options in all cases. While modern computers usually have a large amount of memory, and program size hardly matters for a single application, users stress computers more and more by having several applications running simultaneously. On the other hand, program size does impact their startup time, important for small, often-used commands. Besides, size is still very important when addressing small devices with limited resources.

As mentioned in Section 1, due to the different granularity of instructions, larger object files and executables are expected when compiling to C. The ratio depends heavily on the program and the optimizations applied. Size increase

[9] This is the only piece of information used in our benchmarks that cannot be currently determined by CiaoPP. It should be noted, though, that the absence of this annotation would only make the final executable less optimized, but never incorrect.

Program	Bytecode	Non opt. C	Opt1. C	Opt2. C
queens11	7167	36096 (5.03)	29428 (4.10)	42824 (5.97)
crypt	12205	186700 (15.30)	107384 (8.80)	161256 (13.21)
primes	6428	50628 (7.87)	19336 (3.00)	31208 (4.85)
tak	5445	18928 (3.47)	18700 (3.43)	25476 (4.67)
deriv	9606	46900 (4.88)	46644 (4.85)	97888 (10.19)
poly	13541	163236 (12.05)	112704 (8.32)	344604 (25.44)
qsort	6982	90796 (13.00)	67060 (9.60)	76560 (10.96)
exp	6463	28668 (4.43)	28284 (4.37)	25560 (3.95)
fib	5281	15004 (2.84)	14824 (2.80)	18016 (3.41)
knights	7811	39496 (5.05)	39016 (4.99)	39260 (5.03)
Average Increase		(7.39 – 6.32)	(5.43 – 4.94)	(8.77 – 7.14)

Table 6. Compared size of object files (bytecode vs. C) including *Arithmetic* - *Geometric* means.

with respect to the bytecode can be as large as 15× when translating to C without optimizations, and the average case sits around a 7-fold increase. This increment is partially due to repeated code in the indexing mechanism, which we plan to improve in the future.[10] Note that, as our framework can mix bytecode and native code, it is possible to use both in order to achieve more speed in critical parts, and to save program space otherwise. Heuristics and translation schemes like those described in [30] can hence be applied (and implemented as a source to source transformation).

The size of the object code produced by `wamcc` is roughly comparable to that generated by `ciaocc`, although `wamcc` produces smaller intermediate object code files. However the final executable / process size depends also on which libraries are linked statically and/or dynamically. The Mercury system is somewhat incomparable in this regard: it certainly produces relatively small component files but then relatively large final executables (over 1.5 MByte).

Size, in general, decreases when using type information, as many runtime type tests are removed, the average size being around five times the bytecode size. Adding determinism information increases the code size because of the additional inlining performed by the C compiler and the more complex parameter passing code. Inlining was left to the C compiler; experiments show that more aggressive inlining does not necessarily result in better speedups.

It is interesting to note that some optimizations used in the compilation to C would not give comparable results when applied directly to a bytecode emulator. For example, a version of the bytecode emulator hand-coded to work with small integers (which can be boxed into a tagged word) performed worse than that obtained doing the same with compilation to C. That suggests that when the overhead of calling builtins is reduced, as is the case in the compilation to C, some optimizations which only produce minor improvements for emulated systems acquire greater importance.

[10] In all cases, the size of the bytecode emulator / runtime support (around 300Kb) has to be added, although not all the functionality it provides is always needed.

5 Conclusions and Future Work

We have reported on the scheme and performance of `ciaocc`, a Prolog-to-C compiler which uses type analysis and determinacy information to improve code generation by removing type and mode checks and by making calls to specialized versions of some builtins. We have also provided performance results. `ciaocc` is still in a prototype stage, but it already shows promising results.

The compilation uses internally a simplified and more homogeneous representation for WAM code, which is then translated to a lower-level intermediate code, using the type and determinacy information inferred by CiaoPP. This code is finally translated into C by the compiler back-end. The intermediate code makes the final translation step easier and will facilitate developing new back-ends for other target languages.

