Efficient and Flexible Access Control via Jones-Optimal Logic Program Specialisation *

Steve Barker (steve.barker@kcl.ac.uk)
Department of Computer Science, King's College, London, WC2R 2LS, UK

Michael Leuschel (leuschel@cs.uni-duesseldorf)
Institut für Informatik, Heinrich-Heine-Universität Düsseldorf, Universitätsstr. 1, D-40225 Düsseldorf †

Mauricio Varea (m.varea@ecs.soton.ac.uk)
School of Electronics and Computer Science, University of Southampton, Highfield, SO17 1BJ, UK

Abstract. We describe the use of a flexible meta-interpreter for performing access control checks on deductive databases. The meta-program is implemented in Prolog and takes as input a database and an access policy specification. For processing access control requests we specialise the meta-program for a given access policy and database by using the LOGEN partial evaluation system. The resulting specialised control checking program is dependent solely upon dynamic information that can only be known at the time of actual access request evaluation. In addition to describing our approach, we give a number of performance measures for our implementation of an access control checker. In particular, we show that by using our approach we get flexible access control with virtually no overhead, satisfying the Jones optimality criterion. The paper also shows how to satisfy the Jones optimality criterion more generally for interpreters written in the non-ground representation.

Keywords: Access Control, Deductive Databases, Partial Evaluation, Program Transformation, Meta-Programming

1. Introduction

The issue of controlling a user's ability to exercise access privileges (e.g., read, write, execute privileges) on a system's resources has long been important in Computer Science. With the advent of the Web and a move towards open, distributed systems, security has assumed even greater importance. In a number of surveys, security issues have been reported by enterprises as being of paramount concern when deciding policies on the publication of Web data, and the availability of Web resources (see, for example, [10]). Security issues, including access control issues, will be of particular importance in the emerging Seman-

* Work partially supported by European Framework 5 Project ASAP (IST-2001-38059). This paper is revised and extended version of [5], incorporating some material from [26].

tic Web, and for e-commerce and distributed business-rule processing applications (see, for example, [17]).

In recent years, a number of researchers have developed sophisticated access control models in which access control requirements are expressed by using rules that are employed in reasoning about authorised forms of access to resources. In these approaches (see, for example, [6], [8], and [19]), the requirements that must be satisfied in order to access resources are represented by using rules expressed in (C)LP languages. Expressing access control policies in (C)LP is natural, and enables many implicit permissions, denials and authorisations to be specified using declarative languages for which well defined semantics and operational methods with attractive theoretical properties (e.g., termination) are known to exist.

An important practical issue that arises with the rule-based approach to access control is the problem of efficiently evaluating access requests when access control requirements are implicitly specified. The problem of efficiently evaluating access requests with respect to rule-based specifications of access policies becomes increasingly important as organisations use ever more complex forms of access control policies. Goal evaluation with respect to complex forms of policy specifications is potentially expensive.

For each of the approaches described in [6], [8], and [19], proposals are made for attempting to ensure that access requests are evaluated efficiently when access control requirements are specified implicitly. In [8] and [19], *view materialisation* approaches (i.e., storing all of the valid authorisations as facts) are described for attempting to optimise access control checks. The motivation for view materialisation is to make explicit the access control information that is implicitly defined in rule form. Making explicit the implicitly specified access control information means that access requests can be evaluated by considering explicitly recorded facts rather than having to derive these facts at query evaluation time. Unfortunately, view materialisation is not necessarily appropriate to use when large numbers of *parametric derivation rules* [7] are used to express access control requirements and when the specification of access control requirements changes dynamically. Dynamic changes arise, for example, when *user session information* [6] is used in the course of deciding whether an access request is authorised or when access to resources depends on satisfying temporal constraints.

In contrast to previous work, we describe an approach, based on partial evaluation, for addressing the problem of efficiently evaluating access requests where large numbers of parametric derivation rules are used to specify access policy requirements; where fine-grained access to data items is required (e.g., access to atomic formulae); where the an-

swer to a user's access request generates the information in a database that the user is permitted to see;[1] and where the static access control information is to be exploited for performance gains.

In overview, we describe an access control checker that is implemented by using a meta-program that is written as a logic program. The meta-program takes as input an access control program and a database. The approach enables the meta-interpreter that we introduce, and hence our access control checker, to be specialised in order to reduce the amount of information that needs to be considered at run time to satisfy a user's access request. In effect, the approach ensures that a minimal amount of information is considered at access request evaluation time. Specifically, the user session information that applies at the time of an access request is used with a form of access control program that is specialised by using the relatively static information that is specified as part of the access control program (e.g, in the examples in this paper we assume that the user, role and hierarchy information is static relative to the information about which users and resources are active).

Although meta-interpreters have previously been developed for efficient constraint checking on databases by Leuschel and De Schreye [29] (using a prototype partial evaluator), to the best of our knowledge, no approach has yet been proposed in the literature for generating specialised access requests via a meta-interpreter that manipulates access requests, access control policies and databases as object level expressions, and that pre-compiles access checking for certain access requests. In this paper, we describe a technique to obtain a specialised access control checker that is more efficient to use than using a database and access control program directly because some of the propagation, simplification and evaluation process is pre-compiled. To this end we present a technique that makes it possible to obtain "Jones optimal" specialisation [20, 21, 31] for a class of meta-interpreters.

In our approach, we consider the use, by *security administrators*, of *role-based access control (RBAC)* policies [6] for specifying authorised forms of access to database objects in a non-distributed environment. In *RBAC*, the most fundamental notion is that of a role. A role is defined in terms of a job function in an organisation (e.g., a *doctor* role in a medical environment); users are assigned to roles. Moreover, access privileges on objects (i.e., *permissions*) are assigned to roles (e.g., the permission to change a patient's prescriptions may be assigned to the role *doctor*). *RBAC* policies have a number of well documented

[1] We restrict attention to retrievals of information in this paper. However, any kind of operation on information can be accommodated by our approach.

advantages [39], and are widely used in practice (see, for example, [13, 40, 35]). Although we restrict our attention to *RBAC* policies in this paper, it should be noted that *RBAC* is a more general form of access control model than the *discretionary access control* and *mandatory access control* approaches that predate *RBAC* [12], and the approach that we describe can be used with more powerful access control methods than *RBAC* (e.g., the access control model described in [4]). It follows that our approach is widely applicable.

The rest of this paper is organised as follows. In Section 2, some background information is provided on the partial evaluation of logic programs, in general, and the LOGEN system, in particular. In Section 3, we present an approach to successfully specialise a class of meta-interpreters, for which we achieve Jones optimality, thereby laying the foundation for optimising access control meta-interpreters. In Section 4, we briefly describe an *RBAC* model, as well as the formulation of *RBAC* policies by using logic programs. In Section 5, we develop an access control meta-interpreter for the evaluation of access requests on databases with respect to *RBAC* policies, and show how the meta-interpreter can be specialised by using LOGEN. In Section 6, we present and discuss performance measures of our approach. Finally, Section 7 concludes the work and suggests future work.

2. Partial Evaluation and the LOGEN System

Partial evaluation [21] is a source-to-source program transformation technique that specialises programs by fixing part of the input of some source program P and then pre-computing those parts of P that only depend on the fixed part of the input. The so-obtained transformed programs are less general than the original, but can be much more efficient. The part of the input to P that is fixed is referred to as the *static* input, while the remainder of the input is called the *dynamic* input.

2.1. Partial Evaluation of Logic Programs

We now describe the process of partial evaluation of logic programs. For logic programs we follow the notational conventions of [30]. In particular, in programs, we denote variables by strings starting with an upper-case symbol, while the notations for constants, functions and predicates begin with a lower-case character. More details about logic programming can be found, e.g., in [30].

Formally, executing a logic program P for an atom A consists of building a so-called *SLD-tree* for $P \cup \{\leftarrow A\}$. Take for example the well-known append program:

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

The SLD-tree for `append([a,b],[c],R)` is presented on the left in Figure 1.[2] The underlined atoms are called selected atoms and □ represents the empty goal. The edges are labelled with the *most general unifiers (mgus)* between the selected atom and the head of a program clause. A branch leading to the empty goal is called *successful* and gives rise to a *computed answer substitution* obtained by composing the mgus on the branch and restricting it to the variables in the top-level goal. Here there is only one successful branch, and its computed answer substitution is `R = [a,b,c]`.

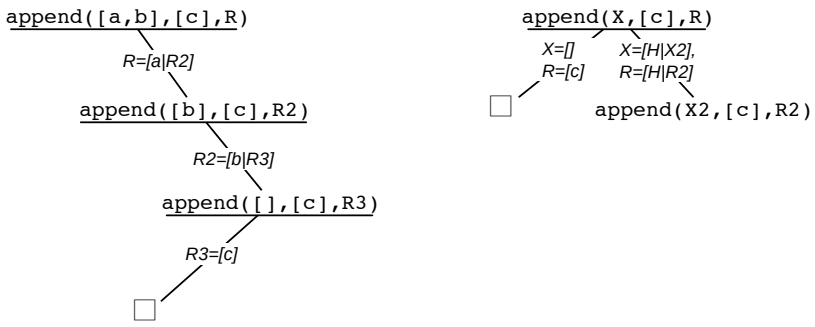

Figure 1. Complete and Incomplete SLD-trees for the `append` program

Partial evaluation for logic programs builds upon this with two important differences:

– At some step in building the SLD-tree, it is possible *not* to select an atom, hence leaving a leaf with a non-empty goal. The motivation is that lack of the full input may cause the SLD-tree to have extra branches, in particular infinite ones. For example, in Figure 1 the rightmost tree is an incomplete SLD-tree for `append(X,[c],R)`, whose full SLD-tree would be infinite. Building such a possibly incomplete tree is called *unfolding*. An *unfolding rule* tells us which atom to select at which point and when not to select an atom. Observe that incomplete branches do not produce computed answers;

[2] In this case there is only a single SLD-tree for the particular goal.

they produce conditional answers that can be expressed as program clauses by taking the resultants of the branches as defined further below.

– Because of the atoms left in the leaves (which will appear in the bodies of the resultants), we may have to build a series of SLD-trees to ensure that every such atom is covered by some root of some tree. The condition that every leaf is an instance of a root is called the *closedness* (sometimes also *coveredness*) condition. In the example of Figure 1, the leaf atom `append(X2,[c],R2)` is already an instance of its root atom; hence, closedness holds and there is no need to build more trees.

