TLP **12** (4–5): 583–600, 2012. © Cambridge University Press 2012
doi:10.1017/S1471068412000191

Annotation of logic programs for independent AND-parallelism by partial evaluation[*]

GERMAN VIDAL
MiST, DSIC, Universitat Politècnica de València, Camino de Vera, S/N, 46022 Valencia, Spain
(*e-mail:* gvidal@dsic.upv.es)

Abstract

Traditional approaches to automatic AND-parallelization of logic programs rely on some static analysis to identify independent goals that can be safely and efficiently run in parallel in any possible execution. In this paper, we present a novel technique for generating annotations for independent AND-parallelism that is based on partial evaluation. Basically, we augment a simple partial evaluation procedure with (run-time) groundness and variable sharing information so that parallel conjunctions are added to the residual clauses when the conditions for independence are met. In contrast to previous approaches, our partial evaluator is able to transform the source program in order to expose more opportunities for parallelism. To the best of our knowledge, we present the first approach to a *parallelizing* partial evaluator.

KEYWORDS: partial evaluation, automatic parallelization, program analysis

1 Introduction

With the widespread adoption of multi-core processors, the generation of automatic parallelizing compilers becomes an urgent need. On the other hand, there exist a number of program optimization techniques (like partial evaluation (Jones *et al.* 1993)) that have not considered the introduction of parallelism so far, thus limiting its potential for improving program performance.

In this work, we tackle the definition of a parallelizing *partial evaluator* which is able to automatically generate annotations for independent AND-parallelism from logic programs. In contrast to traditional approaches to automatic AND-parallelization of logic programs (which rely on some static analyses to identify independent goals that can be safely and efficiently run in parallel in any possible execution), our approach combines both run-time analyses and the dynamic information gathered during partial evaluation. Furthermore, it allows us to transform the source program in order to expose more opportunities for parallelism (e.g., we can have different specializations of a given clause so that some of them are parallelized and some are not, without adding run-time conditions).

[*] This work has been partially supported by the Spanish *Ministerio de Economía y Competitividad (Secretaría de Estado de Investigación, Desarrollo e Innovación)* under grant TIN2008-06622-C03-02 and by the *Generalitat Valenciana* under grant PROMETEO/2011/052.

Partial evaluation. Partial evaluation (Jones *et al.* 1993) is a well-known technique for program specialization. From a broader perspective, some partial evaluators are also able to optimize programs further by, e.g., shortening computations, removing unnecessary data structures and composing several procedures or functions into a comprehensive definition. Within this broader approach, given a program and a *partial* (incomplete) call, the essential components of partial evaluation are: the construction of a *finite* representation—generally a graph—of the possible executions of (any instance of) the partial call, followed by the systematic extraction of a *residual* program (i.e., the partially evaluated program) from this graph. Intuitively, optimization can be achieved by compressing paths in the graph, by deleting unfeasible paths, and by renaming expressions while removing unnecessary function symbols. In this paper, we propose a novel source of optimization based on transforming some sequential constructions of residual programs into parallel ones.

The theoretical foundations of partial evaluation for (normal) logic programs was first put on a solid basis by Lloyd and Shepherdson (1991). When *pure* logic programs are considered, the term *partial deduction* is often used. Roughly speaking, in order to compute the partial deduction of a logic program P w.r.t. a set of atoms $\mathscr{A} = \{A_1, \ldots, A_n\}$, one should construct finite—possibly incomplete—SLD trees for the atomic goals $\leftarrow A_1, \ldots, \leftarrow A_n$, such that every leaf is either successful, a failure, or only contains atoms that are *instances* of $\{A_1, \ldots, A_n\}$; this is the so-called *closedness* condition (Lloyd and Shepherdson 1991). The residual program then includes a *resultant* of the form $A_i\sigma \leftarrow Q$ for every non-failing root-to-leaf derivation $\leftarrow A_i \hookrightarrow^*_\sigma \leftarrow Q$ in the SLD trees. Similarly, we say that a residual program P' is *closed* when every atom in the body of the clauses of P' is an instance of a partially evaluated atom (i.e., an appropriate specialized definition exists).

From an algorithmic perspective, in order to partially evaluate a program P w.r.t. an atom A, one starts with the initial set $\mathscr{A}_1 = \{A\}$ and builds a *finite* (possibly incomplete) SLD tree for $\leftarrow A$. Then, all atoms in the leaves of this SLD tree which are not instances of A are added to the set, thus obtaining $\mathscr{A}_2$, and so forth. In order to keep the sequence $\mathscr{A}_1, \mathscr{A}_2, \ldots$ finite, some *generalization* is often required, e.g., by replacing some predicate arguments with fresh variables. Some variant of the *homeomorphic embedding* ordering (Leuschel 2002) is often used to detect potential sources of non-termination.

A sketch of this algorithm is shown in Figure 1, where the unfolding rule $unf(\mathscr{A}_i)$ builds finite SLD trees for the atoms in $\mathscr{A}_i$ and returns the associated resultants, function *atoms* returns the atoms in the bodies of these resultants, and the abstraction operator $abs(\mathscr{A}_i, \mathscr{A}')$ returns an approximation of $\mathscr{A}_i \cup \mathscr{A}'$ so that the sequence $\mathscr{A}_1, \mathscr{A}_2, \ldots$ is kept finite.

Motivation. Depending on *when* control issues—like deciding which atoms should or should not be unfolded—are addressed, two main approaches to partial evaluation can be distinguished. In *offline* approaches to partial evaluation, these decisions are taken beforehand by means of a so called *binding-time* analysis (where we know which parameters are known but not their values). In contrast, *online* partial evaluators take decisions on the fly (so that actual values of static data are available).

```
Initialization: $i := 1$;  $\mathcal{A}_i := \{A\}$;
Repeat
        $\mathcal{A}_{i+1} := abs(\mathcal{A}_i, atoms(unf(\mathcal{A}_i)))$;
        $i := i + 1$
Until  $\mathcal{A}_i \approx \mathcal{A}_{i-1}$ (variants)
Return  $unf(\mathcal{A}_i)$
```

Fig. 1. Partial evaluation procedure.

While offline partial evaluators are usually faster, online ones produce more accurate results. Partial evaluators for logic programs have mostly followed the online approach (e.g., SAGE (Gurr 1994), Mixtus (Sahlin 1990), SP (Gallagher 1991), ECCE (Leuschel *et al.* 2006)), though some offline partial evaluators have been also developed (e.g., LOGEN (Leuschel *et al.* 2006)).

Recently, we have proposed in (Vidal 2011) a hybrid approach to partial evaluation that does not fit well in neither the offline nor the online style of partial evaluation. Basically, we follow a typical online partial evaluation scheme, but augment it with run-time information gathered from a pre-processing static analysis. There are some previous approaches that combine the online and offline styles of partial evaluation. However, the novelty is that (Vidal 2011) considers collecting *run-time* information rather than *partial evaluation* time information in a pre-processing stage (as *binding-time* analyses do).

In this paper, we want to push this approach forward by defining a parallelizing partial evaluator that generates annotations for independent AND-parallelism.