We have found that optimizing a WAM bytecode emulator is more difficult and results in lower speedups, due to the larger granularity of the bytecode instructions. The same result has been reported elsewhere [2], although some recent work tries to improve WAM code by means of local analysis [20].

We expect to also be able to use the information inferred by CiaoPP (e.g., determinacy) to improve clause selection and to generate a better indexing scheme at the C level by using hashing on constants, instead of the linear search used currently. We also want to study which other optimizations can be added to the generation of C code without breaking its portability, and how the intermediate representation can be used to generate code for other back-ends (for example, GCC RTL, CIL, Java bytecode, etc.).

References

1. Colmerauer, A.: The Birth of Prolog. In: Second History of Programming Languages Conference. ACM SIGPLAN Notices (1993) 37–52
2. Van Roy, P.: 1983-1993: The Wonder Years of Sequential Prolog Implementation. Journal of Logic Programming **19/20** (1994) 385–441
3. Pereira, F.: C-Prolog User's Manual, Version 1.5, University of Edinburgh. (1987)
4. Warren, D.: An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025 (1983)
5. Ait-Kaci, H.: Warren's Abstract Machine, A Tutorial Reconstruction. MIT Press (1991)
6. Taylor, A.: High-Performance Prolog Implementation. PhD thesis, Basser Department of Computer Science, Unversity of Sidney (1991)
7. Krall, A., Berger, T.: The VAM_{AI} - an abstract machine for incremental global dataflow analysis of Prolog. In de la Banda, M.G., Janssens, G., Stuckey, P., eds.: ICLP'95 Post-Conference Workshop on Abstract Interpretation of Logic Languages, Tokyo, Science University of Tokyo (1995) 80–91
8. Van Roy, P., Despain, A.: High-Performace Logic Programming with the Aquarius Prolog Compiler. IEEE Computer Magazine (1992) 54–68
9. Swedish Institute for Computer Science PO Box 1263, S-164 28 Kista, Sweden: SICStus Prolog 3.8 User's Manual. 3.8 edn. (1999) Available from `http://www.sics.se/sicstus/`.

10. Mariën, A.: Improving the Compilation of Prolog in the Framework of the Warren Abstract Machine. PhD thesis, Katholieke Universiteit Leuven (1993)
11. Diaz, D., Codognet, P.: Design and Implementation of the GNU Prolog System. Journal of Functional and Logic Programming **2001** (2001)
12. Jones, S.L.P., Ramsey, N., Reig, F.: C--: A Portable Assembly Language that Supports Garbage Collection. In Nadathur, G., ed.: International Conference on Principles and Practice of Declarative Programming. Number 1702 in Lecture Notes in Computer Science, Springer Verlag (1999) 1–28
13. Codognet, P., Diaz, D.: WAMCC: Compiling Prolog to C. In Sterling, L., ed.: International Conference on Logic Programming, MIT PRess (1995) 317–331
14. Quintus Computer Systems Inc. Mountain View CA 94041: Quintus Prolog User's Guide and Reference Manual—Version 6. (1986)
15. Santos-Costa, V., Damas, L., Reis, R., Azevedo, R.: The Yap Prolog User's Manual. (2000) Available from `http://www.ncc.up.pt/~vsc/Yap`.
16. Demoen, B., Nguyen, P.L.: So Many WAM Variations, So Little Time. In: Computational Logic 2000, Springer Verlag (2000) 1240–1254
17. Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-García, P., Puebla, G.: The Ciao Prolog System. Reference Manual (v1.8). The Ciao System Documentation Series–TR CLIP4/2002.1, School of Computer Science, Technical University of Madrid (UPM) (2002) System and on-line version of the manual available at `http://clip.dia.fi.upm.es/Software/Ciao/`.
18. Puebla, G., Hermenegildo, M.: Abstract Specialization and its Applications. In: ACM Partial Evaluation and Semantics based Program Manipulation (PEPM'03), ACM Press (2003) 29–43 Invited talk.
19. Winsborough, W.: Multiple Specialization using Minimal-Function Graph Semantics. Journal of Logic Programming **13** (1992) 259–290
20. Ferreira, M., Damas, L.: Multiple Specialization of WAM Code. In: Practical Aspects of Declarative Languages. Number 1551 in LNCS, Springer (1999)
21. Mills, J.: A high-performance low risc machine for logic programming. Journal of Logic Programming (6) (1989) 179–212
22. Taylor, A.: LIPS on a MIPS: Results from a prolog compiler for a RISC. In: 1990 International Conference on Logic Programming, MIT Press (1990) 174–189
23. Hermenegildo, M., Greene, K.: The &-Prolog System: Exploiting Independent And-Parallelism. New Generation Computing **9** (1991) 233–257
24. Puebla, G., Bueno, F., Hermenegildo, M.: An Assertion Language for Constraint Logic Programs. In Deransart, P., Hermenegildo, M., Maluszynski, J., eds.: Analysis and Visualization Tools for Constraint Programming. Number 1870 in LNCS. Springer-Verlag (2000) 23–61
25. Hermenegildo, M., Puebla, G., Bueno, F., López-García, P.: Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In: 10th International Static Analysis Symposium (SAS'03). Number 2694 in LNCS, Springer-Verlag (2003) 127–152
26. Cabeza, D., Hermenegildo, M.: A New Module System for Prolog. In: International Conference on Computational Logic, CL2000. Number 1861 in LNAI, Springer-Verlag (2000) 131–148
27. Van Roy, P.: Can Logic Programming Execute as Fast as Imperative Programming? PhD thesis, Univ. of California Berkeley (1990) Report No. UCB/CSD 90/600.
28. Henderson, F., Somogyi, Z.: Compiling Mercury to High-Level C Code. In Nigel Horspool, R., ed.: Proceedings of Compiler Construction 2002. Volume 2304 of LNCS., Springer-Verlag (2002) 197–212