DEFINITION 1. *Let P be a program, $G = \leftarrow Q$ a goal, D a finite SLD-derivation of $P \cup \{G\}$ ending in $\leftarrow B$, and θ the composition of the* mgu*s in the derivation steps. Then the formula $Q\theta \leftarrow B$ is called the* **resultant** *of D.*

For example, the resultants of the derivations in the right tree of Figure 1 are:

```
append([],[c],[c]).
append([H|X2],[c],[H|R2]) :- append(X2,[c],R2).
```

Partial evaluation starts from an initial set of atoms $\mathcal{A}$ provided by the user that is chosen in such a way that all runtime queries of interest are *covered* by $\mathcal{A}$, meaning that all atoms occurring inside runtime queries are an instance of some atom in $\mathcal{A}$. As we have seen, constructing a specialised program requires us to build an SLD-tree for each atom in $\mathcal{A}$. Moreover, one can easily imagine that ensuring closedness may require revision of the set $\mathcal{A}$, i.e., adding new atoms to $\mathcal{A}$ or replacing existing atoms by a more general one. Hence, when controlling partial evaluation, it is natural to separate the control into two components (as already pointed out in [15, 34]):

– The *local control* guides the construction of the finite SLD-tree for each atom in $\mathcal{A}$ and thus determines *what* the residual clauses for the atoms in $\mathcal{A}$ are.

– The *global control* determines the content of $\mathcal{A}$, it decides *which* atoms are ultimately unfolded (taking care that $\mathcal{A}$ remains closed for the initial atoms provided by the user).

In that context, one also talks about *local termination*, i.e., making sure that all the SLD-trees are finite, and about *global termination*,

i.e., making sure that the construction of the set $\mathcal{A}$ terminates. More details on partial evaluation for logic programs and how to control it can be found, e.g., in [25].

2.2. Offline Partial Evaluation

The control of partial evaluation can be broadly classified into *online* and *offline* approaches. Online partial evaluators have a single specialisation phase and take all their control decisions during specialisation. Offline partial evaluation (see, e.g., [21]) is divided into two phases, as depicted in Figure 2:

- First a *binding-time analysis* (BTA) is performed which, given a program and an approximation of the input available for specialisation, approximates all values within the program and generates annotations that steer (or control) the specialisation process.
- A (simplified) *specialisation phase*, which is guided by the result of the BTA.

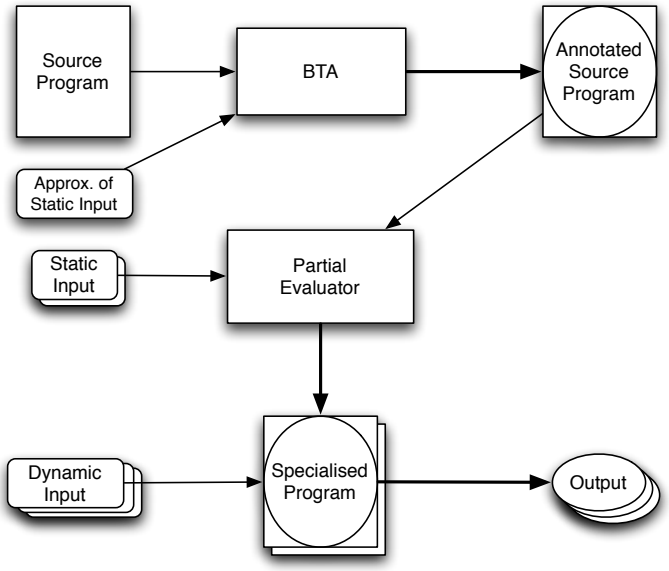

Figure 2. Offline Partial Evaluation

Because of the preliminary BTA, the specialisation process itself can be performed very efficiently and with predictable results.

2.3. The logen System

The logen system [27] is an *offline* partial evaluator for Prolog. Also, as we will show in Section 3, the logen system is well suited to specialise interpreters, something that we will aim to exploit later in the paper for efficient access control.

As noted above, an offline specialiser works on an annotated version of the source program. logen uses two kinds of annotations:

- *Filter declarations*, which declare which arguments to which predicates are static and which ones are dynamic. This influences the global control (only): dynamic arguments are always replaced by a variable before adding an atom to $\mathcal{A}$, while static arguments are kept as they are.
- *Clause annotations*, which indicate for every call in the body how that call should be treated during unfolding. This thus influences the local control only, which is effectively hard-wired. For now, we assume that a call is either annotated with **memo** — indicating that it should not be unfolded – or with **unfold** — indicating that it should be unfolded. We introduce more annotations later on.

There is, of course, an interplay between these two kinds of annotations; we will return to this issue in the discussion below.

First, let us consider an example of an annotated version of the unspecialised `append` program from the beginning of Section 2.1:

```
:- filter append(dynamic,static,dynamic).
append([],L,L).
append([H|X],Y,[H|Z]):- append(X,Y,Z).
                         memo
```

In this example, the filter declarations annotate the second argument to `append` as static while the others are marked dynamic and the clause annotations annotate the recursive call in the second clause as **memo**. Given such annotations and a specialisation query `append(X,[c],Z)`, the logen system would unfold exactly as depicted in the right tree of Figure 1 and would produce the resultants above.

The following is a general algorithm for offline partial evaluation given filter declarations and clause annotations.

Algorithm 2.1 (offline partial evaluation)
Input: A program P and an atom A

$\mathcal{A} = \{A\}$
repeat
select an unmarked atom A in $\mathcal{A}$ and mark it

build an SLD-tree τ_A for A using the clause annotations in the annotated source program: A, as well as all literals marked as **unfold** are unfolded; literals marked as **memo** are not unfolded
for every leaf atom S of τ_A that was annotated as **memo**
do
generalise S into S' by replacing all arguments declared as dynamic by the filter declarations with a fresh variable
if no variant of S' is in $\mathcal{A}$ **then** add it to $\mathcal{A}$ **end if**
end do
pretty print the resultants of τ_A
until all atoms in $\mathcal{A}$ are marked.

In practice, renaming transformations [16] are also involved: Every atom in $\mathcal{A}$ is assigned a new predicate name, whose arity is the number of arguments declared as dynamic (static arguments do not need to be passed around; they have already been built into the specialised code). For example, the resultants of the derivations in the right tree of Figure 1 would get transformed into the following, where the static argument has been removed:

```
append__0([],[c]).
append__0([H|X2],[H|R2]) :- append__0(X2,R2).
```

The full treatment in LOGEN is a lot more complicated as LOGEN supports a more user-friendly syntax as well as various features, some of which are introduced in the sections that follow.

3. Specialisation of Interpreters and Jones Optimality

Partial evaluation is especially useful when applied to interpreters. In that setting, the static input is typically the object program being interpreted, while the actual call to the object program is dynamic. Partial evaluation can then produce a more efficient, specialised version of the interpreter, which is sometimes akin to a compiled version of the object program [14]. Some successful applications in that area are, e.g., [9] where Bondorf and Palsberg compile action semantic into competitive (Scheme) code. More examples and references can be found, e.g., in [21].

The ultimate goal when specialising interpreters is to achieve *Jones optimality* [20, 21, 31], i.e., fully getting rid of a layer of interpretation (called the "optimality criterion" in [21]). More precisely, suppose we have a self-interpreter `sint` for a programming language L, i.e., an interpreter for L written in that same language L, and then specialise

`sint` for a particular object program `p`; what we would like to obtain is a specialised interpreter `p'` that is at least as efficient as `p` (see Figure 3). One uses a self-interpreter, rather than an interpreter in general, to be able to compare directly the running times of `p` and `p'` (as they are written in the same programming language L).

More formally, if D is the input domain of `p` and $t_p(\mathtt{i})$ is the running time of the program p on the input `i`, we require that $\forall d \in D : t_{\mathtt{p}'}(\mathtt{d}) \leq t_{\mathtt{p}}(\mathtt{d})$.

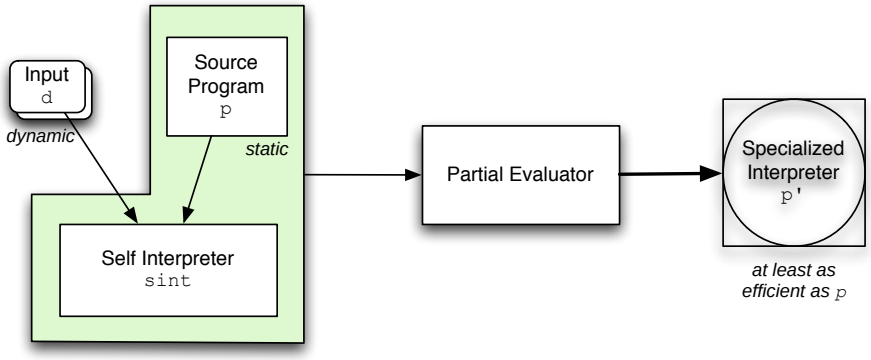

Figure 3. Jones Optimality

In this section, we will show how to achieve Jones-optimality for the classical "vanilla" self-interpreter for logic programs [18, 1] . In the following section, we will extend this interpreter into an interpreter for access control; Jones optimality will ensure that we do not pay a performance penalty if no access control is needed. First though, we present a very simple interpreter and show how it can be specialised using LOGEN.

