Optimising Active Database Rules by Partial Evaluation and Abstract Interpretation

James Bailey[1], Alexandra Poulovassilis[2], Simon Courtenage[3]

[1]Dept. of Computer Science, University of Melbourne, jbailey@cs.mu.oz.au
[2]School of Computer Science and Information Systems, Birkbeck College, Univ. of London, ap@dcs.bbk.ac.uk
[3]Cavendish School of Computer Science, University of Westminster, courtes@westminster.ac.uk

Abstract. A key issue for active databases is optimising the execution event-condition-action rules. In this paper we show how partial evaluation provides a formal and general route to optimising such rules. We produce a specialised version of the rule execution semantics for each possible sequence of actions that may execute from the current database state. This gives the opportunity to optimise rule execution for each particular sequence of actions. We obtain information about possible sequences of rule executions actions by applying abstract interpretation to the rule execution semantics. Our techniques are applicable both statically, i.e. at rule compilation time, and dynamically, during rule execution.

1 Introduction

Active databases provide reactive functionality by supporting event-condition-action rules of the form 'on *event* if *condition* do *actions*'. Active rules allow reactive functionality to be defined and managed centrally within the database rather than being encoded in diverse applications programs. Active rules thus have the potential to improve the robustness and maintainability of database applications. However, poor performance is one of the main reasons that application developers cite in their reluctance to deploy active rules [14]. Thus, optimising the run-time execution of active rules is a key issue for their wider acceptance and deployment. Given their data-intensive nature, previous research has aimed at either adopting existing database optimisation techniques, or developing special-purpose solutions for improving execution efficiency. In contrast, in this paper we show how the programming language framework of *partial evaluation* provides a formal and general route to optimising active database rules.

Partial evaluation [13, 12] aims to improve program efficiency by producing specialised versions of the program for specific input values. In the case of sets of active rules, the 'program' being optimised is the rule execution semantics and the 'input values' are the current database state and the rule actions currently awaiting execution. Producing a specialised version of the rule execution semantics for each possible sequence of actions that may execute on that database state provides the opportunity to optimise rule execution for each particular sequence of actions, for example by abstraction of common sub-queries from the sequences of conditions that will need to be evaluated.

We obtain information about possible sequences of actions by applying abstract interpretation [1, 6, 18] to the rule execution semantics, using an abstract representation of the current database state and current action(s) awaiting execution. In [2], we presented a framework for termination analysis of active rules using abstract interpretation. Here, we use similar techniques for generating the set of input values with respect to which the rule execution semantics should be partially evaluated and optimised. Our techniques are applicable both statically, i.e. at rule compilation time, and dynamically, during rule execution.

The contributions of this work are as follows: We introduce for the first time partial evaluation and abstract interpretation as techniques for globally optimising sets of active rules. The combination of these techniques generalises a number of optimisations already found in the active database literature. A key difference between this previous work and our approach is that our optimisations are automatically derived using general principles. This places rule optimisation on a sound theoretical footing, and also provides the opportunity to discover new optimisations. In particular, we show how abstract interpretation can be used to generate information about sequences of rule actions that will be executed, and this approach produces more possibilities for rule optimisation than previous graph-based analyses.

This paper is structured as follows. Section 2 specifies the rule execution semantics that we assume for the purposes of this paper and discusses abstract interpretation of these semantics. It also shows how abstract interpretation can be integrated with rule execution in order to avoid condition evaluation by making use of cheap incremental inferencing techniques. Section 3 shows how partial evaluation can be used for optimising specialisations of the rule execution semantics for single rule actions. Section 4 extends the approach to possible sequences of rule actions, using abstract interpretation to derive such sequences. Section 5 shows how abstract interpretation can also be applied at run-time, to generate sequences of rule actions that will definitely be executed so that these can be dynamically optimised. Section 6 discusses the costs and benefits our techniques and compares our approach with related work on optimising active rules. Section 7 summarises our contributions and outlines directions for future work.