As it is well known, two goals $(G_1, G_2)\theta$ are *strictly independent* if, for every pair of variables $(x, y) \in (\mathcal{V}ar(G_1), \mathcal{V}ar(G_2))$, either (i) they are equal, $x = y$, and $x\theta$ is ground (i.e., $\mathcal{V}ar(x\theta) = \emptyset$) or (ii) they are different, $x \neq y$, and their values, $x\theta$ and $y\theta$, do not share a common variable (i.e., $\mathcal{V}ar(x\theta) \cap \mathcal{V}ar(y\theta) = \emptyset$). In order to have this information available at partial evaluation time, we need some *run-time* information that is not usually present in partial evaluation schemes. For this purpose, we introduce a hybrid partial evaluation scheme with the following features:

- First, a pre-processing stage performs both a groundness and sharing analysis, so that we get call and success patterns for each predicate.
- Then, we apply a rather simple partial evaluation stage that only performs one-step unfolding. This is very limited in general and propagates almost no information. However, in our context, we do not aim at aggressively propagating static data but only groundness and sharing information. In this way, the potential for generating annotations for the implicit independent AND-parallelism can be better evaluated.
- Finally, a post-processing stage extracts residual rules from the partial evaluation computations and, in some cases, replaces sequential conjunctions by parallel ones, thus boosting the performance of the residual program.

A proof-of-concept implementation of the parallelizing partial evaluator is available at `http://kaz.dsic.upv.es/litep.html`. Despite its simplicity (a thousand lines of Prolog code), the results for definite logic programs (including some built-in's) are very encouraging.

The paper is organized as follows. Section 2 presents the different stages of our parallelizing partial evaluation scheme. Then, Section 3 summarizes our findings from an experimental evaluation of the new technique and, finally, Sect. 4 concludes and discusses some possibilities for future work. Correctness results can be found in the online appendix.

2 Parallelizing partial evaluation

In this section, we present our partial evaluation scheme in a stepwise manner. We do so for clarity of presentation but these stages can be interleaved (and actually they are in our implementation).

2.1 *Pre-processing stage*

Our pre-processing stage consists of two different analyses. The first one is a simple call and success pattern analysis that resembles a mode analysis. The formal definition of the analysis can be found elsewhere (e.g., in (Leuschel and Vidal 2009)).

We consider *groundness* call and success patterns π denoted by a list of natural numbers which represent the (definitively) ground arguments of a predicate. The underlying abstract domain is thus very simple: $\{$definitively ground, possibly non-ground$\}$. As mentioned in (Leuschel and Vidal 2009), the analysis could be made more precise by considering a richer abstract domain (including elements like list, nonvar, etc). This is orthogonal to the topics of this paper and thus we keep the two element domain for simplicity. The greatest lower bound operator $\sqcap$ on patterns is defined in the natural way by the set union, i.e., given two patterns π_1, π_2 for predicate p/n, we let $\pi_1 \sqcap \pi_2 = \pi_1 \cup \pi_2$.

Basically, given an initial query and the groundness call patterns for the atoms in this query, the analysis infers for every predicate p/n a number of call and success patterns of the form $p/n : \pi_{in} \overset{gr}{\mapsto} \pi_{out}$ such that π_{in} and π_{out} are subsets of $\{1, \ldots, n\}$ denoting the arguments π_{out} of p/n which are definitely ground after a successful derivation, assuming that it is called with ground arguments π_{in}. The analysis is started with a number of *entry points* to the program, together with their initial groundness call patterns.

Example 1
Consider the well known definition of *append*/3:

$$\begin{array}{lcl} append([\,], Y, Y). & & \\ append([H|T], Y, [H|TY]) & \leftarrow & append(T, Y, TY). \end{array}$$

Given the initial groundness call patterns $\pi_1 = \{1\}$ and $\pi_2 = \{1, 2\}$ for *append*/3, the call and success pattern analysis would return the following mappings:

$$append/3 : \ \{1\} \ \overset{gr}{\mapsto} \ \{1\} \qquad append/3 : \ \{1,2\} \ \overset{gr}{\mapsto} \ \{1,2,3\}$$

Their meaning should be clear: if $append(t_1, t_2, t_3)$ is called with t_1 ground, we can only ensure that t_1 will be ground after a successful derivation. In contrast, if it is called with both t_1 and t_2 ground, then t_3 will be also ground after a successful derivation.

For guaranteeing the independence of goals, we also consider the information gathered by a dependency analysis like that of (Debray 1989). Basically, for a given predicate $p/3$, the analysis computes mappings with *sharing* call and success patterns μ like, e.g., $\langle\{1,2\},\{1,2,3\},\{2,3\}\rangle$, which indicates that the first argument may share variables with the second argument, the second argument may share variables with the first and third arguments, and the third argument may share variables with the second argument. Again, the analysis infers for every predicate p/n a number of call and success patterns of the form $p/n : \mu_{in} \overset{sh}{\mapsto} \mu_{out}$ such that μ_{in} and μ_{out} belong to the domain $2^{\{1,\ldots,n\}} \times \ldots \times 2^{\{1,\ldots,n\}}$ (a tuple of n sets) and μ_{out} denotes the dependencies of p/n which hold after a successful derivation, assuming that it is called with the dependencies denoted by μ_{in}.[1]

In this case, the least upper bound operator $\sqcup$ on sharing patterns is defined as follows: given patterns $\mu = \langle\vartheta_1, \ldots, \vartheta_n\rangle$ and $\mu' = \langle\vartheta'_1, \ldots, \vartheta'_n\rangle$ for some predicate p/n, we have $\mu \sqcup \mu' = \langle\vartheta_1 \cup \vartheta'_1, \ldots, \vartheta_n \cup \vartheta'_n\rangle$. Note that, in contrast to the greatest lower bound on groundness patterns that may increase the number of ground variables (and thus the accuracy of the result), the least upper bound on sharing patterns may lose accuracy since more dependencies can be obtained.

Example 2
Consider again $append/3$. Given the sharing call patterns $\mu_1 = \langle\{1\},\{2\},\{3\}\rangle$ and $\mu_2 = \langle\{1,2\},\{1,2\},\{3\}\rangle$, the dependency analysis would return the following:

$$\begin{array}{lll} append/3: & \langle\{1\},\{2\},\{3\}\rangle & \overset{sh}{\mapsto} & \langle\{1,3\},\{2,3\},\{1,2,3\}\rangle \\ append/3: & \langle\{1,2\},\{1,2\},\{3\}\rangle & \overset{sh}{\mapsto} & \langle\{1,2,3\},\{1,2,3\},\{1,2,3\}\rangle \end{array}$$

Here, we consider two possibilities: first, if *append* is called with three independent arguments then, after a successful derivation, the third argument may be bound to a value that shares variables with either the first and the second arguments; on the other hand, if *append* is called with the two first arguments bound to terms containing shared variables, then all three arguments may depend on each other after a successful derivation.

2.2 *Partial evaluation stage*

Now, we present the proper partial evaluation stage of the parallelizing partial evaluator.

[1] We note that a sharing pattern like $\langle\{1\},\{2\},\{3\}\rangle$ assumes that all three argument are independent and, moreover, that no variable sharing can be introduced through a single argument; i.e., we assume that predicate arguments are always linear. We keep this restriction for simplicity but could easily be overcome.

In principle, one could consider checking independence of goals using the information available solely at partial evaluation time. This approach, however, would be generally incorrect for a number of reasons. First, the notion of *closedness* (see Sect. 1) allows run-time atoms to be covered by instances of partial evaluation atoms. Therefore, $q(X,X)$ is closed w.r.t. $q(X,Y)$. This means that goals can be independent at partial evaluation time but need not be independent at run-time. Moreover, whenever we split a goal of an incomplete computation into atomic subgoals, we are also loosing some *context* information that might be essential for checking independence, as the following example illustrates:

Example 3
Consider the following program

$$\begin{array}{l} p(X,Y) \leftarrow q(X), r(Y). \\ eq(X,X). \\ \ldots \end{array}$$

Given the goal $eq(A,B), p(A,B)$, if we split it into its atomic subgoals $eq(A,B)$ and $p(A,B)$, and partially evaluate them independently, we could derive the goal $q(A), r(B)$ and *incorrectly* assume that $q(A)$ and $r(B)$ are independent.