29. Somogyi, Z., Henderson, F., Conway, T.: The execution algorithm of Mercury: an efficient purely declarative logic programming language. JLP **29** (1996)
30. Tarau, P., De Bosschere, K., Demoen, B.: Partial Translation: Towards a Portable and Efficient Prolog Implementation Technology. Journal of Logic Programming **29** (1996) 65–83

Generation of Stripped-down Runtime Systems using Abstract Interpretation and Program Specialization

Germán Puebla[1], Manuel Hermenegildo[1,2], and Jesús Correas[3]

[1] Department of Computer Science
Technical University of Madrid (UPM)
{german,herme}@fi.upm.es
[2] Depts. of Computer Science and Electrical and Computer Engineering
University of New Mexico (UNM)
[3] SIP, Complutense University of Madrid (UCM)
jcorreas@fdi.ucm.es

1 Introduction and Motivation

Libraries and modules are a fundamental tool for developing large applications, as they allow sharing common code between different programs and they provide a clean interface to widely used routines. Many development environments based on bytecode virtual machine emulators often provide a full-featured library with large amounts of code (as for example the Java run-time environment). Such systems are composed of two different environments: on one hand, a software development kit for program development, comprising a compiler and set of libraries, and on the other hand a runtime system, which contains a virtual machine interpreter (and/or a just-in-time compiler) and a bytecode version of the libraries. Such systems present a lot of advantages for the programmer: interoperability (to some extent in pervasive devices), a generic and independent programming interface, etc. However, these runtime systems tend to use an excessive amount of space in both memory and permanent storage, as their libraries are programmed for a very general usage, covering lots of possible cases. In addition, programmers tend to develop libraries that are more general purpose than is actually needed for a specific application, in order to make the library as reusable as possible. This approach, useful for program development, results however in applications which include significant amounts of useless code fragments.