3.1. Propositional Logic Interpreter

We first introduce a simple propositional logic interpreter to demonstrate the basic annotations of LOGEN. The interpreter handles the connectives *and*, *or*, *true*, *false* and propositional variables. The predicate `int/2` takes two arguments: the propositional formula and the environment containing the list of all propositional variables that are true. The predicate succeeds if the formula evaluates to true given the environment.

```
int(true,_).
int(and(X,Y),Env) :- int(X,Env), int(Y,Env).
```

```
int(or(X,_Y),Env) :- int(X,Env).
int(or(_X,Y),Env) :- int(Y,Env).
int(var(X),Env) :- member(X,Env).

member(X,[X|_]).
member(X,[_|T]) :- member(X,T).
```

As was indicated in Figure 2, the source program that serves as input for LOGEN has to be annotated using filter declarations for the global control and clause annotations for the local control. The **filter** declarations describe the arguments of residual predicates to the specialiser. Top level predicates that one intends to specialise must be declared in this way, as well as any subsidiary predicate that cannot be fully unfolded. For example, for the above program we could declare:

```
:- filter int(static, dynamic).
:- filter member(dynamic, dynamic).
```

In other words, we assume that the propositional formula (the first argument of `int/2`) is known at specialisation time (**static**) while the environment will only be known at run time (**dynamic**). As the propositional formula is known at specialisation time (**static**) all calls to `int/2` can be unfolded in the clause annotations. As concerns the variable lookups in the environment, these cannot be fully unfolded and hence we have to mark the call to `member` as a **memo**:

```
int(var(X),Env) :- member(X,Env).
```
memo

Let us now specialise the interpreter for the logical formula: $((var(a) \wedge var(b)) \vee false) \wedge true$. The output from specialisation is a new version of the program, which just checks that $var(a)$ and $var(b)$ are both true in the environment. Observe that `member__1/2` is a specialised version of `member/2`; however, the specialised version is simply a renaming of the original as all its arguments were declared as dynamic:

```
int__0(A) :-  member__1(a,A), member__1(b,A).

member__1(A,[A|_]).
member__1(A,[_|B]) :- member__1(A,B).
```

3.2. The Vanilla Self-Interpreter

The interpreter that we described above was very well suited to specialisation because the propositional formula was ground (i.e., contained no free variables). This "ground representation" enabled us to mark arguments as **static** and to use LOGEN in straightforward way. However, many interpreters in Prolog use a *non-ground* representation [1, 18], in order to reuse the efficient Prolog unification mechanism for object level expressions. The classical example is the *vanilla self-interpreter* (see, e.g., [1, 18]). This interpreter is a self-interpreter because it can handle the language in which it is written. The following is the vanilla self-interpreter, along with an encoding of the double-append object program (which concatenates three lists):

```
solve(empty).
solve(and(A,B)) :- solve(A), solve(B).
solve(X) :- clause(X,Y), solve(Y).

clause(dapp(X,Y,Z,R),and(app(Y,Z,YZ),app(X,YZ,R))).
clause(app([],L,L),empty).
clause(app([H|X],Y,[H|Z]),app(X,Y,Z)).
```

The `clause/2` facts describe the object program to be interpreted, while `solve/1` is the self-interpreter executing the object program. In practice, `solve` will often be instrumented so as to provide extra functionality for, for instance, debugging, analysis (e.g., using abstract unifications instead of concrete unification) or transformation. We will actually use an instrumented form of `solve/1` later in this paper to implement our access control strategies. However, even without these extensions the vanilla interpreter provides enough challenges for partial evaluation. Indeed, we would like to achieve Jones optimality, i.e., specialise the interpreter so as to obtain a residual program that is at least as efficient as the object program being interpreted. For example, one would like to specialise our vanilla interpreter for the query `solve(dapp(X,Y,Z,R))` and obtain a specialised interpreter that is at least as efficient as:

```
dapp(X,Y,Z,R) :- app(Y,Z,YZ), app(X,YZ,R).
app([],L,L).
app([H|X],Y,[H|Z]) :- app(X,Y,Z).
```

Online partial evaluators such as ECCE [28] or MIXTUS [38] come close to achieving Jones-optimality for many object programs (and are fully automatic, requiring no annotation). However, they will not do

so for *all* object programs. We refer the reader to [33] (discussing the parsing problem) and the more recent [41] and [24] for more details. In [41], Vanhoof and Martens present a particular specialisation technique that can achieve Jones-optimality for the vanilla interpreter, but the technique is very specific to that interpreter and, as far as we understand, does not scale to extensions of it (such as our access control interpreter, which we will present later in the paper).

In the rest of this section, we show how LOGEN *can* achieve Jones-optimality for the vanilla interpreter.

3.3. The Nonvar Binding-Time Annotation

First, we have to present a new feature of LOGEN, which is essential when specialising interpreters that use the non-ground representation. In addition to marking arguments to predicates as static or dynamic, LOGEN now also supports the annotation **nonvar**. This means that the argument is not necessarily ground but has at least a top-level function symbol at specialisation time. When generalising the call, LOGEN keeps the top-level function symbol while replacing all its sub-arguments by fresh variables. Finally, these subarguments become arguments in the specialised version constructed by LOGEN.

A small example will help to illustrate this annotation:

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

Marking every call as **memo** (hence no unfolding), we obtain the following specialised program for the call `p(f(Z,Z))`. The commented lines indicate the renamings that LOGEN has performed.

```
/* p(f(A,B)) :- p__0(A,B).  p(g(A)):-p__1(A).  */
/*  p(h(A)):-p__2(A). */
p__0(A,A) :-  p__1(a).
p__1(A) :-    p__2(A).
p__2(a).
p__2(A) :-    p__0(A,A).
```

If we mark the last call in the original source program as **memo** and all others as **unfold**, we obtain:

```
/* p(f(A,B)) :- p__0(A,B). */
p__0(A,A).
p__0(A,A) :- p__0(a,a).
```

3.4. Jones-Optimality for Vanilla

The vanilla interpreter, as shown earlier, is actually a badly written program as it mixes the control structures `and` and `empty` with the actual calls to predicates of the object program. This means that the vanilla interpreter will not behave correctly if the object program contains predicates `and/2` or `empty/0`. This fact also poses problems typing the program. Even more importantly for us, it also prevents one from annotating the program effectively for LOGEN. Indeed, statically there is no way to know whether any of the three recursive calls to `solve/1` has a control structure or a user call as its argument. For LOGEN, this means that we can only mark the call `clause(X,Y)` as **unfold**. Marking any of the `solve/1` calls as **unfold** may lead to non-termination of the specialisation process. This also means that we cannot mark the argument to `solve/1` as **nonvar**, as it may actually become a variable. Indeed, take the call `solve(and(p,q))`; this call will be generalised into `solve(and(X,Y))` and after unfolding with the second clause we get the calls `solve(X)` and `solve(Y)`. Hence, we obtain very little specialisation and Jones-optimality is not achieved.

Two ways to solve this problem are as follows:

- Assume that the control structures are used in a principled, predictable way that will allow us to produce a better annotation.
- Rewrite the interpreter so that it is clearly typed, allowing us to produce an effective annotation as well as solving the problem with the name clashes between object program and control structures.

We will pursue these solutions in the remainder of this section.

3.4.1. *Structuring conjunctions.*

The first solution is to enforce a standard way of writing down conjunctions within `clause/2` facts by requesting that every conjunction is either `empty` or is an `and` whose left part is an atom and whose right part is a conjunction. For the example above, this means that we have to rewrite the `clause/2` facts as follows:

```
clause(dapp(X,Y,Z,R),
       and(app(Y,Z,YZ),and(app(X,YZ,R),empty))).
clause(app([],L,L),empty).
clause(app([H|X],Y,[H|Z]), and(app(X,Y,Z),empty)).
```

This allows us to predict the contents of a conjunction and thus to annotate the interpreter more effectively, without risking non-termination:

```
:- filter solve(nonvar).
solve(empty).
```

```
solve(and(A,B)) :- solve(A), solve(B).
                   memo      unfold
solve(X) :- clause(X,Y), solve(Y).
            unfold       unfold
```

Given our assumption about the structure of conjunctions, the above annotation will ensure termination of specialisation:

- **Local termination**: The call to `clause(X,Y)` can be unfolded as before as `clause/2` is defined by facts. The calls `solve(B)` and `solve(Y)` can be unfolded as we know that `B` and `Y` are conjunctions. LOGEN will deconstruct the `and/2` and `empty/0` function symbols. However, as `solve(A)` is marked **memo**, the possibly recursive predicates of the object program are not unfolded.

- **Global termination**: At the point when we memo `solve(A)`, the variable `A` will be bound to a predicate call. As we have marked the argument to `solve/1` as **nonvar**, generalisation will just keep the top-level predicate symbol. As there are only finitely many predicate symbols in the object program, global termination is ensured.

Specialising `solve(dapp(X,Y,Z,R))` now gives a Jones-optimal output:

```
/* solve(dapp(A,B,C,D)) :- solve__0(A,B,C,D). */
/* solve(app(A,B,C)) :- solve__1(A,B,C). */
solve__0(B,C,D,E) :-  solve__1(C,D,F), solve__1(B,F,E).
solve__1([],B,B).
solve__1([B|C],D,[B|E]) :-  solve__1(C,D,E).
```

LOGEN will, in general, produce a specialised program that is slightly better than the original program in the sense that it will generate code only for those predicates that are reachable in the *predicate dependency graph* [22] from the initial call. For example, for `solve(app(X,Y,R))` only two clauses for `app/3` will be produced, but no clause for `dapp/4`.

It is relatively easy to see that Jones optimality will be achieved for any properly encoded object program and any call to the object program. Indeed, any call of the form `solve(p(`$t_1, \ldots, t_n$`))` will be generalised into `solve(p(_,...,_))` keeping information about the predicate being called; unfolding this will only match the clauses of `p` as the call `clause(X,Y)` is marked **unfold** and all of the parsing structure (`and/2` and `empty/0`) will then be removed by further unfolding, leaving only predicate calls to be memoised. These are then generalised and specialised in the same manner.

3.4.2. *Rewriting Vanilla.*
The more principled solution is to rewrite the vanilla interpreter, so that the control structures and the object level atoms are clearly separated. The attentive reader may have noticed that above we have actually enforced that conjunctions are stored as lists, with `empty/0` playing the role of `nil/0` and `and/2` playing the role of ./2. The following vanilla interpreter makes this explicit and thus properly enforces this encoding. It is also more efficient, as it no longer attempts to find definitions of `empty` and `and` within the `clause` facts.

```
l_solve([]).
l_solve([H|T]) :- solve_atom(H), l_solve(T).
solve_atom(H) :- clause(H,Bdy), l_solve(Bdy).

clause(dapp(X,Y,Z,R), [app(Y,Z,YZ), app(X,YZ,R)]).
clause(app([],R,R), []).
clause(app([H|X],Y,[H|Z]), [app(X,Y,Z)]).
```

We can now annotate all calls to `l_solve` as **unfold**, knowing that this will only deconstruct the conjunction represented as a list. However, the call to `solve_atom` cannot be unfolded, as this would lead to infinite unfolding for a recursive object program. LOGEN now produces the following specialised program for the `solve_atom(dapp(X,Y,Z,R))` query, having marked the argument to `solve_atom` calls as **nonvar**[3].

```
solve_atom__0(B,C,D,E) :- solve_atom__1(C,D,F),
                          solve_atom__1(B,F,E).
solve_atom__1([],B,B).
solve_atom__1([B|C],D,[B|E]) :- solve_atom__1(C,D,E).
```

We have again achieved Jones optimality, which holds for any object program and any object-level query.