Furthermore, the use of an abstraction operator might also involve the loss of some dependencies (e.g., generalizing $p(X,Y,f(Y))$ to $p(X,Y,Z)$ with Z a fresh variable).

In summary, the information available at partial evaluation time is not enough to determine the run-time independence of a goal.[2] Therefore, as mentioned before, in this paper we consider that the partial evaluator includes a pre-processing stage where *run-time* groundness and sharing information is gathered.

In particular, we design a rather simple partial evaluator with the following distinguishing features:

- only one-step unfolding of atomic goals is performed;
- no static data are provided (i.e., the initial goal has different variables as arguments);
- every atomic goal is enriched with groundness and sharing call patterns that are propagated through partial evaluation.

The fact that we do not consider partially instantiated initial goals, together with the fact that only one-step unfolding is performed, allows us to better identify the potential for generating annotations for independent AND-parallelism. Moreover, it makes the online partial evaluator scale up better to medium and large applications.

Our partial evaluator deals with sets of *extended* atoms (instead of sets of atoms, as in the algorithm of Figure 1).

[2] Of course, we could avoid splitting goals, do not use an abstraction operator and only allow variants to be closed, but then the termination of partial evaluation could not be ensured.

Definition 1 (*extended atom*)

We consider extended atoms of the form (A, π, μ) where A is an atom, π is a groundness call pattern for A, and μ is a sharing call pattern for A. This notion is extended in the natural way to queries and goals. We denote the empty extended query by *true*.

Given an extended query $\mathcal{Q}$, we introduce the following auxiliary function: $query(\mathcal{Q}) = A_1, \ldots, A_n$, if $\mathcal{Q} = (A_1, \pi_1, \mu_1), \ldots, (A_n, \pi_n, \mu_n)$.

The number of different specialized versions of an atom will be determined, not only by its shape (as it is usually the case), but also by the different combinations of groundness and sharing call patterns. For instance, $(p(X,Y), \{1,2\}, \langle\{1\},\{2\}\rangle)$ and $(p(X,Y), \{1\}, \langle\{1\},\{2\}\rangle)$ would give rise to different specialized versions.

Another distinguishing feature of our scheme is that, in contrast to previous approaches, we do not explicitly distinguish between the so-called *local* and *global* levels (as in (Gallagher 1993)). Rather, we construct a single partial evaluation tree that comprises both levels. Moreover, our partial evaluation process performs just one pass since residual rules can be produced immediately after every unfolding step (rather than in a post-process, as it is often done since the unfolding tree can be modified during the partial evaluation process).

In the following, we denote by $\pi(A)$ the (definite) *ground* arguments of A according to π, i.e., $\pi(p(s_1, \ldots, s_k)) = \{s_j \mid j \in \pi\}$. Also, we denote by $\mu(A)$ the set of (possibly) shared variables in A according to μ, i.e., $\mu(p(s_1, \ldots, s_k)) = \{(x, y) \in (\mathcal{V}ar(s_i), \mathcal{V}ar(s_j)) \mid i, j \in \{1, \ldots, k\}, i \neq j, \{i, j\} \subseteq s \in \mu\}$. Before introducing the notion of SLD resolution over extended queries, we need the following preparatory definition, which is used to propagate groundness and sharing call patterns to the atoms in the body of a clause.

Definition 2 (*entry procedure*)

Let $H \leftarrow B_1, \ldots, B_n$ be a clause and (A, π, μ) an extended atom such that A and H unify. We denote with *entry* a function that propagates π and μ to $B_1, \ldots, B_n$. Formally, $entry(\pi, \mu, (H \leftarrow B_1, \ldots, B_n)) = ((B_1, \pi_1, \mu_1), \ldots, (B_n, \pi_n, \mu_n))$ if, for all $B_i = p_i(t_{i1}, \ldots, t_{im_i})$, $i = 1, \ldots, n$, the following conditions hold:

- $j \in \pi_i$ iff $\mathcal{V}ar(t_{ij}) \subseteq \mathcal{V}ar(\pi(H))$ (i.e., all variables in t_{ij} are ground in H according to π).
- $\{1, \ldots, m_i\} \supseteq \{j_1, \ldots, j_k\} \in \mu_i$ iff there are (non necessarily different) variables $(x_{j_1}, \ldots, x_{j_k}) \in (\mathcal{V}ar(t_{ij_1}), \ldots, \mathcal{V}ar(t_{ij_k}))$ such that for every pair of different variables x_{j_r}, x_{j_s}, we have $(x_{j_r}, x_{j_s}) \in \mu(H)$ (i.e., either the terms share some variable or have different variables that are shared in H according to μ).

Note that the entry procedure is independent of A (only its associated groundness and sharing call patterns matter), since we want the results for a partial evaluation time atom A be valid for every run-time atom $A\theta$.

Example 4

Let us consider the following program for computing Fibonacci numbers:

$$\begin{array}{llll}
(C_1) & fibonacci(0,1). & & (C_2)\ \ fibonacci(1,1). \\
(C_3) & fibonacci(M,N) & \leftarrow & M > 1,\ M1\ is\ M-1,\ fibonacci(M1,N1), \\
 & & & M2\ is\ M-2,\ fibonacci(M2,N2),\ N\ is\ N1+N2.
\end{array}$$

Here, $entry(\{1\},\langle\{1\},\{2\}\rangle,C_3)$ returns the following extended query:

$$\begin{array}{rll}
(M > 1, & \{1,2\}, & \langle\{1\},\{2\}\rangle), \\
(M1\ is\ M-1, & \{2\}, & \langle\{1\},\{2\}\rangle), \\
(fibonacci(M1,N1), & \{\}, & \langle\{1\},\{2\}\rangle), \\
(M2\ is\ M-2, & \{2\}, & \langle\{1\},\{2\}\rangle), \\
(fibonacci(M2,N2), & \{\}, & \langle\{1\},\{2\}\rangle), \\
(N\ is\ N1+N2, & \{\}, & \langle\{1\},\{2\}\rangle)
\end{array}$$

We are now ready to introduce the notion of extended SLD resolution:

Definition 3 (*extended SLD resolution*)

Extended SLD resolution, denoted by $\rightsquigarrow$, is a natural extension of SLD resolution over extended queries. Formally, given a program P, an extended query $\mathcal{Q} = (A_1,\pi_1,\mu_1),\ldots,(A_n,\pi_n,\mu_n)$, and a computation rule $\mathcal{R}$, we say that $\leftarrow \mathcal{Q} \rightsquigarrow_{P,\mathcal{R},\sigma} \leftarrow \mathcal{Q}'$ is an *extended SLD resolution step* for $\mathcal{Q}$ with P and $\mathcal{R}$ if the following conditions hold:[3]

- $\mathcal{R}(\mathcal{Q}) = (A_i,\pi_i,\mu_i)$, $1 \leqslant i \leqslant n$, is the selected extended atom,
- $H \leftarrow B_1,\ldots,B_m$ is a renamed apart clause of P,
- A_i and H unify with $\sigma = mgu(A_i,H)$, and
- $\mathcal{Q}' = entry(\pi_i,\mu_i,(H \leftarrow B_1,\ldots,B_m))\sigma$.[4]

Trivially, extended SLD resolution is a conservative extension of SLD resolution: given extended queries $\mathcal{Q},\mathcal{Q}'$, we have that $\leftarrow \mathcal{Q} \rightsquigarrow_\sigma \leftarrow \mathcal{Q}'$ implies $\leftarrow query(\mathcal{Q}) \hookrightarrow_\sigma \leftarrow query(\mathcal{Q}')$.