2 Rule Optimisation using Abstract Interpretation

2.1 The Rule Execution Semantics

We specify the rule execution semantics that we are assuming for the purposes of this paper using a typed functional meta language supporting lazy evaluation [20]. The rule execution semantics are expressed as a function, *execSched*, which takes as input a *database* and a *schedule*.

The schedule consists of a list rule actions to be executed. The database consists of a set relation names and an extent associated with each one. Relations are of three kinds: user-defined relations and, for each user-defined relation R, two *event* relations, *insEventR* and *delEventR* and two *delta* relations $\triangle R$ and $\bigtriangledown R$. *insEventR* (*delEventR*) is non-empty if and only if the latest action executed was an insertion into (deletion from) R. $\triangle R$ ($\bigtriangledown R$) contains the tuples inserted into (deleted from) R by the latest action executed[1].

Active rules take the form 'on *event* if *condition* do *actions*'. The event part may be $\triangle R$, $\bigtriangledown R$, *insEventR* or *delEventR*, for some R. The condition part is a query. We define a rule's *event-condition* query to be the conjunction of its event query and its condition query. A rule is said to be *triggered* if the relation specified in its event part evaluates to non-empty. A rule *fires* if it is triggered and its condition part evaluates to non-empty i.e. if its event-condition query evaluates to non-empty.

Each rule has a list of one or more actions, each action being of the form $Ins\ R\ q$ or $Del\ R\ q$ for some user-defined relation R and query q. Each rule also has a *coupling mode*, which may be either *Immediate* or *Deferred*. With Immediate coupling mode, if the rule fires then its actions are prefixed to the current schedule; with Deferred coupling mode, they are suffixed. If multiple rules with the same coupling mode fire, the actions of higher-priority rules precede those of lower-priority ones on the schedule. We assume that all rules have the same *binding mode*, whereby the delta relation names in each action's query part, q, are bound to the database state in which the rule's condition is evaluated and all other relation names in q are bound to the database state in which the action is executed. A greater variety of coupling modes and binding modes can be handled by our rule analysis and optimisation techniques, which are generically applicable, but here we confine ourselves to this subset for ease of exposition. We refer the reader to [19] for a detailed description of the coupling and binding possibilities for active rules, and to [14] for an in-depth discussion of the practical aspects of deploying active rules in database applications.

We specify the rule execution semantics as a recursive function *execSched* which takes a database and schedule, and repeatedly executes the first action on the schedule, updating the schedule with the actions of rules that fire along the way (we use `teletype` font to specify *execSched* below in order to emphasize that it is an executable specification). If *execSched* terminates, it outputs the final database state and the final, empty, schedule. If it fails to terminate, it produces no output:

```
execSched : (DBState,Schedule) -> (DBState,Schedule)
execSched (db,[])  = (db,[])
execSched (db,a:s) =
    execSched o schedRules (exec (a,db), a:s)
```

[1] Allowing both event and delta relations means that both *semantic* and *syntactic* triggering are supported.

```
schedRules : (DBState,Schedule) -> (DBState,Schedule)
schedRules (db,a:s) =
    let (db,pre,suf) = fold schedRule (db,[],[]) (triggers a)
    in  (db,pre++s++suf)

schedRule : RuleId -> (DBState,Schedule,Schedule) ->
                      (DBState,Schedule,Schedule)
schedRule i (db,pre,suf) =
    if (eval (ecq i) db) = {}
    then (db,pre,suf)
    else updateSched (actions i,mode i,db,pre,suf)

updateSched (actions,Immediate,db,pre,suf) =
    (db, pre ++ (bind actions db),suf)
updateSched (actions,Deferred, db,pre,suf) =
    (db, pre, suf ++ (bind actions db))
```

In the above specification, we assume that rules are identified by unique rule identifiers. The functions *ecq*, *actions* and *mode* take a rule identifier and return the event-condition query, the list of actions, and the mode of the rule, respectively. The function *triggers* takes a rule action, and returns the id's of rules triggered by that action, in order of their priority.