3.4.3. *Reflections.*
We now summarise the key features that enabled us to achieve Jones optimality:

– First, the offline approach allows us to precisely steer the specialisation process in a predictable manner: we know exactly how the interpreter will be specialised independently of the complexity of the object program. A problem with online techniques is that they may work well for some object programs, but can then be

[3] The predicate `l_solve` does not have to be given a filter declaration as it is only unfolded and never residualised.

"fooled" by other (more or less contrived) object programs (see [24] and [41]). On the other hand, online techniques are capable of removing several layers of self-interpretation in one go. An offline approach will typically only be able to remove one layer at a time.

- Second, it was also important to have sufficiently refined annotations at our disposal. Without the **nonvar** annotation, we would not have been able to specialise the original vanilla self-interpreter: we cannot mark the argument to `solve` as static, and marking it as dynamic means that no specialisation will occur. Hence, considerable rewriting of the interpreter would have been required if we just had **static** and **dynamic** at our disposal.
- Third, it is important that the self-interpreter is written in such a way that the specialiser can distinguish between conjunctions and object level calls and can treat them differently.

It turns out that our approach bears similarity to the approach presented by Lakhotia and Sterling in [23]. Indeed, in [23] the authors present a very simple partial evaluator guided by user annotations, along with the annotations required to successfully specialise simple interpreters. However, the scheme presented in [23] only deals with the local control of partial evaluation (completely ignoring the problem of global control, which was at the time less well understood) and also cannot deal with side-effects or other Prolog built-ins (such as if-then-else).

3.5. Adding negation and built-ins

Previously, we have shown how to extend the Jones-optimality result above to more sophisticated interpreters, e.g., to a debugging interpreter where speedups exceeded factors of 25 [26]. It was also shown that LOGEN produced a Jones-optimal result for every object program P and call C, provided that none of the predicates reachable from C are debugged. In other words, we pay no performance penalty if we do not use the debugging feature of the interpreter. If we do debug predicates, however, then we obtain an efficient version of the interpreter, where the debugging has been woven into the object program.

In this paper, we will elaborate on another extension of the above interpreter, where we add support for negations and built-ins. This will pave the way for a more challenging extension of the interpreter: implementing sophisticated access control strategies. For this we add clauses to handle negation and the built-in predicates respectively, resulting in the following extended interpreter (for simplicity we have only handled two built-ins; the extension to more built-ins is obvious):

```
l_solve([]).
l_solve([H|T]) :- solve_literal(H), l_solve(T).

solve_literal(not(H)) :- \+ solve_literal(H).
solve_literal(H) :- (user_predicate(H)->solve_atom(H);fail).
solve_literal(C) :- built_in(C).

solve_atom(H) :- my_clause(H,Bdy), l_solve(Bdy).

built_in('\='(X,Y)) :- X\=Y.
built_in(is(X,Y)) :- X is Y.
user_predicate(Call) :-
  my_clause(H,_),functor(H,F,N),functor(Call,F,N).
```

We can now obtain Jones-optimal specialisation by annotating all calls to `solve_literal/1` and `my_clause/2` as **unfold** and the call to `solve_atom/1` as **memo**. To be able to deal with built-ins, we need to use the annotations **call** and **rescall** from LOGEN whose meaning is as follows: a call annotated as **call** is completely evaluated; while a call annotated as **rescall** is added to the residual code without modification (for built-ins that cannot be evaluated). In our case, the calls to `\=/2` and `is/2` have to be marked **rescall**, while the calls to `functor/3` have to be marked **call**. As before, we have the following filter declaration:

```
:- filter solve_literal(nonvar).
```

If we now add the fact,

```
my_clause(not_abc(X),[not(app(_,[X|_],[a,b,c]))]),
```

we can specialise, e.g., `solve_atom(not_abc(A))`, thereby producing the following Jones-optimal result:

```
/* solve_atom(not_abc(A)) :- solve_atom__1(A). */
solve_atom__0(A) :-
        \+solve_atom__1([a,b,c],[A|_],_).

/* solve_atom(app(A,B,C)) :- solve_atom__1(C,B,A). */
solve_atom__1(A,A,[]).
solve_atom__1([B|C],A,[B|D]) :- solve_atom__1(C,A,D).
```

Later in the paper, we will further expand this interpreter, basically by integrating access control checks into `user_predicate`.

4. RBAC Policies as Logic Programs

We now move to the task of specialising access control policies. For this, we begin by describing some key notions in logic programming that are important in our approach. We then show how our RBAC policies may be represented as logic programs. In Section 5, we describe how access requests may be efficiently evaluated, with respect to our RBAC policies.

4.1. Preliminaries

We define an *RBAC* model and *RBAC* policies in the language of (function-free) *normal clause form logic* [2] (i.e., the language of *Datalog* with negation), with certain predicates in the alphabet Σ of the language having a fixed intended interpretation. As we only admit function-free clauses, the only terms of relevance to Σ will be constants and variables. As stated earlier, we denote variables that appear in clauses by using symbols that appear in upper case (at least the first character); constants will be denoted by lower case symbols.

A *normal clause* is a formula of the form:[4]

$$C \leftarrow A_1, \ldots, A_m, \neg B_1, \ldots, \neg B_n \qquad (m \geq 0, n \geq 0).$$

The *head*, C, of the normal clause above is a single *atom*. The *body* of the clause (i.e., $A_1, \ldots, A_m$, $\neg B_1, \ldots, \neg B_n$) is a conjunction of literals. Each A_i literal ($1 \leqslant i \leqslant m$) is a *positive literal*; each $\neg B_j$ literal ($1 \leqslant j \leqslant n$) is a *negative literal*. In the case of a negative literal, the relevant type of negation is *negation as failure* [11]. A clause with an empty body is an *assertion* or a *fact*. A clause with a non-empty head and a non-empty body is a *rule*. A *relational database* is a (finite) set of facts; a normal *deductive database* is a (finite) set of normal clauses; a normal deductive database that includes no negative literal is a *definite database*. The set of facts in a deductive database Δ is referred to as the *extensional* part of Δ (or the EDB of Δ) and the set of rules in Δ is referred to as the *intensional* part of Δ (or the IDB of Δ).

The access control programs that we consider are always *locally stratified* (a realistic assumption for most practical policies) and hence have a unique *perfect model* [36], which can be computed in PTIME. Having a 2-valued model theoretic semantics is important for ensuring that authorised forms of access are unambiguously specified.

[4] Observe that we assume associativity and commutativity of conjunction.

4.2. RBAC Policies as Logic Programs

In this section, we describe a simple type of $RBAC$ policy that could be used by a security administrator to protect the information in deductive databases. More specifically, the type of policy that we describe is based on the $RBAC^P_{H2A}$ model that is formally defined in [6] and which we describe in more detail below. We only consider one type of access control policy in this paper because our principal concern is to describe the generalities of using a meta-programming approach for access request checking and access policy optimisation by partial evaluation. It should be noted, however, that any of the policies from [6] may be represented by using our meta-programming approach with minor modifications.

In RBAC in general, a user may be assigned to any number of roles; permissions are assigned to a role r to permit those users that are assigned to r to exercise the privileges on objects that are assigned to the same r. The capability of assigning users to roles and permissions to roles are primitive requirements of all RBAC models. The most basic category of RBAC model, $RBAC_F$ or flat RBAC [6], requires that these types of assignment are supported. The $RBAC^P_{H2A}$ model extends $RBAC_F$ to include the notion of an RBAC role hierarchy (see below) in addition to user-role and permission-role assignments. Note that the H in $RBAC^P_{H2A}$ denotes that role hierarchies are permitted, the $2A$ denotes that the $RBAC^P_{H2A}$ model is at the "second level" in the family of RBAC models [6] and permits RBAC hierarchies of arbitrary complexity,[5] and the P in $RBAC^P_{H2A}$ denotes that permission assignments are admitted, but not denial assignments.

Formally, an $RBAC^P_{H2A}$ program is a finite set of normal clauses (see Section 4.1) specified with respect to a domain of discourse that includes:

- A set $\mathcal{U}$ of *users*.
- A set $\mathcal{O}$ of *objects*.
- A set $\mathcal{A}$ of *access privileges*.
- A set $\mathcal{R}$ of *roles*.

An $RBAC^P_{H2A}$ program includes a set Φ of logical axioms, a set of normal clauses that define the axioms that must be included in every $RBAC^P_{H2A}$ program. A security administrator adds a set of additional non-logical axioms Ψ to Φ to represent application-specific requirements. The normal clauses in $\Psi \cup \Phi$ are described below.

[5] In contrast, the $RBAC^P_{H2B}$ models only allow restricted forms of role hierarchy to be defined.

In an $RBAC^P_{H2A}$ program, a user is specified as being assigned to a role by using definitions of a `ura/2` predicate (where `ura` is short for "user role assignment"). The assignment of an access privilege on an object to a role is expressed by using definitions of a `pra/3` predicate in the $RBAC^P_{H2A}$ program (where `pra` is short for "permission role assignment"). The semantics of these predicates in an arbitrary $RBAC^P_{H2A}$ program Π, where Π is the pair (Φ, Ψ), may be expressed thus:

- $\Pi \models ura(u, r)$ if and only if user $u \in \mathcal{U}$ is assigned to role $r \in \mathcal{R}$;

- $\Pi \models pra(a, o, r)$ if and only if the access privilege $a \in \mathcal{A}$ on object $o \in \mathcal{O}$ is assigned to the role $r \in \mathcal{R}$.

By separating the assignment of users to roles from the assignment of permissions to roles it is possible for user-role and permission-role assignments to be changed independently of each other in implementations of $RBAC^P_{H2A}$ policies. Thus, access policy maintenance is simplified relative to the discretionary access control policies that were, until recently, used as a matter of course to help to protect the information in databases.

In the $RBAC^P_{H2A}$ model, an $RBAC^P_{H2A}$ *role hierarchy* is defined as a (partially) ordered (and finite) set of roles. The ordering relation is a role seniority relation. In an $RBAC^P_{H2A}$ program Π, a 2-place predicate $senior\_to(r_i, r_j)$ is used to define the seniority ordering between pairs of roles i.e., the role $r_i \in \mathcal{R}$ is a more senior role (or more powerful role) than role $r_j \in \mathcal{R}$. If r_i is senior to r_j then any user assigned to the role r_i has at least the permissions that users assigned to role r_j have. Role hierarchies are important for implicitly specifying the inheritance of access to resources.