In the following, we use $pred(A)$ to denote the predicate symbol of atom A.

As it is common practice, we avoid infinite unfolding by means of a well-known strategy based on the use of the *homeomorphic embedding* ordering. Intuitively, we say that atom A_i *embeds* atom A_j, denoted by $A_i \trianglerighteq A_j$, when A_j can be obtained from A_i by deleting symbols (see (Leuschel 2002) for a precise definition).

Definition 4 (*variant, embedding*)

We say that two (extended) atoms (A,π,μ) and (A',π',μ') are *variants*, denoted by $(A,\pi,\mu) \approx (A',\pi',\mu')$ if there is a renaming substitution ρ such that $A\rho = A'$, $\pi = \pi'$ and $\mu = \mu'$.

We say that (A,π,μ) *embeds* (A',π',μ'), denoted by $(A,\pi,\mu) \trianglerighteq (A',\pi',\mu')$, if $A \trianglerighteq A'$, $\pi = \pi'$ and $\mu = \mu'$.

[3] We often omit P, $\mathcal{R}$ and/or σ in the notation of an extended SLD resolution step when they are clear from the context.

[4] We let $((B_1,\pi_1,\mu_1),\ldots,(B_n,\pi_n,\mu_n))\sigma = (B_1\sigma,\pi_1,\mu_1),\ldots,(B_n\sigma,\pi_n,\mu_n)$.

(variant) $$\frac{\exists (A', \pi', \mu') \in \mathit{memo}.\ (A, \pi, \mu) \approx (A', \pi', \mu')}{\langle (A, \pi, \mu), \mathcal{Q}; \mathit{memo}\rangle \xrightarrow{v} \langle \mathcal{Q}; \mathit{memo}\rangle}$$

(failure) $$\frac{\nexists \mathcal{Q}'.\ \leftarrow (A, \pi, \mu) \rightsquigarrow_\sigma \leftarrow \mathcal{Q}'}{\langle (A, \pi, \mu), \mathcal{Q}; \mathit{memo}\rangle \xrightarrow{f} \langle \mathcal{Q}; \mathit{memo}\rangle}$$

(embedding) $$\frac{\exists (A', \pi', \mu') \in \mathit{memo}.\ (A, \pi, \mu) \trianglerighteq (A', \pi', \mu')}{\langle (A, \pi, \mu), \mathcal{Q}; \mathit{memo}\rangle \xrightarrow{e} \langle \mathcal{Q}; \mathit{memo}\rangle}$$

(nonuser) $$\frac{\mathit{pred}(A) \text{ is not defined in the user's program clauses}}{\langle (A, \pi, \mu), \mathcal{Q}; \mathit{memo}\rangle \xrightarrow{n} \langle \mathcal{Q}; \mathit{memo}\rangle}$$

(parallel) $$\frac{\leftarrow (A, \pi, \mu) \rightsquigarrow_\sigma \leftarrow \mathcal{Q}' \ \wedge\ \exists (\mathcal{Q}_1, \mathcal{Q}_2, \mathcal{Q}_3, \mathcal{Q}_4) \in \mathit{partition}_\mu(\mathcal{Q}')}{\langle (A, \pi, \mu), \mathcal{Q}; \mathit{memo}\rangle \xrightarrow{p}_\sigma \langle \mathcal{Q}_1, \mathcal{Q}_2, \mathcal{Q}_3, \mathcal{Q}_4, \mathcal{Q}; \mathit{memo} \cup \{(A, \pi, \mu)\}\rangle}$$

(unfolding) $$\frac{\leftarrow (A, \pi, \mu) \rightsquigarrow_\sigma \leftarrow \mathcal{Q}' \ \wedge\ \nexists (\mathcal{Q}_1, \mathcal{Q}_2, \mathcal{Q}_3, \mathcal{Q}_4) \in \mathit{partition}_\mu(\mathcal{Q}')}{\langle (A, \pi, \mu), \mathcal{Q}; \mathit{memo}\rangle \xrightarrow{u}_\sigma \langle \mathit{prop}(\mathcal{Q}', \mathit{true}), \mathcal{Q}; \mathit{memo} \cup \{(A, \pi, \mu)\}\rangle}$$

Fig. 2. Partial evaluation semantics.

Our partial evaluation semantics is formalized by means of the (labelled) state transition system shown in Figure 2. The partial evaluator deals with *states*, defined as follows:

Definition 5 (*state*)
A *state* is a pair of the form $\langle \mathcal{Q}; \mathit{memo}\rangle$ where $\mathcal{Q}$ is a sequence of extended atoms[5] and *memo* is a set of extended atoms (the atoms already partially evaluated, which are recorded to guarantee termination).

An *initial state* has the form $\langle (A, \pi, \mu); \{\}\rangle$. A *final state* has the form $\langle \epsilon; \mathit{memo}\rangle$, where ϵ denotes an empty sequence.

A *successful* partial evaluation starts with an initial state and (non-deterministically, because of the unfolding rule) constructs a number of derivations of the form $\langle (A, \pi, \mu); \{\}\rangle \longrightarrow^* \langle \epsilon; \_\rangle$, where $\longrightarrow^*$ denotes the reflexive and transitive closure of $\longrightarrow$. The process does not return anything but the trace itself, that will be used for producing residual rules (see the next section).

Let us now explain the rules of the partial evaluation semantics. Rule (variant) discards an extended atom if it is a variant of an already partially evaluated extended atom. Rule (failure) also discards an extended atom when it cannot be unfolded (e.g., when A does not unify with the head of any clause).

The next rule, (embedding), discards an extended atom when it embeds a previously partially evaluated extended atom. This rule is necessary in order to ensure that partial evaluation always terminates. Rule (nonuser) allows us to deal with built-in's and other extra-logical features of Prolog by leaving calls to the original predicates, as we will see in the next section.

[5] Note that this sequence is not an extended query. Rather, this is the queue of (extended) atomic goals to be partially evaluated.

The interesting rules are (parallel) and (unfolding). In the following, we assume a fixed left-to-right selection rule as in Prolog. Therefore, we use a function *prop* to propagate groundness and sharing success patterns to the atoms to the right of a given atom before splitting an extended query. This is necessary because only the partial evaluation of atomic goals is allowed and, thus, this information should be propagated before the query is split into its constituents in order to avoid a serious loss of accuracy.

Definition 6 (*pattern propagation*)
Let $\mathcal{Q}_1, \mathcal{Q}_2$ be extended queries, with $\mathcal{Q}_1 = (A_1, \pi_1, \mu_1), \ldots, (A_n, \pi_n, \mu_n)$ and $\mathcal{Q}_2 = (A_{n+1}, \pi_{n+1}, \mu_{n+1}), \ldots, (A_m, \pi_m, \mu_m)$. We define the function *prop* to propagate success patterns to the right as follows:[6]

- $prop(\mathcal{Q}_1, \mathcal{Q}_2) = \mathcal{Q}_2$ if $n = 0$ (i.e., $\mathcal{Q}_1$ is an empty query);
- $prop(\mathcal{Q}_1, \mathcal{Q}_2) = ((A_1, \pi_1, \mu_1), prop(\mathcal{Q}'_1, \mathcal{Q}'_2))$ if $n > 0$,
  $pred(A_1) : \pi_1 \overset{gr}{\mapsto} \pi'_1$, $pred(A_1) : \mu_1 \overset{sh}{\mapsto} \mu'_1$,
  $entry(\pi'_1, \mu'_1, (A_1 \leftarrow A_2, \ldots, A_m)) = (A_2, \pi'_2, \mu'_2), \ldots, (A_m, \pi'_m, \mu'_m)$,
  $\mathcal{Q}'_1 = (A_2, \pi_2 \sqcap \pi'_2, \mu_2 \sqcup \mu'_2), \ldots, (A_n, \pi_n \sqcap \pi'_n, \mu_n \sqcup \mu'_n)$, and
  $\mathcal{Q}'_2 = (A_{n+1}, \pi_{n+1} \sqcap \pi'_{n+1}, \mu_{n+1} \sqcup \mu'_{n+1}), \ldots, (A_m, \pi_m \sqcap \pi'_m, \mu_m \sqcup \mu'_m)$.