The function *exec* executes an action a on a database *db* and returns the updated database. The function *schedRules* applies the function *schedRule* to each rule triggered by a (in order of the rules' priority). *schedRule* determines whether a given rule fires by invoking the *eval* function to evaluate its event-condition query w.r.t the current database state. If so, *updateSched* is called to update the schedule prefix or suffix. The function *bind* replaces the delta relation names in an action's query part by the contents of these relations in the current database state. $\circ$ denotes function composition, $[\,]$ the empty list, $(x : y)$ a list with head x and tail y, and $++$ is the list append operator. We also assume the following function which "folds" a binary function f into a list:

```
fold f x []     = x
fold f x (y:ys) = fold f (f y x) ys
```

Discussion. The above rule execution semantics are a simplification of the framework we presented in [2]. The optimisation techniques we describe here are applicable to the full framework, but we confine ourselves to this subset for ease of exposition. The above semantics encompass most of the functionality of SQL3's statement-level triggers [15] — see [3] for a full discussion of this point. They do not encompass BEFORE triggers or UPDATE events but the optimisation techniques we describe here are easily extended to these also. The above semantics do encompass semantic triggering and Deferred rule coupling, which are not supported in SQL3. Finally, do not directly consider row-level triggers here but we outline how they too could be handled in the Conclusions section.

2.2 The Abstract Execution Semantics

The abstract counterpart to *execSched* is *execSched**, given below. *execSched** is identical to *execSched* except that it operates on abstract databases and abstract schedules, and that at the "leaves" of the computation the functions *eval* and *exec* are replaced by abstract counterparts *eval** and *exec**; we distinguish abstract types and functions from their concrete counterparts by suffixing their names with a '*'.

An abstract database consists of a set of identifiers and an abstract value associated with one. Generally, these abstract values will be drawn from different domains for different abstractions. An abstract schedule consists of a list of abstract actions. An abstract action may contain abstract values in its query part, arising from the binding of delta relation names to the current abstract database state. Rules and queries are syntactic objects which are common to both the concrete and the abstract semantics. The functions *triggers*, *ecq*, *actions* and *updateSched* are the same in both semantics (in particular, *updateSched* and *bind* are polymorphic over concrete and abstract database states and schedules):

```
execSched* : (DBState*,Schedule*) -> (DBState*,Schedule*)
execSched* (db*,[])    = (db*,[])
execSched* (db*,a*:s*) =
    execSched* o schedRules* (exec* (a*,db*), a*:s*)

schedRules* : (DBState*,Schedule*) -> (DBState*,Schedule*)
schedRules* (db*,a*:s*) =
    let (db*,pre*,suf*) = fold schedRule* (db*,[],[]) (triggers a*)
    in  (db*,pre* ++ s*++ suf*)

schedRule* : RuleId -> (DBState*,Schedule*,Schedule*) ->
                       (DBState*,Schedule*,Schedule*)
schedRule* i (db*,pre*,suf*) =
    if (eval* (ecq i) db* = False)
    then (db*,pre*,suf*)
    else updateSched (actions i,mode i,db*,pre*,suf*))
```

In general, there is no guarantee that *execSched** will terminate and so we need a criterion for halting it. If the abstract domain is finite (which it is for the abstraction that we use for rule optimisation) a simple way is to maintain a history of the (db^*, *head* s^*) arguments passed to *execSched** and to halt if a repeating argument is detected — this is also the approach that we adopted for dynamic analysis of rule termination and it is discussed in [3]. In practice, the length of the execution history is unlikely to be large and thus such checking can be done efficiently.

In [2], we discussed the use of abstract interpretation for rule termination analysis and we identified three specific abstractions, two suitable for static analysis and one for dynamic analysis. In [3], we further explored the third of these abstractions and we addressed the pragmatics of dynamic termination analysis in active databases. Here, we briefly review this abstraction again since, as we will see later, it can also be used for rule optimisation.

With this abstraction, the abstract database consists of an identifier corresponding to each event query and each condition query in the rule set. These identifiers are assigned values from the 3-valued domain $\{True, False, Unknown\}$.