The semantics of the `senior_to` relation may be expressed thus:

- $\Pi \models senior\_to(r_i, r_j)$ if and only if the role $r_i \in \mathcal{R}$ is senior to the role $r_j \in \mathcal{R}$ in an $RBAC^P_{H2A}$ role hierarchy.

The `senior_to` relation may be defined as the transitive and reflexive closure of an irreflexive binary relation `ds` (where `ds` is short for "directly senior to"). The semantics of `ds` may be expressed, in terms of an $RBAC^P_{H2A}$ program Π, thus:

- $\Pi \models ds(r_i, r_j)$ iff the role $r_i \in \mathcal{R}$ $(r_i \neq r_j)$ is senior to the role $r_j \in \mathcal{R}$ in an $RBAC^P_{H2A}$ role hierarchy defined in Π and there is no role $r_k \in \mathcal{R}$ such that $[ds(r_i, r_k) \wedge ds(r_k, r_j)]$ holds where $r_k \neq r_i$ and $r_k \neq r_j$.

In $RBAC^P_{H2A}$ programs, an $RBAC^P_{H2A}$ role hierarchy is defined by the following set of clauses (in which '_' is an anonymous variable):

$$\begin{array}{lcl} senior\_to(R1, R1) & \leftarrow & ds(R1, \_). \\ senior\_to(R1, R1) & \leftarrow & ds(\_, R1). \\ senior\_to(R1, R2) & \leftarrow & ds(R1, R2). \\ senior\_to(R1, R2) & \leftarrow & ds(R1, R3), senior\_to(R3, R2). \end{array}$$

The definition of `senior_to` assumes that the following property holds:

$$\forall r_i \in \mathcal{R}\ \exists r_j \in \mathcal{R}\ [(ds(r_i, r_j)\ \vee\ ds(r_j, r_i))\ \wedge\ (r_i \neq r_j)].$$

That is, there is no role $r_i \in \mathcal{R}$ in an $RBAC^P_{H2A}$ role hierarchy such that r_i is not related under ds to another role r_j $(r_i \neq r_j)$ i.e., there are no "isolated" roles in an $RBAC^P_{H2A}$ role hierarchy, and a single role is not a $RBAC^P_{H2A}$ role hierarchy. Although it is possible to define multiple $RBAC^P_{H2A}$ role hierarchies in an $RBAC^P_{H2A}$ program, the $RBAC^P_{H2A}$ programs that we consider in this paper are restricted to defining a single $RBAC^P_{H2A}$ role hierarchy.

In RBAC, users activate and deactivate roles in the course of *session management*. The assignment of a user $u_i \in \mathcal{U}$ to a role $r_j \in \mathcal{R}$ in a session may be represented by using a set of $active(u_i, r_j)$ facts. A user u_i appends an `active/2` fact to an $RBAC^P_{H2A}$ program to activate the role r_j and retracts an $active(u_i, r_j)$ fact when u_i no longer wishes to be active in r_j.

User-role and permission-role assignments are related via the notion of an *authorisation*. An authorisation is a triple (u, a, o) that expresses that the user u has the a access privilege on the object o. In $RBAC^P_{H2A}$ programs, an *authorisations clause* may be used to specify that a user $u_i \in \mathcal{U}$ has the $a_k \in \mathcal{A}$ access privilege on object $o_l \in \mathcal{O}$. In the case of $RBAC^P_{H2A}$ programs, the authorisations clause is defined thus:

$$\begin{array}{lcl} permitted(U, A, O) & \leftarrow & ura(U, R1), \\ & & active(U, R1), \\ & & senior\_to(R1, R2), \\ & & pra(A, O, R2). \end{array}$$

The rule that defines `permitted` is used to express that a user U may exercise the A access privilege on object O: if U is assigned to the role $R1$, U is active in $R1$, $R1$ is senior to a role $R2$ in an $RBAC^P_{H2A}$ role hierarchy, and $R2$ has been assigned the A access privilege on O.

It follows from the discussion above that in an $RBAC^P_{H2A}$ program $\Pi = (\Phi, \Psi)$ the set of logical axioms Φ includes the definitions of

`senior_to` and `permitted` whereas the set of non-logical axioms Ψ is expressed in terms of the predicates in the set {`ura`, `pra`, `ds`, `active`}.

In the context of specialising an $RBAC^P_{H2A}$ program Π, we note that the definitions of `ura`, `pra`, `ds` and `active` are part of the object level information that is used to protect the object level database in our approach. Moreover, the sets of clauses defining the extensions of the `ura`, `pra`, and `ds` relations are static relative to the set of `active` atoms in Π. That is, the set of `active` facts will change dynamically as users activate and deactivate roles. The aim of our approach is to specialise $RBAC^P_{H2A}$ programs to enable efficient access control checks to be performed by only considering user session information expressed via the set of `active` facts that is current at the time of a user's access control request.

5. The Access Control Meta-interpreter

5.1. The Interpreter

In this section, we describe the meta-interpreter that we propose for efficient access request evaluation on deductive databases that are protected by $RBAC^P_{H2A}$ programs.[6] It extends the meta-interpreter from Section 3.5. The following Prolog code is part of this meta-interpreter for $RBAC^P_{H2A}$ programs. Note that `solve_literal` has been renamed to `holds_read` and `l_solve` to `l_holds_read`.

```
holds_read(User,not(Object)) :-
          \+(holds_read(User,Object)).
holds_read(_User,Object) :- built_in(Object).

holds_read(User,Object) :-
          permitted(User,read,Object),
          fact(Object), call(Object).

holds_read(User,Object) :-
          permitted(User,read,Object),
          rule(Object,Body),
          l_holds_read(User,Body).

l_holds_read(_U,[]).
l_holds_read(U,[H|T]) :- holds_read(U,H),
                         l_holds_read(U,T).
```

[6] Recall that we restrict our attention to a consideration of read access.

```
built_in('='(X,X)).
built_in('is'(X,Y)) :- X is Y.

holds(U,O) :- holds_read(U,O).
```

Note that `fact/1` and `rule/2` describe the protected database: `fact` simply describes all predicates that the EDB provides, while `rule/2` is an encoding of all rules (views) of the IDB. More precisely, a normal clause of the form $C \leftarrow A_1, \ldots, A_m, \neg B_1, \ldots, \neg B_n$ is represented as:

$$rule(C, [A_1, \ldots, A_m, not(B_1), \ldots, not(B_n)]).$$

We use the following definition of `permitted`, as described in Section 4.

```
permitted(User,Op,Obj) :- ura(User,Role),
                          active(User,Role),
                          senior_to(Role,R2),
                          pra(R2,Op,Obj).
```

5.2. Partial Evaluation of the Interpreter

To be able to successfully specialise this interpreter using LOGEN, we used one additional improvement to the LOGEN system. Indeed, due to the presence of negated calls, the **nonvar** annotation on its own is no longer precise enough to capture the second argument of memoised calls to `holds_read`. For example, if a call such as `holds_read(u1, not(q(a,b)))` gets memoised then LOGEN would be instructed to treat the argument to `not/1` as dynamic and to thus specialise `holds_read(u1, not(X))`. In other words, essential information is thrown away and we would no longer be able to achieve Jones optimality. One solution lies in *disjunctive binding types*.[7] More precisely, LOGEN is now capable of handling disjunctions within the filter annotations — essentially by searching for the first matching binding type at specialisation time. One can also use more expressive type definitions referring to the basic argument annotations (**static**, **dynamic**, and **nonvar**); these are called *binding-types* (rather than binding-times). In our case, we can now provide the following filter declarations for our interpreter:

```
:- type literal = (not(type(literal)) ; nonvar).
:- filter  holds_read(static, type(literal)).
```

[7] Another solution would have been to rewrite the interpreter and make sure that `holds_read` was never marked as **memo**. This would have led to very similar results.

This means that if LOGEN has to memoise `holds_read(u1,not(q(a,b)))` then the first binding-type `not(type(literal))` will match the argument `not(q(a,b))` and it will generalise the call into `holds_read(u1, not(q(X,Y)))` rather than into `holds_read(u1,not(X))`.

We now memoise the calls to `holds_read` in the first clause of `holds_read` itself as well as in the second clause of `l_holds_read`. All other user predicates are marked as **unfold** and all calls to built-ins are marked as **rescall**. The full annotation can be found in Appendix C.

As an example of our approach, consider an $RBAC^P_{H2A}$ program Π with the following sets of facts:

$$DS = \{ds(r1, r2)\}.$$

$$ACTIVE = \{active(u1, r1), active(u2, r2)\}.$$

$$URA = \{ura(u1, r1), ura(u1, r2), ura(u2, r2)\}.$$

$$PRA = \{pra(r1, read, s(\_)), pra(r2, read, p(\_)), pra(r2, read, q(\_, \_)), pra(r1, read, r(\_, \_))\}.$$

Moreover, suppose that Π is used to protect the following database Δ in which `p` and `s` are EDB predicates and `q` and `r` are IDB predicates:

$$fact(p(X)).$$
$$fact(s(X)).$$
$$rule(q(X, Y), [p(X), p(Y)]).$$
$$rule(r(X, Y), [q(X, Y), s(X)]).$$

The access request `holds_read(u1,q(A,B))` issued by user `u1`, to read all instances of `q` from Δ, can now be specialised by LOGEN into:

```
holds_read(u1,q(A,B)) :- holds_read__0(A,B).

permitted__1(B,C) :- active(u1,r1).
permitted__1(D,E) :- active(u1,r2).

permitted__4(B) :- active(u1,r1).
permitted__4(C) :- active(u1,r2).

holds_read__3(B) :- permitted__4(B), p(B).

holds_read__0(B,C) :- permitted__1(B,C),
                      holds_read__3(B),
                      holds_read__3(C).
```

By inspection, it is possible to see that the effect of such a specialisation is to reduce a predicate like `permitted`, which is defined in terms of the relatively static predicates `ura`, `pra`, `ds` and `senior_to`, to tests on the run-time information that is generated in the course of session management, i.e., `active/2` facts.

It can also be seen that, if the specialiser were given access to the `active` facts, one could obtain a specialised program that is structurally almost[8] identical to the object database, hence achieving Jones optimality:

```
holds_read(u1,q(A,B)) :- holds_read__0(A,B).
holds_read__3(B) :- p(B).
holds_read__0(B,C) :- holds_read__3(B),
                      holds_read__3(C).
```

6. Performance Measures

In this section, we give some performance measures for the meta-programming approach that we propose for evaluating access requests on deductive databases that are protected by using $RBAC^P_{H2A}$ programs. Our testing involved comparing the evaluation of access requests on (i) non-specialised, and (ii) LOGEN specialised $RBAC^P_{H2A}$ meta-programs. We also repeated our (retrieval) access requests using direct unprotected calls to the databases, i.e., without any access control. This enabled us to evaluate the overhead incurred by access control.