Observe that the two arguments of function *prop* are not needed for unfolding a goal. However, this formulation will become useful later when also using *prop* to partition a goal.

Example 5
Consider again the Fibonacci program of Example 4 and the result of the *entry* procedure. Thus we have $fibonacci(A, B) \rightsquigarrow_{\{A \mapsto M, B \mapsto N\}} (M > 1, \{1,2\}, \langle\{1\},\{2\}\rangle)$, $(M1\ is\ M{-}1, \{2\}, \langle\{1\},\{2\}\rangle), (fibonacci(M1, N1), \{\}, \langle\{1\},\{2\}\rangle), (M2\ is\ M{-}2, \{2\}, \langle\{1\}, \{2\}\rangle), (fibonacci(M2, N2), \{\}, \langle\{1\},\{2\}\rangle), (N\ is\ N1 + N2, \{\}, \langle\{1\},\{2\}\rangle)$. We assume the following call and success patterns:

$$\begin{array}{rlrl} is/2: & \{2\} \overset{gr}{\mapsto} \{1,2\} & is/2: & \langle\{1\},\{2\}\rangle \overset{sh}{\mapsto} \langle\{1\},\{2\}\rangle \\ fibonacci/2: & \{1\} \overset{gr}{\mapsto} \{1,2\} & fibonacci/2: & \langle\{1\},\{2\}\rangle \overset{sh}{\mapsto} \langle\{1\},\{2\}\rangle \end{array}$$

Then, for instance, we have

$$\begin{array}{l} prop(((M > 1, \{1,2\}, \langle\{1\},\{2\}\rangle),\ M1\ is\ M - 1, \{2\}, \langle\{1\},\{2\}\rangle), \\ \quad (fibonacci(M1, N1), \{\}, \langle\{1\},\{2\}\rangle),\ (M2\ is\ M - 2, \{2\}, \langle\{1\},\{2\}\rangle), \\ \quad (fibonacci(M2, N2), \{\}, \langle\{1\},\{2\}\rangle), (N\ is\ N1 + N2, \{\}, \langle\{1\},\{2\}\rangle)),\ true) \\ = ((M > 1, \{1,2\}, \langle\{1\},\{2\}\rangle),\ (M1\ is\ M - 1, \{2\}, \langle\{1\},\{1\}\rangle), \\ \quad (fibonacci(M1, N1), \{1\}, \langle\{1\},\{2\}\rangle),\ (M2\ is\ M - 2, \{2\}, \langle\{1\},\{2\}\rangle), \\ \quad (fibonacci(M2, N2), \{1\}, \langle\{1\},\{2\}\rangle),\ (N\ is\ N1 + N2, \{2\}, \langle\{1\},\{2\}\rangle)) \end{array}$$

so we know that, when the last call $N\ is\ N1 + N2$ is performed, $N1 + N2$ is ground.

[6] Note the non-standard use of function *entry* to propagate success patterns to the right, despite the fact that $A_1 \leftarrow A_2, \ldots, A_m$ is not really a program clause.

Before explaining the rules (parallel) and (unfolding), we still need one more auxiliary function, *partition*, which is used to check if a query contains some subgoals that can be executed in parallel (i.e., if they are strictly independent at run-time):

Definition 7 (*partition*)
Let (A, π, μ) be an extended atom such that $(A, \pi, \mu) \rightsquigarrow_\sigma \mathcal{Q}$. We introduce the function $\mathsf{partition}_\mu$ as follows:[7]

- $\mathit{partition}_\mu(\mathcal{Q}) = (\mathcal{Q}'_1, \mathcal{Q}''_2, \mathcal{Q}''_3, \mathcal{Q}''''_4)$ if $\mathcal{Q}$ contains at least two extended atoms, $\mathcal{Q} = \mathcal{Q}_1, \mathcal{Q}_2, \mathcal{Q}_3, \mathcal{Q}_4$, with $\mathcal{Q}_2$ and $\mathcal{Q}_3$ non-empty queries, $(\mathcal{Q}'_1, (\mathcal{Q}'_2, \mathcal{Q}'_3, \mathcal{Q}'_4)) = \mathit{prop}(\mathcal{Q}_1, (\mathcal{Q}_2, \mathcal{Q}_3, \mathcal{Q}_4))$, $\mathcal{Q}'_2$ and $\mathcal{Q}'_3$ are independent, $(\mathcal{Q}''_2, \mathcal{Q}''_4) = \mathit{prop}(\mathcal{Q}'_2, \mathcal{Q}'_4)$, $(\mathcal{Q}''_3, \mathcal{Q}'''_4) = \mathit{prop}(\mathcal{Q}'_3, \mathcal{Q}''_4)$, and $\mathcal{Q}''''_4 = \mathit{prop}(\mathcal{Q}'''_4, \mathit{true})$.

Here, strict independence of $\mathcal{Q}'_2$ and $\mathcal{Q}'_3$ is checked using the standard notion (see Section 1) and taking into account the groundness call patterns available from the extended atoms and the sharing call pattern for the head of the clause, i.e., the variables in $\mathcal{V}ar(\mathcal{Q}'_2) \cap \mathcal{V}ar(\mathcal{Q}'_3)$ must be ground according to the groundness call patterns in $\mathcal{Q}'_2, \mathcal{Q}'_3$ and each pair of different variables $(x, y) \in (\mathcal{V}ar(\mathcal{Q}'_2), \mathcal{V}ar(\mathcal{Q}'_3))$ should not be shared in the head of the clause according to μ.

Example 6
Consider again the Fibonacci program of Example 4 and the extended SLD resolution step of Example 5. By applying function *partition* to the derived extended query, we get

$$\begin{aligned}
\mathcal{Q}_1 =& (M > 1, \{1\}, \langle\{1\}, \{2\}\rangle),\\
\mathcal{Q}_2 =& (M1\ \mathit{is}\ M - 1, \{2\}, \langle\{1\}, \{2\}\rangle), (\mathit{fibonacci}(M1, N1), \{1\}, \langle\{1\}, \{2\}\rangle),\\
\mathcal{Q}_3 =& (M2\ \mathit{is}\ M - 2, \{2\}, \langle\{1\}, \{2\}\rangle), (\mathit{fibonacci}(M2, N2), \{1\}, \langle\{1\}, \{2\}\rangle),\\
\mathcal{Q}_4 =& (N\ \mathit{is}\ N1 + N2, \{2\}, \langle\{1\}, \{2\}\rangle)
\end{aligned}$$

which means that the queries $(M1\ \mathit{is}\ M - 1, \mathit{fibonacci}(M1, N1))$ and $(M2\ \mathit{is}\ M - 2, \mathit{fibonacci}(M2, N2))$ can be safely run in parallel at run-time.

Now, rules (parallel) and (unfolding) should be clear. When an atom is unfolded and the body of the selected clause can be run in parallel (which is determined by function *partition*), rule (parallel) applies. Note that we consider a simple algorithm where the atoms of a query cannot be reordered (i.e., we respect Prolog's computation rule). Of course, more elaborated strategies exist (see, e.g., (Muthukumar *et al.* 1999; Gras and Hermenegildo 2009)), but we consider them out of the scope of this paper.