$exec^*$ uses a function $infer$ to deduce new truth values for these queries. $infer$ takes a query q, a truth value inferred for q w.r.t. a previous abstract database state, and the sequence of actions applied to the database since that inference, and returns a new truth value for q. This inferencing is performed using incremental techniques that determine the effect of updates on queries $[10, 21]$. For reasons of space, we do not repeat here the inferencing algorithms that we use, and refer the reader to [3] for their details. Since the properties being tested are undecidable in general, it is of course possible that $Unknown$ truth values will be inferred for queries.

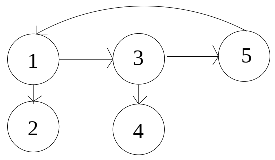
$eval^*\ q\ db^*$ returns the truth value of the event-condition query q in the current abstract database db^*. Notice that an $Unknown$ truth value for an event-condition query is treated in the same way as $True$ by the abstract execution semantics i.e. if it is not known whether a rule would fire, the abstract execution assumes that it *will* fire. In this way, the abstract execution encompasses all possible actual executions, and perhaps also some additional ones. It thus gives a conservative test for rule termination, in that if $execSched^*\ (db^*, s^*)$ terminates so will $execSched\ (db, s)$ for all databases db and schedules s whose abstract representation is db^* and s^* respectively[2].

Example 1. Consider the following rule set. Assume that all the rules have Immediate coupling mode, rule 2 has higher priority than rule 3, and rule 4 higher priority than rule 5.

$$\begin{array}{lll}
1: \text{on } \triangle R_9 & 2: \text{on } \triangle R_0 & 3: \text{on } \triangle R_0 \\
\quad \text{if } R_1 - R_0 & \quad \text{if } R_2 \cup (R_3 \bowtie R_4) & \quad \text{if } (R_3 \bowtie R_4) - R_5 \\
\quad \text{do } Ins\ R_0\ R_1 & \quad \text{do } Del\ R_1\ (R_7 \times R_8) & \quad \text{do } Ins\ R_5\ R_3 \cup (R_7 \times R_8) \\
 & & \\
4: \text{on } \triangle R_5 & 5: \text{on } \triangle R_5 & \\
\quad \text{if } R_2 \cup (R_3 \bowtie R_4) & \quad \text{if } R_7 \times R_8 & \\
\quad \text{do } Del\ R_4\ (R_5 \cup R_6) & \quad \text{do } Ins\ R_9\ (R_7 \times R_8) &
\end{array}$$

The triggering graph of these rules is as follows:

[2] We note that this is a departure from standard abstract interpretation where both branches of the conditional would be pursued and then merged. We can simplify this in our setting provided the abstraction correctly represents the effect on the abstract database state of rules scheduled in the abstract execution which would not be scheduled in some concretisation of this abstract execution. This property is indeed satisfied by the particular abstraction we are concerned with here.



Given an initial schedule consisting of the action of rule 1, i.e. $[Ins\ R_0\ R_1]$, and an initial abstract database in which all condition queries have value $Unknown$, a trace of the abstract execution of these rules on each successive call to $execSched^*$ is as follows, where c_i denotes the condition query of rule i and a_i the action of rule i:

Iteration	c_1	c_2	c_3	c_4	c_5	Schedule
1	U	U	U	U	U	$[a_1]$
2	F	U	U	U	U	$[a_2, a_3]$
3	F	U	U	U	U	$[a_3]$
4	F	U	U	U	U	$[a_4, a_5]$
5	F	U	U	U	U	$[a_5]$

We see that the execution of rule 1's action on iteration 1 causes its condition to become False (the details of the inferencing techniques used to deduce this can be found in [3]). Thereafter rule 1's condition remains False. At iteration 5, it is evaluated again and its falsity means that rule 1 cannot fire at this point. We can therefore conclude that if rule 1 is the first rule triggered then rule execution will definitely terminate within 5 iterations. ■

In general, our rule termination test consts of running $execSched^*$ once for each possible initial singleton schedule, with an initial abstract database in which all queries have an $Unknown$ value. If all invocations of $execSched^*$ terminate, then definite termination of the set of active rules can be concluded. Otherwise, the set of rules is deemed to be possibly non-terminating. We refer the reader to [2, 3] for a more detailed discussion of abstract interpretation for rule termination analysis. In Section 2.3 below we turn to its use for rule optimisation. Before doing so, we first consider the issue of the *correctness* of the abstract semantics.