In a pervasive system scenario, the space needed by a program is of vital importance in this kind of devices, and the use of general development tools and libraries is usually very restricted. In the case of Java, the alternative for developing software for small devices is to use a different development kit and runtime system, the Java Micro Edition.[4] It contains several runtime systems and development kits depending on how powerful the target device is, and several prepackaged sets or libraries for different functionalities (Java TV, Java Phone, etc.) This approach avoids the excessive use of resources done in the general approach,

[4] http://java.sun.com/products

but at the same time constraints the range of runtime system libraries available to programmers. If a specific functionality not existing in the reduced runtime system is needed by the programmer, then it must be added to the program by hand in case it is possible, therefore losing the advantages of having a general programming library available in the extended runtime system. Moreover, it may be the case that the runtime system with additional libraries does not fit into the device's memory, but it would fit if the procedures in the library which are not used by the specific application being installed on it were removed.

In contrast, the approach presented in this work is to allow the programmer to use any part of a general runtime system library, and to apply abstract interpretation-based analysis and specialization techniques in order to remove all unnecessary code during execution. This dead code can be removed from both user programs or runtime system libraries, generating a specialized version of the runtime system for a given application.

Moreover, our approach can be easily extended to the specialization of runtime system libraries for a set of programs, instead of specializing them for only one program. Then, a specialized version for the runtime libraries can be generated to be installed in a pervasive device, including exactly the functionalities needed by the set of programs that will execute in such pervasive system.

This work will only take into consideration the specialization of user and runtime system libraries. It will not deal with the specialization of the bytecode emulator itself, which is studied in Deliverable D16 and which is not written in the same source language as the libraries.

2 Execution Model Based on Abstract Machines

A basic execution model is generally composed of three different parts, depicted in Figure 1 and detailed as follows:

- The bytecode which corresponds to user programs.
- The bytecode which implements the runtime system libraries. This code is usually shared among all the user programs installed on the system.[5]
- The abstract machine emulator (or a platform-dependent just-in-time compiler), which interprets (or compiles) the bytecode.

When a runtime system is to be installed on a small device (like pervasive devices) the traditional approach is to define the version of the runtime system which best fits in the resources available in the device.

The main drawback of this approach is that the set of features provided by each runtime version is fixed. If a functionality is not included in the standard runtime system, it must be added manually by the programmer, even if there are pervasive devices powerful enough to host runtime libraries richer than the ones included in the runtime version designed for them.

[5] Although in some cases a compiled user program may comprise the set of libraries needed.

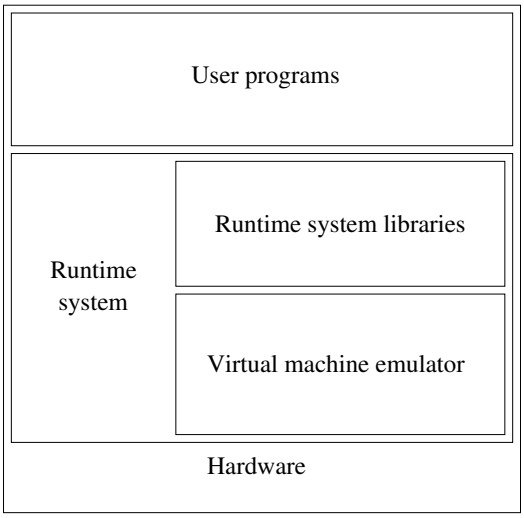

Fig. 1. Generic runtime system

3 Runtime library specialization

A different approach, taken in this work, is to allow the programmer to use the complete full-featured set of libraries of the most general version of the system. During compilation, both the program modules and the runtime libraries are stripped-down for the specific use of that program, using abstract interpretation-based techniques. The advantages are twofold: on one hand, the programmer can use general libraries previously developed for other applications; on the other hand, the runtime libraries are included in the final runtime system only if they are to be used by the program. Moreover, the level of granularity when adding libraries to the runtime system is even finer than traditional compilers (that decide whether a library must be included or not if there are procedures invoked from the program, but they cannot decide if a procedure can be excluded from the library if it is not invoked from any part of the program), as abstract interpretation-based specialization detects and performs dead-code elimination, even when using very simple abstract domains.