The $RBAC^P_{H2A}$ programs that we use in our tests have included a definition of the `senior_to` relation that represents an $RBAC^P_{H2A}$ role hierarchy with 53 roles arranged as a complete lattice, and with each node/role of outdegree 3 or indegree 3. The `senior_to` relation has been materialised into a set of 312 pairs of ground binary assertions. That is, we assume the use of a partial materialisation approach such that only the role hierarchy is materialised (but not the authorisations). In practice, `senior_to` will not change frequently; as such, we envisage `senior_to` being materialised to avoid the run-time costs of recomputing `senior_to` each time an access request is evaluated.

We have experimented with variants of the $RBAC^P_{H2A}$ role hierarchy by increasing the depth of the role lattice. The summation that follows

[8] While `holds_read__0` is structurally equivalent to `q/2`, we have an extra interface clause for the fact `p/1`. It is possible to get full structural identity by slightly rewriting the interpreter and memoising on `holds_read_rule` rather than `holds_read`. A determinate post-unfolding post-processing would also get rid of the additional clauses for the facts.

describes the number of pairs of roles in the `senior_to` relation as defined by the $RBAC^P_{H2A}$ role hierarchy that we use in testing:

$$N + 2 \sum_{i=1}^{d-1/2} 3^i + (N_d * P_{>1})$$

In the summation above, N is the total number of nodes in the role lattice, d is the depth of the lattice, N_d is the number of nodes at depth d in the lattice, and $P_{>1}$ is the number of paths of length 2 or greater from a node at depth d.

The unique junior-most role/bottom element in the $RBAC^P_{H2A}$ role hierarchies that we used in our initial set of tests is assigned the read permission on all of the logical consequences of the databases that we use in testing. Moreover, our initial set of tests is based on a single user that is assigned to the most senior role/unique top element in the $RBAC^P_{H2A}$ role hierarchies/complete lattices that are used in our testing. Access requests are evaluated for this user. Our choice of user-role assignment and permission-role assignments imply that our initial set of tests are based on a worst-case scenario that involves the maximum amount of inheritance of permissions, by the senior-most role from the junior-most role, whenever an access request is evaluated.

The queries that we use in testing involve computing two binary relations `tcp` and `cycle`, and a unary relation `q`. The `tcp` relation is the transitive closure of a 2-place predicate `p`; the `cycle` relation involves computing a transitive closure in order to determine elements in the reflexive closure of `p`; the definition of `q` is a variant of the well-known `win` program.[9] The `tcp` program was chosen for inclusion in testing because of its practical significance; `cycle` was chosen because it involves some expensive recursive processing; the `q` program was chosen because it combines recursion and negation, and is a useful benchmark test for performance studies.

The definitions of the `tcp`, `cycle` and `q` predicates are expressed in our database thus:

$$\begin{aligned} tcp(X,Y) &\leftarrow p(X,Y). \\ tcp(X,Y) &\leftarrow p(X,Z), tcp(Z,Y). \end{aligned}$$

$$\begin{aligned} cycle(X,Y) &\leftarrow p(X,Y). \\ cycle(X,Y) &\leftarrow cycle(X,Z), p(Z,Y). \end{aligned}$$

[9] The `win` program describes a two-player game in which a player wins if his or her opponent has no move to make. The formalisation of this two-person game may be expressed by the clause: $win(X) \leftarrow move(X,Y), \neg win(Y)$.

$$q(X) \leftarrow p(X,Y), \neg q(Y).$$

The 2-place `p` predicate is defined by a set of 2495 facts. A total of 499 `p` facts are used to represent the chain:

$$p(a_1,a_2), p(a_2,a_3), \ldots, p(a_{498},a_{499}), p(a_{499},a_{500}).$$

An additional 1996 `p` facts are used to achieve a fan-out factor of 5. That is, for each `p` fact with the first argument a_i, where $1 \leqslant i \leqslant 499$, there are four `p` facts with the second argument of `p` equal to the value b_j, where $1 \leqslant j \leqslant 4$. For example, $p(a_1,b_1)$, $p(a_1,b_2)$, $p(a_1,b_3)$, $p(a_1,b_4)$. For the `cycle` program, at the n^{th} call to `cycle`, a chain of $(n-1)$ elements in the transitive closure of `p` is computed, and hence the goal clause $p(a_n,a_{n-1})$ is evaluated. An additional fact $p(a_{500},a_1)$ is added to the 2495 `p` facts used with `tcp` to represent the end of the cycle.

The successful $tcp(a_1,a_{500})$ query that we use in our testing involves computing a 500 element chain starting from the element a_1 and ending with the element a_{500}. To evaluate the `tcp` query by using SLD-resolution,[10] a search space comprising 499 SLD-trees with root $\leftarrow tcp(a_n,a_{500})$, where $1 \leqslant n \leqslant 499$, was generated. Each of these 499 SLD-trees spawns 5 subtrees; 4 of which fail, and one that succeeds. The four failing cases have a b_j value ($1 \leqslant j \leqslant 4$) as the second argument of a `p` fact; the succeeding subtree terminates with an answer clause of the form $p(a_s,a_t)$ where $t = s+1$, $1 \leqslant s \leqslant 499$ and $2 \leqslant t \leqslant 500$. The evaluation of the $cycle(a_1,a_1)$ query involves computing every chain from a_1 to a_w ($2 \leqslant w \leqslant 500$) in the transitive closure of `p`, until $p(a_{500},a_1)$ succeeds and hence $p(a_1,a_1)$ succeeds. The failing query in our suite of tests ($tcp(a_1,a_{501})$) is an attempt to compute a 501 element chain that terminates at the element a_{501}. The successful $q(a_1)$ query involves generating 499 failing SLDNF-trees for the 499 evaluations of the $\neg q(c_m)$ subgoal, where $1 \leqslant m \leqslant 499$. The one successful SLDNF-derivation is generated from the ground clause: $q(a_1) \leftarrow r(a_1,c_{500}), \neg q(c_{500})$.

The results of the testing of our example queries (with the 53 role $RBAC^P_{H2A}$ hierarchy that is materialised as 312 `senior_to` facts) are summarised in Table I (for the non-specialised case), and Tables II and III (for the specialised case). The queries denoted by Q_i, $1 \leqslant i \leqslant 4$, in these tables have the following meanings:

– Q_1 is the successful $tcp(a_1,a_{500})$ query;

[10] Our description here assumes the use of SLD-resolution, but naturally extends to SLG-resolution as implemented in XSB.

- Q_2 is the failed $tcp(a_1, a_{501})$ query;
- Q_3 is the successful $cycle(X, Y)$ query (all 145,850 solutions);
- Q_4 is the successful $q(X)$ query (all 1,170 solutions).

The query times are expressed in seconds, and all results are averaged over 10 runs. The time LOGEN needed to generate a specialised specialiser from the interpreter[11] was 0.050s. The prior binding-time analysis was performed (once and for all) by hand using LOGEN's graphical interface that allows easy annotation and provides colouring feedback on static and dynamic parts.[12] Timings were obtained on a Powerbook G4 1 GHz, 1GB SDRAM, with SICStus Prolog 3.11.0 and Mac OS X 10.3.2. Run times for XSB were obtained on the same machine using XSB Prolog 2.6. In our experiments, we make use of XSB's distinctive feature: it terminates for both recursive and non-recursive Datalog programs. This mechanism is known as *tabling* in XSB Prolog [37], and allows XSB Prolog to be used as a deductive database. This allows, for instance, the evaluation of query Q_3, whose evaluation does not terminate otherwise.

Table I. Average retrieval times for the non-specialised case.

Query	*With* $RBAC^P_{H2A}$	*Without* $RBAC^P_{H2A}$	*Overhead*
Q_1 *(SICStus)*	*0.135 s*	*0.003 s*	*0.132 s*
Q_1 *(XSB)*	*0.100 s*	*0.000 s*	*0.100 s*
Q_2 *(SICStus)*	*1.372 s*	*0.004 s*	*1.368 s*
Q_2 *(XSB)*	*0.100 s*	*0.000 s*	*0.100 s*
Q_3 *(XSB)*	*1.460 s*	*1.080 s*	*0.380 s*
Q_4 *(SICStus)*	*9.64 s*	*0.060 s*	*9.580 s*
Q_4 *(XSB)*	*0.109 s*	*0.010 s*	*0.099 s*

Table I shows how much overhead is introduced by the access control policy. For example, query Q_3, which takes 1.08 seconds to execute,

[11] LOGEN uses the cogen approach to specialisation and does not specialise annotated programs directly: it first generates a specialised specialiser from the annotated program (i.e., performing the second Futamura projection [14]). This has to be done only once for the entire experimentation.

[12] It is acceptable to perform the BTA by hand, as the annotation has to be generated only once and it is independent of the database as well as the access control policy.

requires an extra 0.38 seconds when access control is performed. Observe that the entry $0.000s$ means that the run time was too small to measure. There is no entry for SICStus for Q_3 as, due to the lack of tabling, the query does not terminate. Our aim is to minimise the overhead introduced by the $RBAC^P_{H2A}$ policy. By specialisation of the meta-interpreter, we considerably reduce this overhead, as illustrated in Table II. It can be observed that, after applying the LOGEN tool, the average retrieval time with access control (not counting specialisation time) is improved by a factor of up to 42.88. In all cases, the retrieval time after specialisation falls between the average times of the two previous approaches, i.e., *with* and *without* access control.

Table II. Timings (with SICStus) for retrieval with access control after normal and after aggressive specialisation.