An abstract database approximates a number of real databases and an abstract schedule a number of real schedules. These possible concretisations are obtained by applying a *concretisation function* to the abstract database or schedule. Suppose the following holds for all $db^* \in DBState^*$ and $s^* \in Schedule^*$, where *conc* is the concretisation function corresponding to the chosen abstraction (see [2] for details of *conc*):

$$conc\ (execSched^*\ (db^*, s^*)) \supseteq \{execSched\ (db, s) \mid (db, s) \in conc\ (db^*, s^*)\} \quad (1)$$

The LHS of this condition is the set of database/schedule pairs whose abstract representation is the output of $execSched^*\ (db^*, s^*)$. The RHS is the set of database/schedule pairs output by $execSched\ (db, s)$ for all (db, s) whose abstract representation is (db^*, s^*). Thus, condition (1) states that abstract execution from an initial state (db^*, s^*) represents a superset of the set of real executions from all (db, s) whose abstract representation is (db^*, s^*).

Two rule properties for which $execSched^*$ is conservative if (1) holds are *rule termination* and *rule unreachability*: for the first, if $execSched^*\ (db^*, s^*)$ terminates, then so must all $execSched\ (db, s)$ such that $(db, s) \in conc\ (db^*, s^*)$; for the second, if a rule i is not scheduled during the execution of $execSched^*\ (db^*, s^*)$ then it cannot be scheduled during the execution of any $execSched\ (db, s)$ such that $(db, s) \in conc\ (db^*, s^*)$. The following theorem gives sufficient conditions for (1) to hold:

Theorem 1. Condition (1) holds if
(i) for all abstract actions a^* and abstract databases db^*,
$conc\ (exec^*\ (a^*, db^*)) \supseteq \{exec\ (a, db) \mid (a, db) \in conc\ (a^*, db^*)\}$, and
(ii) for all event-condition queries q and abstract databases db^*,
$eval^*\ q\ db^* = False \Rightarrow (\forall db \in conc\ db^*\ .\ eval\ q\ db = \{\})$. ■

An example of a rule property for which (1) is not sufficient is *rule reachability*, in that if a rule i is scheduled during $execSched^*\ (db^*, s^*)$ then it may, or may not, be scheduled during some $execSched\ (db, s)$ such that $(db, s) \in conc\ (db^*, s^*)$. However, suppose that in addition to (1) the converse also holds for all $db^* \in DBState^*$ and $s^* \in Schedule^*$:

$$\{execSched\ (db, s) \mid (db, s) \in conc\ (db^*, s^*)\} \supseteq conc\ (execSched^*\ (db^*, s^*)) \quad (2)$$

Then, if a rule i is scheduled during $execSched^*\ (db^*, s^*)$, it must be scheduled during all $execSched\ (db, s)$ such that $(db, s) \in conc\ (db^*, s^*)$.

Theorem 2. Condition (2) holds if
(i) for all abstract actions a^* and abstract databases db^*,
$\{exec\ (a, db) \mid (a, db) \in conc\ (a^*, db^*)\} \supseteq conc\ (exec^*\ (a^*, db^*))$, and
(ii) for all event-condition queries q and abstract databases db^*,
$eval^*\ q\ db^* \neq False \Rightarrow (\forall db \in conc\ db^*\ .\ eval\ q\ db \neq \{\})$. ■

Observation. The abstraction described in [3] and used in Example 1 above satisfies the conditions of Theorem 1. It also satisfies the conditions of Theorem 2 provided $eval^*$ returns only True or False i.e. not if $eval^*$ returns $Unknown$. This is because $Unknown$ event-condition queries are treated as $True$ by $execSched^*$ whereas they may be $False$ in some concrete execution.