This approach provides an additional advantage. When there are several applications that are to be executed in a given pervasive device, the runtime system is usually shared between them. The procedure depicted above can be easily adapted to perform the specialization of the runtime libraries for all the applications in the system. Therefore, the generated runtime system will include all the features needed by those applications and specialized for them, but it will not include other libraries or procedures inside libraries not used by the given set of programs.

4 A Practical application using CiaoPP and Ciao

4.1 The Ciao Runtime System Structure

The Ciao runtime system has the same general structure than other runtime systems like Java. As detailed before, it is composed of a bytecode emulator written in C and a wide set of engine and system libraries. The Ciao language includes a strict module system [1]. System libraries are encapsulated in Ciao modules, although some internal libraries are written in C for several technical reasons (some libraries are needed by the virtual machine, others have strict efficiency requirements, or they need to access low-level operating system resources).

Ciao libraries look like a user Ciao module, although they present slight differences: they are precompiled, and can be used from a user program using a `use_module(library(...))` construct, instead of including the complete path to the library module file. Even built-in procedures (not written in Prolog but embedded in the runtime engine) are listed in Ciao library modules, denoting with `impl_defined` declarations that they are not defined inside the module. The compiler then links the built-in predicate declaration with the actual fragment of runtime engine code. This approach to built-ins and libraries allows a very high degree of library specialization.

Ciao libraries are classified into two categories: *engine libraries*, those which are mandatory for the execution of any Ciao program, and the remaining libraries, which are necessary only if the user program needs their functionality. Figure 2 shows the structure of the Ciao runtime system.

By considering libraries as regular user files, the Ciao compiler is able to determine which libraries are needed for the user program: the compiled program will have the minimal set of libraries, instead of all runtime libraries as traditional runtime systems. This means that the compiler strips-down the runtime system at a module level, but if a library module is included, all procedures in the module will be included, even if they are not used by the program. In the following sections a finer-grained runtime system reduction is proposed.

4.2 Analysis and Specialization of modular Ciao programs

The analysis and specialization is performed using CiaoPP, the Ciao preprocessor, based on abstract interpretation. Only programs written in the Ciao language and its extensions can be processed, and library code implemented in C in the runtime engine cannot be processed nor specialized. Therefore, procedures declared as `impl_defined` are conservatively handled by the preprocessor.

The analysis of a modular program in CiaoPP is implemented following the framework described in [2]. In summary, modules in the program are analyzed in turn, marking the call patterns for imported predicates as pending for analysis. When the imported module is analyzed, all pending patterns are processed. If the analysis results are more precise than those obtained in previous analyses of that module, the modules which import it are marked for reanalysis. This process

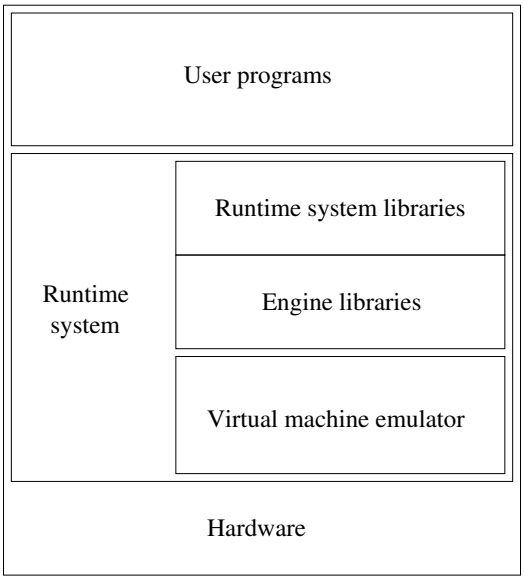

Fig. 2. Ciao runtime system

terminates when there are no marked modules in the program with pending call patterns.

For this work a very simple abstract domain has been used. It only contains two abstract values, $\top$ and $\bot$, representing if a given predicate is used or not. More complex domains would bring more precise results on code reachability.

The results of this inter-modular analysis framework are the starting point of the specialization of modular programs. The specializer takes the list of calling patterns generated for every module, and removes the code that is unreachable from these calling patterns.