When the body of the clause cannot be partitioned so that some subgoals are run in parallel, rule (unfolding) applies (which will give rise to a sequential clause, as we will see later). Here, we apply function *prop* in order to propagate groundness and sharing information to the extended atoms before they are split in the next step (since only the unfolding of atomic goals is considered).

[7] In order not to encumber the notation, we assume that $\mathcal{Q}'_i$ refers to the same extended query $\mathcal{Q}_i$ after some processing.

In both rules, we add the selected extended atom to the set of already partially evaluated extended atoms.

All transition rules are labelled with a letter that identifies the rule applied. This will become useful to generate residual rules (see the next section).

Example 7
Consider again the Fibonacci program of Example 4. Given the initial state

$$\mathcal{S}_0 = \langle (fibonacci(A,B), \{1\}, \langle \{1\}, \{2\} \rangle), \{\} \rangle$$

we have three partial evaluation derivations starting from $\mathcal{S}_0$:

$$\begin{array}{l}
\mathcal{S}_0 \xrightarrow{u}_{\{A \mapsto 0, B \mapsto 1\}} \langle \epsilon, \{(fibonacci(A,B), \{1\}, \langle \{1\}, \{2\} \rangle)\} \rangle \\
\mathcal{S}_0 \xrightarrow{u}_{\{A \mapsto 1, B \mapsto 1\}} \langle \epsilon, \{(fibonacci(A,B), \{1\}, \langle \{1\}, \{2\} \rangle)\} \rangle \\
\mathcal{S}_0 \xrightarrow{p}_{\{A \mapsto M, B \mapsto N\}} \langle (\mathcal{Q}_1, \mathcal{Q}_2, \mathcal{Q}_3, \mathcal{Q}_4), \{(fibonacci(A,B), \{1\}, \langle \{1\}, \{2\} \rangle)\} \rangle \\
\qquad \xrightarrow{n} \langle (\mathcal{Q}_2, \mathcal{Q}_3, \mathcal{Q}_4), \{(fibonacci(A,B), \{1\}, \langle \{1\}, \{2\} \rangle)\} \rangle \\
\qquad \xrightarrow{n} \langle ((fibonacci(M1,N1), \{1\}, \langle \{1\}, \{2\} \rangle), \mathcal{Q}_3, \mathcal{Q}_4), \\
\qquad\qquad \{(fibonacci(A,B), \{1\}, \langle \{1\}, \{2\} \rangle)\} \rangle \\
\qquad \xrightarrow{v} \langle (\mathcal{Q}_3, \mathcal{Q}_4), \{(fibonacci(A,B), \{1\}, \langle \{1\}, \{2\} \rangle)\} \rangle \\
\qquad \xrightarrow{n} \langle ((fibonacci(M2,N2), \{1\}, \langle \{1\}, \{2\} \rangle), \mathcal{Q}_4), \\
\qquad\qquad \{(fibonacci(A,B), \{1\}, \langle \{1\}, \{2\} \rangle)\} \rangle \\
\qquad \xrightarrow{v} \langle (\mathcal{Q}_4), \{(fibonacci(A,B), \{1\}, \langle \{1\}, \{2\} \rangle)\} \rangle \\
\qquad \xrightarrow{n} \langle \epsilon, \{(fibonacci(A,B), \{1\}, \langle \{1\}, \{2\} \rangle)\} \rangle
\end{array}$$

Note that predicates not defined in the user's program (like $>$ or *is*) are not unfoldable and that $\mathcal{Q}_1, \mathcal{Q}_2, \mathcal{Q}_3, \mathcal{Q}_4$ are the extended queries of Example 6.

2.3 *Post-Processing stage*

Once the partial evaluation stage terminates, we produce renamed, residual rules associated to the transitions of the partial evaluation semantics. In the following, we assume that there is a function *ren* that takes an extended atom and returns a renamed atom whose predicate name is fresh and depends on the patterns of the extended atom. We do not present the details of this renaming function here since it is a standard renaming as introduced in, e.g., (Benkerimi and Lloyd 1989; De Schreye *et al.* 1999). For instance,

$$ren(fibonacci(X,Y), \{1\}, \langle \{1\}, \{2\} \rangle) = fibonacci\_1\_1\_2(X,Y)$$

Note, however, that non-user predicates are not renamed, e.g.,

$$ren(M1 \; is \; M-1, \{2\}, \langle \{1\}, \{2\} \rangle) = M1 \; is \; M-1$$

The generation of residual rules proceeds as follows:

- We do not generate residual clauses associated to the application of rules (variant) nor (failure).
- For embedding steps of the form $\langle (A, \pi, \mu), \mathcal{Q}; memo \rangle \xrightarrow{e} \langle \mathcal{Q}; memo \rangle$ we produce a residual rule of the form $ren(A, \pi, \mu) \leftarrow A$. This means that some

atoms will not be closed but defined in terms of calls to the original predicates (and, thus, the clauses of the original program should be added to the residual program).

- For nonuser steps $\langle (A,\pi,\mu),\mathcal{Q};memo\rangle \xrightarrow{n} \langle \mathcal{Q};memo\rangle$, we do not generate residual rules since non-user calls are not renamed.
- For an unfolding step $\langle (A,\pi,\mu),\mathcal{Q};memo\rangle \xrightarrow{u}_\sigma \langle prop(\mathcal{Q}',true),\mathcal{Q};memo\cup \{(A,\pi,\mu)\}\rangle$, we produce a residual rule of the form

$$ren(A,\pi,\mu) \leftarrow ren(B_1,\pi_1,\mu_1),\ldots,ren(B_n,\pi_n,\mu_n).$$

where $prop(\mathcal{Q}') = ((B_1,\pi_1,\mu_1),\ldots,(B_n,\pi_n,\mu_n))$.

- Finally, for a parallel step $\langle (A,\pi,\mu),\mathcal{Q};memo\rangle \xrightarrow{p}_\sigma \langle \mathcal{Q}_1,\mathcal{Q}_2,\mathcal{Q}_3,\mathcal{Q}_4,\mathcal{Q};memo \cup\{(A,\pi,\mu)\}\rangle$, we produce a residual rule of the form

$$\begin{array}{rcl} ren(A,\pi,\mu) & \leftarrow & ren(B_1,\pi_1,\mu_1),\ldots,ren(B_n,\pi_n,\mu_n),\\ & & (ren(B_{n+1},\pi_{n+1},\mu_{n+1}),\ldots,ren(B_m,\pi_m,\mu_m)\\ & & \&\ ren(B_{m+1},\pi_{m+1},\mu_{m+1}),\ldots,ren(B_k,\pi_k,\mu_k)),\\ & & ren(B_{k+1},\pi_{k+1},\mu_{k+1}),\ldots,ren(B_l,\pi_l,\mu_l). \end{array}$$

where
$\mathcal{Q}_1 = ((B_1,\pi_1,\mu_1),\ldots,(B_n,\pi_n,\mu_n))$,
$\mathcal{Q}_2 = ((B_{n+1},\pi_{n+1},\mu_{n+1}),\ldots,(B_m,\pi_m,\mu_m))$,
$\mathcal{Q}_3 = ((B_{m+1},\pi_{m+1},\mu_{m+1}),\ldots,(B_k,\pi_k,\mu_k))$,
$\mathcal{Q}_4 = ((B_{k+1},\pi_{k+1},\mu_{k+1}),\ldots,(B_l,\pi_l,\mu_l))$.