Query	*Specialisation Time*	*Average Retrieval Time*	*Speedup Factor*	*Overhead of RBAC*
Q_1	*0.01 s*	*0.007 s*	*19.3*	*0.004 s*
Q_2	*0.01 s*	*0.032 s*	*42.88*	*0.028 s*
Q_4	*0.01 s*	*0.950 s*	*10.15*	*0.890 s*
Q'_1	*0.01 s*	*0.003 s*	*45*	*0.000 s*
Q'_2	*0.01 s*	*0.004 s*	*343*	*0.000 s*
Q'_4	*0.01 s*	*0.060 s*	*120.5*	*0.000 s*

There is, of course, a penalty introduced by this approach: the *specialisation time*, i.e., the time it takes *logen* to (re-)generate a specialised version of the meta-interpreter with an $RBAC^P_{H2A}$ policy. This is necessary whenever the policy changes or when other static data used during the specialisation changes. However, Table II shows that adding together the *average run time* and the *specialisation time* does not exceed the original run times with $RBAC^P_{H2A}$. By adjusting the annotations of the interpreter (i.e., marking more calls as unfoldable), a more aggressive specialisation can be obtained. This is shown in the lower half of Table II where Q'_i is the same query as Q_i, but involves unfolding the call to the `senior_to` clause (which is likely to remain unchanged for a long time). As can be seen, the overhead compared to evaluation without access control has been reduced to zero.

Table III shows the results when using XSB Prolog with tabling. It can be observed that, for the more aggressive criteria, the overhead is again much reduced (even though not quite reaching zero as above). For query Q_3, the specialised interpreter (see Appendix A) is almost

Table III. Timings (with XSB and tabling) for retrieval with access control after normal and after aggressive specialisation.

Query	*Specialisation Time*	*Average Run-Time*	*Speedup*
Q_1	*0.010 s*	*0.026 s*	*3.85*
Q_2	*0.010 s*	*0.024 s*	*4.17*
Q_3	*0.010 s*	*1.220 s*	*1.20*
Q_4	*0.010 s*	*0.016 s*	*6.81*
Q'_1	*0.010 s*	*0.010 s*	*10*
Q'_2	*0.010 s*	*0.008 s*	*12.5*
Q'_3	*0.010 s*	*1.130 s*	*1.29*
Q'_4	*0.010 s*	*0.013 s*	*8.38*

identical to the database without access control;[13] i.e., we have achieved Jones optimality. The only drawback of the aggressive specialisation is that each time `senior_to` changes there is an overhead of $10ms$ for specialisation (as well as the time needed to load the new specialised interpreter, which was around $10ms$ in our experiments). However, as *senior_to* will not change very often in practice, we are not paying a high price in terms of access control flexibility.

So far, the testing that we have described has been based on the scenario where the unique senior-most user inherits all access privileges on all objects (i.e., logical consequences) from the unique junior-most role in the $RBAC^P_{H2A}$ role hierarchy. In this case, it is "easy" for LOGEN to specialise with impressive speedups. In practice, however, access request evaluation will often involve significantly less permission inheritance than the maximal inheritance scenario that we have hitherto considered. We have therefore conducted tests for the case that is the diametrical opposite of maximal inheritance: the case of a user with zero inheritance of permissions i.e., a user assigned to and active in the junior-most element of our example $RBAC^P_{H2A}$ role hierarchy. The results of our testing in the zero inheritance case are given in Table IV.

In the zero-inheritance scenario, far fewer computations need to be pre-computed by the partial evaluator. The results in this case confirm what we expected: both in SICStus and in XSB Prolog, the speedups are less impressive than in the cases where multiple permission inheri-

[13] We believe the fact that our specialised interpreter runs slightly slower is probably due to caching issues.

Table IV. Retrieval times for the zero inheritance case.

Query	*Non-specialised*	*Specialised*	*Speedup*
Q_1 *(SICStus)*	*0.060 s*	*0.010 s*	*6*
Q_1 *(XSB)*	*0.012 s*	*0.010 s*	*1.2*
Q_2 *(SICStus)*	*0.180 s*	*0.020 s*	*9*
Q_2 *(XSB)*	*0.033 s*	*0.008 s*	*4.125*
Q_3 *(XSB)*	*2.36 s*	*0.000 s*	*"∞"*
Q_4 *(SICStus)*	*1.290 s*	*0.130 s*	*9.92*
Q_4 *(XSB)*	*0.190 s*	*0.110 s*	*1.72*

tance applies. Again, query Q_3 could only be evaluated in XSB Prolog due to termination issues.

We have also briefly considered an alternative Prolog system: Ciao Prolog. Timings for Ciao Prolog were obtained using version 1.11 (patch 164) on a Powermac G5 Dual 2.5 GHz, 4.5GB of RAM (this machine is faster than the one for SICStus and XSB; for example the query Q_4 runs in 3.4 s rather than 9.64 s on the Powermac G5 Dual). The results of our experiments in this case are summarised in Table V. It can be noted that the speedups obtained exceed a factor of 500 for aggressive specialisation. This is to be explained by the fact that the current version of Ciao Prolog has a higher overhead for meta-calls than SICStus or XSB, due to its tighter module system. That is, the interpretation overhead is higher, thus allowing us to achieve higher speedups.

Finally, we performed tests for an average-case scenario. For that, we considered a user u_n assigned to and active in role r_{25}. That is, we considered a user assigned to a role placed "half-way" up the 53 role $RBAC^P_{H2A}$ hierarchy (where roles are in the range $r_1 \ldots r_{53}$, and r_1 is the junior-most role and r_{53} is the senior-most role). This test involved u_n requesting the retrieval of a fact. In practice, access control might be most frequently used to check permissions for simple fact queries on a database. For the average-case scenario, we tested both specialised and non-specialised versions of the $RBAC^P_{H2A}$ interpreter with the following query Q_5:

– Q_5 is the successful $p(a_{499}, a_{500})$ query (access to a single fact) run 10000 times.

Table VI shows the timing results for u_n requesting to know whether $p(a_{499}, a_{500})$ is a logical consequence of the database. The query had to

Table V. Retrieval times (running in Ciao) for the non-specialised and specialised cases.

Query	*With* $RBAC^P_{H2A}$	*Without* $RBAC^P_{H2A}$	*Overhead*
Q_1 *(Ciao)*	*1.530 s*	*0.003 s*	*1.527 s*
Q_2 *(Ciao)*	*4.490 s*	*0.013 s*	*4.477 s*

Query	*Specialisation Time*	*Average Specialised Run-Time*	*Speedup*
Q_1 *(Ciao)*	*0.010 s*	*0.040 s*	*38.25*
Q_2 *(Ciao)*	*0.010 s*	*0.160 s*	*28.06*
Q'_1 *(Ciao)*	*0.010 s*	*0.003 s*	*510*
Q'_2 *(Ciao)*	*0.010 s*	*0.010 s*	*449*

be run 10000 times due to the very low retrieval time when accessing single facts, even in the non-specialised version.

Table VI. Retrieval times for the average case scenario.

Query	*Non-specialised*	*Specialised*	*Speedup*
Q_5 *(SICStus)*	*0.580 s*	*0.130 s*	*4.46*
Q_5 *(XSB)*	*0.121 s*	*0.031 s*	*3.9*

7. Conclusions and Further Work

We have described a partial evaluation approach for specialising access control checking on deductive databases. Our approach uses the LOGEN system to specialise $RBAC^P_{H2A}$ programs. To achieve our results we have first shown how to achieve Jones optimality for the classical vanilla self-interpreter, using the new binding type **nonvar**. The results of our experiments, using the LOGEN system, have revealed that program specialisation produces significant improvements in access request evaluation times on deductive databases protected by $RBAC^P_{H2A}$ programs. Our approach makes it possible to efficiently incorporate access control checks for deductive databases, incurring little overhead. In fact, in some cases it was possible to evaluate access requests on de-

ductive databases as efficiently as evaluating the same requests without processing access control information.

There are a number of additional issues to investigate in future work. As far as the specialisation technology is concerned, a challenging task is to obtain the annotations for our meta-interpreter automatically by an improved binding-time analysis. For the access control application, we intend to investigate the possibility of extending our approach to distributed environments where entities that request access to resources may not be known to enterprises with resources to protect, decisions on access may need to be delegated to third-parties (e.g., in the case where the information about the identity or attributes of requesters may be required), and the notion of a job function, which is central in RBAC, may not apply (as requesters for access to an enterprise's resources may have no connection with the enterprise). We also intend to extend our work to consider the processing of more general forms of policy information (e.g., business-rule specifications) and to apply our approach to emerging access control models for controlling access to Web resources (see, for example, [4]).

Acknowledgements

The authors would like to thank Annie Liu and the anonymous referees of PEPM'04 for their very helpful comments on a previous version of this paper. We are also grateful to Julia Lawall, Peter Sestoft, and the anonymous referees of Higher Order and Symbolic Computation, for their careful feedback and useful suggestions, and to Stephen-John Craig for helping out with the experiments in LOGEN.

References

1. K. R. Apt and F. Turini. *Meta-logics and Logic Programming.* MIT Press, 1995.
2. C. Baral and M. Gelfond. Logic programming and knowledge representation. *JLP*, 19/20:73–148, 1994.
3. S. Barker. Protecting deductive databases from unauthorized retrieval and update requests. *Journal of Data and Knowledge Engineering*, 23(3):231–285, 2002.
4. S. Barker. Web usage control in RSCLP. In *Proc. 18th IFIP WG Conf. on Database Security*, 2004.
5. S. Barker, M. Leuschel, and M. Varea. Efficient and flexible access control via logic program specialisation. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 190–199, New York, NY, USA, 2004. ACM Press.

6. S. Barker and P. Stuckey. Flexible access control policy specification with constraint logic programming. *ACM TISSEC*, 6(4):501–546, 2003.
7. E. Bertino, C. Bettini, E. Ferrari, and P. Samarati. An access control model supporting periodicity constraints and temporal reasoning. *ACM TODS*, 23(3):231–285, 1998.
8. E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A system to specify and manage multipolicy access control models. In *Proc. IEEE 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY 2002)*, 2002.
9. A. Bondorf and J. Palsberg. Generating action compilers by partial evaluation. *Journal of Functional Programming*, 6(2), 1996.
10. A. Briney. Information security 2000. *Information Security*, pages 40–68, 2000.
11. K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum, 1978.
12. C. Date. *An Introduction to Database Systems*. Addison-Wesley, 2003.
13. D. Ferraiolo, J. Cugini, and R. Kuhn. Role-based access control (RBAC): Features and motivations. In *Proc. of the 11th Annual Computer Security Applications Conf.*, pages 241–248, 1995.
14. Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. Reprinted from Systems · Computers · Controls 2(5), 1971, with a foreword.
15. J. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.
16. J. Gallagher and M. Bruynooghe. Some low-level transformations for logic programs. In M. Bruynooghe, editor, *Proceedings of Meta90 Workshop on Meta Programming in Logic*, pages 229–244, Leuven, Belgium, 1990.
17. B. Grosof and T. Poon. Representing agent contracts with exceptions using XML rules, ontologies and process descriptions. In *WWW 2003*, pages 340–349, 2003.
18. P. Hill and J. Gallagher. Meta-programming in logic programming. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 421–497. Oxford Science Publications, Oxford University Press, 1998.
19. S. Jajodia, P. Samarati, M. Sapino, and V. Subrahmanian. Flexible support for multiple access control policies. *ACM TODS*, 26(2):214–260, 2001.
20. N. D. Jones. Partial evaluation, self-application and types. In M. S. Paterson, editor, *Automata, Languages and Programming*, LNCS 443, pages 639–659. Springer-Verlag, 1990.
21. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
22. K. Kunen. Signed data dependencies in logic programs. *J. Log. Program.*, 7(3):231–245, 1989.
23. A. Lakhotia and L. Sterling. How to control unfolding when specializing interpreters. *New Generation Computing*, 8:61–70, 1990.
24. M. Leuschel. Homeomorphic embedding for online termination of symbolic methods. In T. Æ. Mogensen, D. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation - Essays dedicated to Neil Jones*, LNCS 2?56, pages 379–403. Springer-Verlag, 2002.