4.3 General algorithm for Runtime Generation

Given the analyzer and specializer for modular programs included in `CiaoPP`, the procedure for generating runtime libraries for a given program is as follows:

1. Determine the inter-modular graph of the program, including all needed libraries
2. Copy all these files to a separate place.
3. Perform the analysis of the copy of the user program and the copied libraries.
4. Perform the analysis of special startup code (in order not to lose code to be executed before the main predicate of the user program, as explained below).
5. Specialize the modular program, generating transformed source files for all the modules of the program and libraries.

	Not specialized		Specialized
program	default libs.	manual	
minimal	2,260,293	816,835	501,081
qsort	2,277,134	822,275	504,374
queens	2,263,025	833,441	503,140

Table 1. Executable size comparison in bytes.

4.4 Empirical results

In a first approach, user programs and non-engine libraries were considered for specialization. Engine libraries were excluded, as they were thought as not specializable. However, the results were not as good as one would expect: around a 10% of code reduction in a minimal program. Some engine libraries needed other libraries defined in `Ciao`, losing opportunities for specialization (e.g., the sorting library is needed by aggregation predicates for implementing `setof/3`, needed in turn by the debugger, used by the engine).

The second approach is to generate a specialized version of all runtime system libraries, including engine libraries. All libraries are analyzed and specialized, leaving procedures implemented in C unchanged, but specializing all `Ciao` code.

In this approach, some specific engine modules require special treatment. During analysis and specialization, predicates are marked as needed by the program along the list of modules starting from the startup predicate (defined in the user program as `main/0` or `main/1`). Nevertheless, as mentioned before there is some startup code written in `Ciao` which is executed before the user program starts, and which therefore needs to be preserved in the final, specialized code. As this code is not called from any point of the user program nor the libraries, it will be removed by the specializer. Therefore, additional calling patterns for such code must be provided to `CiaoPP` together with the user program calling patterns.

The second approach brings much better results (even if they are still preliminary since the system can be improved significantly), and they are detailed in Table 1 for some simple examples.

In this table, Numbers correspond to a static compilation of the examples, which includes the libraries needed by the program, but does not include the virtual machine emulator itself. The first example is the smallest `Ciao` program, while `qsort` and `queens` are simple benchmarks which include some additional libraries: `qsort` uses `append/3` from the lists handling library and `write/1` for printing out the results, and `queens` uses the `Ciao` statistics library to get the time spent in the benchmark and `format/2` for formatted output.

the second and third columns correspond to the traditional compilation of the programs, including the default set of `Ciao` libraries in the first case, or just the minimal set of libraries, in the second case. The fourth column is the result of specializing the code of programs and libraries, removing dead code.

5 Conclusions and Future Work

Despite the preliminary nature of this work, We have already obtained an important reduction on size of libraries by removing library procedures which are not used by a given program. Furthermore, a significant improvement is expected using richer domains, as other abstract domains can detect additional fragments of unused code (for example, using modes domains in programs with tests on the instantiation of variables). Another source of improvement is the detection of dead-code for built-ins written in C. Currently procedures written in C are not removed from the system engine even if they are not called from anywhere in the program. A procedure can be implemented to get the annotations produced by `CiaoPP` analyzer to generate C code only for the library procedures which are used in the program. This work will be complemented with the work on Deliverable D16 regarding abstract machine specialization, in order to cover more opportunities for size reduction. We expect the combination of these techniques to result in very significantly reduced footprints for executables, as required in pervasive systems.

References

1. D. Cabeza and M. Hermenegildo. The Ciao Module System: A New Module System for Prolog. In *Special Issue on Parallelism and Implementation of (C)LP Systems*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.
2. G. Puebla, J. Correas, M. Hermenegildo, F. Bueno, M. García de la Banda, K. Marriott, and P. J. Stuckey. A Generic Framework for Context-Sensitive Analysis of Modular Programs. In M. Bruynooghe and K. Lau, editors, *Program Development in Computational Logic, A Decade of Research Advances in Logic-Based Program Development*, number 3049 in LNCS, pages 234–261. Springer-Verlag, Heidelberg, Germany, August 2004.