Example 8
For instance, for the derivations of Example 7, we produce the following residual program:

$$\begin{array}{l} fibonacci\_1\_1\_2(0,1).\\ fibonacci\_1\_1\_2(1,1).\\ fibonacci\_1\_1\_2(M,N) \leftarrow M > 1,\ (M1\ is\ M-1, fibonacci\_1\_1\_2(M1,N1)\\ \qquad\qquad\qquad\qquad\qquad\qquad \&\ M2\ is\ M-2, fibonacci\_1\_1\_2(M2,N2)),\\ \qquad\qquad\qquad\qquad\quad N\ is\ N1+N2. \end{array}$$

2.4 Correctness and termination issues

The core of our new proposal mainly involves new control strategies, but the main procedure is still an instance of the standard partial evaluation framework, so its correctness should not be an issue. In particular, our partial evaluation scheme can be seen as an instance of the procedure of Benkerimi and Lloyd (1989), though in our case an atom is closed only if it is a variant (rather than an instance) of an already partially evaluated atom. Our approach is correct though since we add calls to the predicates of the original program for non-closed atoms (and the residual program includes a copy of the original program clauses).

Regarding the termination of partial evaluation, this is a well studied area and the approach that we consider based on the homeomorphic embedding ordering is quite standard (Leuschel 2002).

Regarding the introduction of parallel conjunctions, in this paper we assume the correctness of the underlying groundness and dependency analyses. Moreover, we prove in the online appendix the correctness of the few functions introduced to propagate groundness and sharing patterns at partial evaluation time, *entry* and *prop*. Of course, the correctness of function *partition* can only be ensured when $\mathcal{Q}'_2$ and $\mathcal{Q}'_3$ only contain user defined predicates or “safe” built-ins (i.e., built-ins without side effects, which do not depend on or may change the order of evaluation, etc).

To summarize, this paper is not concerned with the development of new theoretical developments regarding partial evaluation or program parallelization, but with the design of new control strategies that could allow us to improve existing partial evaluation techniques and use them to extract some implicit independent AND-parallelism of logic programs. Moreover, the proof-of-concept implementation of a parallelizing partial evaluator (that we discuss in the next section) shows that our approach is indeed viable in practice.

3 Experimental evaluation

A prototype implementation of the parallelizing partial evaluator described so far has been developed. It consists of approx. 1000 lines of SWI Prolog code (including the groundness call and success pattern analysis, comments, etc). The only missing component is the sharing analysis, which currently should be provided by the user. In general, built-in’s and extra-logical features are not unfolded, though our tool includes information regarding the propagation of groundness and sharing information for them.

A web interface to our tool is available at `http://kaz.dsic.upv.es/litep.html`.

We have tested it by running some typical benchmarks from the literature on automatic independent AND-parallelization of logic programs (see, e.g., (Muthukumar *et al.* 1999; Gras and Hermenegildo 2009)):

- **amatrix** implements the addition of two matrices (a matrix is a list of lists);
- **fib** computes the well-known Fibonacci function;
- **flatten** is used to flatten a list of lists of any nesting depth into a flat list;
- **hanoi** solves the Towers of Hanoi problem;
- **msort** implements the mergesort algorithm on lists;
- **mmatrix** implements the multiplication of two matrices;
- **palin** recognizes (list) palindromes;
- **qsort** implements the quicksort algorithm on lists;
- **tak** computes the Takeuchi function.

Moreover, in order to test the scalability of the tool, we have also applied our parallelizing partial evaluation tool to itself (**ppeval**). The code of the examples can be found in the tool’s webpage.

We use SWI Prolog’s `concurrent/3` to run goals in parallel. Parallel processes in SWI Prolog, however, are not lightweight. As mentioned in (SWI 2012), *if the goals are CPU intensive and normally all succeeding, typically the number of CPUs is the optimal number of threads. Less does not use all CPUs, more wastes time in*

Table 1. *Experimental evaluation of the parallelizing partial evaluator*

benchmark	Seq	Par1	Par2	Par4	Par6	Par8
fib	1.00	1.00	1.83	2.88	3.82	3.70
hanoi	1.00	1.27	1.54	2.29	2.05	1.97
mmatrix	1.00	1.05	1.07	1.09	1.08	1.07
palin	1.00	1.07	1.79	2.52	2.30	2.41
tak	1.00	0.98	1.31	1.31	1.30	1.31
amatrix	1.00	1.02	0.59	0.30	0.20	0.16
flatten	1.00	1.23	0.72	0.63	0.61	0.81
msort	1.00	1.59	0.86	1.23	1.22	1.26
qsort	1.00	1.73	0.48	0.71	0.72	0.60
ppeval	1.00	1.00	1.15	SO	SO	SO

context switches and also uses more memory. For instance, the unbound number of threads that would be created with the program of Example 8 would perform very badly for even small input values. In order to solve this problem, we replace calls to concurrent/3 by a special version as follows:

```
concurrent_k(A,B,C) :-
 current_threads(N), max_threads(K),!,
 (N < K -> M is N+1,
          retractall(current_threads(_)),assert(current_threads(M)),
          concurrent(2,[B,C],[]),
          current_threads(T), S is T-1,
          retractall(current_threads(_)),assert(current_threads(S)),
         ; call(A) ).
```

Basically, given queries $\mathcal{Q}_1$ and $\mathcal{Q}_2$, *concurrent_k*$((\mathcal{Q}_1,\mathcal{Q}_2),\mathcal{Q}'_1,\mathcal{Q}'_2)$ determines, depending on the current and maximum number of threads, if a sequential goal $(\mathcal{Q}_1,\mathcal{Q}_2)$ or a parallel goal $\mathcal{Q}'_1\&\mathcal{Q}'_2$ should be run (where $\mathcal{Q}'_i$ is the parallel version of $\mathcal{Q}_i$).

Table 1 summarizes our experimental results for the selected benchmarks. We executed SWI-Prolog (Multi-threaded, 64 bits, Version 6.0.2) on a 2.66 GHz Quad-Core Intel Xeon (with 8GB 1066 MHz DDR3 RAM) running Mac OS X v10.7.3. Therefore, one can expect the best results for a maximum of 4 threads. Run times have been obtained using SWI Prolog's get_time/1, which is similar to SICStus walltime and includes CPU time, garbage collection, etc. Rather than timings, we show the relative speedup (i.e., run time of the original program/run time of the residual program; values > 1 are then actual speedups) for each original program (column **Seq**), and its partially evaluated version using 1/2/4/6/8 cores (columns **Par1**/**Par2**/**Par4**/**Par6**/**Par8**). Here, SO indicates a *stack overflow*.

First, we observe that the values of column **Par1** are not always 1.00. This is due to the effects of the partial evaluation. We tried to minimize it, but it seems that for some examples it still has a significant effect. The first group of benchmarks (**fib**, **hanoi**, **mmatrix**, **palin** and **tak**) show the expected results: **Par1** is generally close to 1 and the introduction of parallel threads produces noticeable speedups.