25. 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.
26. M. Leuschel, S. Craig, M. Bruynooghe, and W. Vanhoof. Specializing interpreters using offline partial deduction. In M. Bruynooghe and K.-K. Lau, editors, *Program Development in Computational Logic*, LNCS 3049, pages 341–376. Springer-Verlag, 2004.
27. M. Leuschel, J. Jørgensen, W. VanHoof, and M. Bruynooghe. Offline specialisation in Prolog using a hand-written compiler generator. *TPLP*, 4(1):139–191, 2004.
28. 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.
29. M. Leuschel and D. D. Schreye. Creating specialised integrity checks through partial evaluation of meta-interpreters. *JLP*, 36(1):149–193, 1998.
30. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
31. H. Makholm. On Jones-optimal specialization for strongly typed languages. In W. Taha, editor, *Semantics, Applications, and Implementation of Program Generation*, LNCS 1924, pages 129–148. Springer-Verlag, 2000.
32. K. Marriott and P. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, 1998.
33. B. Martens. *On the Semantics of Meta-Programming and the Control of Partial Deduction in Logic Programming*. PhD thesis, K.U. Leuven, February 1994.
34. B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling, editor, *Proceedings ICLP'95*, pages 597–613, Kanagawa, Japan, June 1995. MIT Press.
35. NIST. The economic impact of role-based access control, 2002. NIST Planning Report 02-01.
36. T. Przymusinski. On the declarative semantics of deductive databases and logic programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan-Kaufmann, 1988.
37. K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453, Minneapolis, Minnesota, May 1994. ACM.
38. D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.
39. R. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST model for role-based access control: Towards a unified standard. In *Proc. 4th ACM Workshop on Role-Based Access Control*, pages 47–61, 2000.
40. SETA. A marketing survey of civil federal government organizations to determine the need for rbac security product, 1996. SETA Corporation.
41. W. Vanhoof and B. Martens. To parse or not to parse. In N. Fuchs, editor, *Logic Program Synthesis and Transformation. Proceedings of LOPSTR'97*, LNCS 1463, pages 322–342, Leuven, Belgium, July 1997.

Appendix

A. Specialised interpreter for Q_3

A series of tests are undertaken in Section 6, showing the efficiency of $RBAC^P_{H2A}$ programs after specialising our interpreter using LOGEN. In this section, we provide the actual code obtained from the specialiser, for both aggressive and non-aggressive specialisation. Annotating the `senior_to` clause of `permitted/3` as a **rescall**, i.e., adding it to the residual code so that it is not evaluated in the specialisation process, leads to the non-aggressive approach shown below:

```
bench__0 :-
        ensure_loaded(database_cycle),
        abolish_all_tables, cputime(A),
        b2__1,
        cputime(B), C is B-A, print(C), nl.
b2__1 :-
        seniorto(r1,r53),
        holds_read_rule__2(_,_), fail.
b2__1.

/* holds_read_rule(steve,cycle(A,B)) :-
      holds_read_rule__2(B,A). */
holds_read_rule__2(B,A) :-
        seniorto(r1,r53),
        p(A,B).
holds_read_rule__2(B,A) :-
        seniorto(r1,r53),
        holds_read_rule__2(C,A),
        seniorto(r1,r53),
        p(C,B).
```

By considering the `senior_to` clause as **unfold**, i.e., being computed in the specialisation, the resulting program may be more efficient, at the expense of having to re-specialise each time a parameter to this clause changes. This slightly more aggressive result is shown as follows (where `bench__0` stays as before):

```
...
b2__1 :-
        holds_read_rule__2(_,_), fail.
b2__1.
```

```
holds_read_rule__2(B,A) :- p(A,B).
holds_read_rule__2(B,A) :-
        holds_read_rule__2(C,A), p(C,B).
```

Observe that `holds_read_rule__2` is isomorphic to the `cycle` predicate, hence Jones-optimality [20, 21, 31, 26] has been achieved.

B. The Full Interpreter

Below is the full code of the interpreter, including a predicate `bench` used for benchmarking our query $Q3$. The predicates for queries $Q1$, $Q2$, and $Q4$ are very similar. The code below is intended for XSB Prolog, minor modifications were done for SICStus (e.g., replacing `cputime/1` by `statistics/2`).

```
ura(steve,r1).

active(steve,r1).

pra(r53,read,p(_,_)).
pra(r53,read,cycle(_,_)).
pra(r53,read,tcp(_,_)).
pra(r53,read,q(_)).

:- table holds_read/2.

holds_read(User,not(Object)) :-
  \+(holds_read(User,Object)).
holds_read(_User,Object) :- built_in(Object).
holds_read(User,Object) :-
  permitted(User,read,Object), fact(Object),
  call(Object).
holds_read(User,Object) :-
  permitted(User,read,Object), derived(Object),
  holds_read_rule(User,Object).

holds_read_rule(User,Object) :- rule(Object,Body),
  l_holds_read(User,Body).

l_holds_read(_U,[]).
```

```
l_holds_read(U,[H|T]) :-
  holds_read(U,H),
  l_holds_read(U,T).

built_in('='(X,X)).
built_in('is'(X,Y)) :- X is Y.

holds(U,O):- holds_read(U,O).

permitted(User,Op,Obj) :-
  ura(User,Role),  active(User,Role),
  seniorto(Role,R2), pra(R2,Op,Obj).

fact(p(_X,_Y)).
derived(cycle(_,_)).
derived(tcp(_,_)).
derived(q(_)).

rule(cycle(X1,X2),[p(X1,X2)]).
rule(cycle(X1,X2),[cycle(X1,X3),p(X3,X2)]).
rule(tcp(X1,X2),[p(X1,X2)]).
rule(tcp(X1,X2),[p(X1,X3),tcp(X3,X2)]).
rule(q(X),[p(X,Y),not(q(Y))]).

% For benchmarking query Q3:
b2 :- holds_read(steve,cycle(_,_)),fail.
b2.
bench :- ensure_loaded('database_cycle'),
    abolish_all_tables, cputime(T1),
    b2,
    cputime(T2), R is T2-T1, print(R),nl.
```

C. The Annotated Interpreter

The annotation file, as used in the experiments (aggressive settings), is presented here. The file is also provided as one of the ready-made examples of LOGEN's web interface, which also enables users to view and edit this file in a friendly fashion.[14]

Note that the use of the **nonvar** annotation was essential to obtain good specialisation results (see also [26]). Also observe that a custom

[14] See `http://stups.cs.uni-duesseldorf.de/~pe/weblogen`.

type `literal` was added so as to avoid throwing away information within a negated call.

```
logen(ura, ura(steve,r1)).
logen(active, active(steve,r1)).
logen(pra, pra(r53,read,p(_,_))).
logen(pra, pra(r53,read,cycle(_,_))).
logen(pra, pra(r53,read,tcp(_,_))).
logen(pra, pra(r53,read,q(_))).
logen(holds_read, holds_read(A,not(B))) :-
        resnot(logen(memo,holds_read(A,B))).
logen(holds_read, holds_read(_,A)) :-
        logen(unfold, built_in(A)).
logen(holds_read, holds_read(A,B)) :-
        logen(unfold, permitted(A,read,B)),
        logen(unfold, fact(B)),
        logen(rescall, call(B)).
logen(holds_read, holds_read(A,B)) :-
        logen(unfold, permitted(A,read,B)),
        logen(unfold, derived(B)),
        logen(unfold, holds_read_rule(A,B)).
logen(holds_read_rule, holds_read_rule(A,B)) :-
        logen(unfold, rule(B,C)),
        logen(unfold, l_holds_read(A,C)).
logen(l_holds_read, l_holds_read(_,[])).
logen(l_holds_read, l_holds_read(A,[B|C])) :-
        logen(memo, holds_read(A,B)),
        logen(unfold, l_holds_read(A,C)).
logen(built_in, built_in(A=A)).
logen(built_in, built_in(A is B)) :-
        logen(rescall, A is B).
logen(holds, holds(A,B)) :-
        logen(unfold, holds_read(A,B)).
logen(permitted, permitted(A,B,C)) :-
        logen(unfold, ura(A,D)),
        logen(unfold, active(A,D)),
        logen(unfold, seniorto(D,E)),
        logen(unfold, pra(E,B,C)).
logen(fact, fact(p(_,_))).
logen(derived, derived(cycle(_,_))).
logen(derived, derived(tcp(_,_))).
logen(derived, derived(q(_))).
```

```
logen(rule, rule(cycle(A,B),[p(A,B)])).
logen(rule, rule(cycle(A,B),[cycle(A,C),p(C,B)])).
logen(rule, rule(tcp(A,B),[p(A,B)])).
logen(rule, rule(tcp(A,B),[p(A,C),tcp(C,B)])).
logen(rule, rule(q(A),[p(A,B),not(q(B))])).

logen(b2, b2) :-
        logen(memo, holds_read(steve,cycle(_,_))),
        logen(rescall, fail).
logen(b2, b2).
logen(bench, bench) :-
        logen(rescall, ensure_loaded(database_cycle)),
        logen(rescall, abolish_all_tables),
        logen(rescall, cputime(A)),
        logen(unfold, b2),
        logen(rescall, cputime(B)),
        logen(rescall, C is B-A),
        logen(rescall, print(C)),
        logen(rescall, nl).

:- type literal = (not(type(literal)) ; nonvar).
:- filter holds_read(static, type(literal)).
```