For the second group of benchmarks (**amatrix**, **flatten**, **msort** and **qsort**), we get a slowdown in almost all cases but in **msort** (and, even in this case, the sequential partial evaluation is faster). Let us take a look at the results. For instance, for **amatrix**, we transform:

```
amatrix([L1|O1],[L2|O2],[L3|O3]):- am1(L1,L2,L3), amatrix(O1,O2,O3).
```

into

```
amatrix_par([A|B],[C|D],[E|F]) :-
                     concurrent_k((am1(A,C,E),amatrix(B,D,F)),
                                  am1(A,C,E), amatrix_par(B,D,F)).
```

and leave the rest of the program untouched. For **quicksort**, we get

```
quicksort_par([A|B],C):-partition(B,A,D,E),
                         concurrent_k((quicksort(D,F),quicksort(E,G)),
                                      quicksort_par(D,F),
                                      quicksort_par(E,G)),
                          append(F,A,G,C).
```

and the rest of the program is not modified. Similar results are obtained for **flatten** and **msort**. Note that the output of our tool is perfectly reasonable (i.e., it coincides with a typical parallelization by hand). So what explains the slowdowns produced? Besides the particularities of these benchmarks, it might be caused by the implemented model of parallel threads in SWI Prolog (which copies ground arguments instead of sharing them). Further investigating this point is a subject of ongoing research; e.g., we plan to test the benchmarks using a different Prolog environment supporting source-level primitives for AND-parallelism. As for the third group, **ppeval**, we do not get a significant speedup but it allows us to check that the approach is viable in practice and scales up well to medium programs (the stack overflow corresponds to running the specialized partial evaluator to partially evaluate itself on 4 or more threads, and seems to be related to the limited size of threads' stacks—i.e., it is not a fault of **ppeval**).

In summary, the experimental evaluation is still preliminary, but it clearly shows that there is a good potential for improving program performance by using a parallelizing partial evaluator. Indeed, one can easily judge by visual inspection of the annotated programs (check the results in `http://kaz.dsic.upv.es/litep.html`) that our parallelizing partial evaluator uncovers as much parallelism opportunities as it is possible. We have not compared our tool with any existing parallelizing compiler for logic programs yet. On the one hand, because our tool is not yet mature enough to deal with realistic Prolog applications. On the other hand, because we could not find a publicly available working system for source-level program parallelization.

4 Concluding remarks and future work

In this work, we have presented a novel approach to parallelizing partial evaluation. Analogously to standard approaches to automatic independent AND-parallelization of logic programs, our partial evaluator uses run-time groundness and dependency information. However, in contrast to these approaches, we can transform the source program in order to expose more opportunities for parallelization. We are not aware of any previous proposal along the same lines. (Consel and Danvy 1992; Sperber *et al.* 1997) considers performing *partial evaluation* in parallel, which is quite a different goal as ours. The closer approach we are aware of is that of (Surati and Berlin 1994), where a standard partial evaluator is used to expose some low level operations of a program so that a parallelization algorithm can be more successfully applied. They consider, however, two independent actions: standard partial evaluation and program parallelization, in contrast to ours. Nevertheless, the idea of combining partial evaluation and static analysis is not new (Jones 1997). Also, the use of partial evaluation to compile an instrumented interpreter can be used to enrich source programs with some additional information that can be useful for debugging or optimizing execution (see, e.g., (Debois 2004; Jones 2004)). Although we are not aware of using it for generating annotations for parallelism so far, partially evaluating an interpreter instrumented with groundness and sharing information (so that conjunctions are executed in parallel when safe) could get similar results as our approach.

Being a novel approach, we consider that there is plenty of room for further improvements. Firstly, one can consider the use of more accurate groundness and sharing analysis. Secondly, our partition procedure to extract two independent subgoals that can be run in parallel is rather simple. We plan to extend it to allow an arbitrary number of parallel subgoals, and also to allow the reordering of some subgoals. We would also like to explore other notions of AND-parallelism like non-strict independent AND-parallelism or, even, dependent AND-parallelism. Finally, the combination of our approach with a more aggressive partial evaluation scheme is also an interesting avenue for future work.

Acknowledgements

The author wants to thank the members of the SWI Prolog mailing list for being so helpful and responsive. We also gratefully acknowledge the anonymous referees for many useful comments and suggestions.

References

SWI Prolog 2012. URL: http://www.swi-prolog.org/.

Benkerimi, K. and Lloyd, J. 1989. *A Procedure for the Partial Evaluation of Logic Programs.* Technical Report TR-89-04, Department of Computer Science, University of Bristol, Bristol, England. May.

Consel, C. and Danvy, O. 1992. Partial evaluation in parallel. *Lisp and Symbolic Computation 5,* 4, 327–342.

De Schreye, D., Glück, R., Jørgensen, J., Leuschel, M., Martens, B. and Sørensen, M. 1999. Conjunctive partial deduction: foundations, control, algorihtms, and experiments. *Journal of Logic Programming 41,* 2/3, 231–277.

Debois, S. 2004. Imperative program optimization by partial evaluation. In *Proc. of PEPM'04*, N. Heintze and P. Sestoft, Eds. ACM, 113–122.

Debray, S. K. 1989. Static inference of modes and data dependencies in logic programs. *ACM Transactions Programming Languages and Systems 11,* 3, 418–450.

Gallagher, J. 1991. *A System for Specialising Logic Programs*. Technical Report TR-91-32, University of Bristol.

Gallagher, J. 1993. Tutorial on Specialisation of Logic Programs. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93)*. ACM, New York, 88–98.

Gras, D. C. and Hermenegildo, M. V. 2009. Non-strict independence-based program parallelization using sharing and freeness information. *Theoretical Computer Science 410,* 46, 4704–4723.

Gurr, C. 1994. *A Self-Applicable Partial Evaluator for the Logic Programming Language Goedel.* PhD Thesis, Department of Computer Science, University of Bristol.

Jones, N. D. 1997. Combining abstract interpretation and partial evaluation (brief overview). In *Proceedings of the 4th International Symposium on Static Analysis (SAS'97)*, P. V. Hentenryck, Ed. Lecture Notes in Computer Science, vol. 1302. Springer, 396–405.

Jones, N. D. 2004. Transformation by interpreter specialisation. *Science of Computer Programming 52*, 307–339.

Jones, N., Gomard, C. and Sestoft, P. 1993. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Englewood Cliffs, NJ.

Leuschel, M. 2002. Homeomorphic embedding for online termination of symbolic methods. In *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones.* Springer LNCS 2566, 379–403.

Leuschel, M., Elphick, D., Varea, M., Craig, S. and Fontaine, M. 2006. The ecce and logen partial evaluators and their web interfaces. In *Proceedings of PEPM'06.* IBM Press, 88–94.

Leuschel, M. and Vidal, G. 2009. Fast offline partial evaluation of large logic programs. In *Proceedings of the 18th Int'l Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2008)*. Springer LNCS 5438, 119–134.

Lloyd, J. and Shepherdson, J. 1991. Partial evaluation in logic programming. *Journal of Logic Programming 11*, 217–242.

Muthukumar, K., Bueno, F., de la Banda, M. J. G. and Hermenegildo, M. V. 1999. Automatic compile-time parallelization of logic programs for restricted, goal level, independent and parallelism. *Journal of Logic Programming 38,* 2, 165–218.

Sahlin, D. 1990. The mixtus approach to automatic partial evaluation of full prolog. In *Proceedings of the 1990 North American Conference on Logic Programming*, S. Debray and M. Hermenegildo, Eds. The MIT Press, Cambridge, MA, 377–398.

Sperber, M., Thiemann, P. and Klaeren, H. 1997. Distributed partial evaluation. In *Proceedings of the 2nd Int'l Workshop on Parallel Symbolic Computation (PASCO 1997)*, H. Hong, E. Kaltofen, and M. A. Hitz, Eds. ACM, 80–87.

Surati, R. J. and Berlin, A. A. 1994. Exploiting the parallelism exposed by partial evaluation. In *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques (PACT'94).* IFIP Transactions, vol. A-50. North-Holland, 181–192.

Vidal, G. 2011. A hybrid approach to conjunctive partial evaluation of logic programs. In *Logic-Based Program Synthesis and Transformation - 20th Int'l Symposium (LOPSTR 2010), Revised Selected Papers.* Springer LNCS 6564, 200–214.