2.3 Mixed Execution Semantics

Our first observation regarding rule optimisation is that it is possible to use the abstract execution to optimise the concrete execution by not evaluating an event-condition query using *eval* if its abstract value is inferred to be $True$ or $False$ by $eval^*$. Conversely, after using *eval* to evaluate event-condition queries whose abstract value is currently $Unknown$, it is possible to upgrade this value to $True$ or $False$ in the abstract database state. This in turn will result in more precise future inferencing of abstract values, and hence in further gains in avoiding query evaluation.

These two observations lead to our *mixed* execution semantics, specified by the function $execSchedM$ below, whereby the concrete and abstract executions proceed together. The abstract database state is consulted for the presence of definite truth values, and it is updated after undertaking query evaluation when this is necessary:

```
execSchedM : (DBState,Schedule,DBState*,Schedule*) ->
             (DBState,Schedule,DBState*,Schedule*)
execSchedM (db,[],db*,[]) = (db,[],db*,[])
execSchedM (db,a:s,db*,a*:s*) =
  execSchedM o schedRulesM (exec (a,db),a:s,exec* (a*,db*),a*:s*)

schedRulesM (db,a:s,db*,a*:s*) =
  let (db,pre,suf,db*,pre*,suf*) =
      fold schedRuleM (db,[],[],db*,[],[]) (triggers a)
  in  (db,pre++s++suf,db*,pre*++s*++suf*)

schedRuleM i (db,pre,suf,db*,pre*,suf*) =
  case (eval* (ecq i) db*) of
    False  :(db,pre,suf,db*,pre*,suf*);
    True   :updateSchedM (actions i,mode i,db,pre,suf,db*,pre*,suf*);
    Unknown:let newdb* = replaceVal (ecq i) db* (eval (ecq i) db)
            in  schedRuleM i (db,pre,suf,newdb*,pre*,suf*)

updateSchedM (actions,Immediate,db,pre,suf,db*,pre*,suf*) =
  (db,pre++(bind actions db),suf,db*,pre*++(bind actions db*),suf*)
updateSchedM (actions,Deferred, db,pre,suf,db*,pre*,suf*) =
  (db,pre,suf++(bind actions db),db*,pre*,suf*++(bind actions db*))
```

Here the function $replaceVal$ takes a query q, an abstract database db^*, and a concrete value, val, for the query, and replaces the abstract value of q in db^* by $True$ or $False$ depending on whether val is non-empty or empty.

To illustrate, consider again the rule set in Example 1. Suppose we have the same initial schedule and abstract database state. Suppose also that in the initial actual database state, the conditions of rules 2 and 4 would evaluate to empty and those of rules 3 and 5 to non-empty. Then a trace of the abstract database state and schedule on each successive call to $execSchedM$ is shown below. We see that at iteration 1, the event-condition queries of rules 2 and 3 are evaluated, the abstract database state is updated accordingly and rule 3 fires. At iteration 2, the event-condition queries of rules 4 and 5 are evaluated, the abstract database state is updated and rule 5 fires. At iteration 3, the abstract value of condition 1 can be used to infer that rule 1 won't fire after the execution of rule 5's action — rule 1's event-condition query need not be evaluated. Notice also how the abstract database state has been "upgraded" with more definite information by this rule execution and will thus be more useful for optimising future rule executions:

Iteration	c_1	c_2	c_3	c_4	c_5	Schedule
1	U	U	U	U	U	$[a_1]$
2	F	F	T	U	U	$[a_3]$
3	F	F	T	F	T	$[a_5]$

Observation. For any (db, s, db^*, s^*) such that $(db, s) \in conc\ (db^*, s^*)$, let

$(db', s', db'^*, s'^*) = schedRulesM\ (exec\ (head\ s, db), s, exec^*\ (head\ s^*, db^*), s^*)$

Then it is possible to show that:

$$(db', s') = schedRules\ (exec\ (head\ s, db), s) \tag{3}$$

$$(db', s') \in conc\ (db'^*, s'^*) \tag{4}$$

$$conc\ (db'^*, s'^*) \subseteq schedRules^*\ (exec^*\ (head\ s^*, db^*), s^*) \tag{5}$$

In other words, the mixed execution gives at each step the same database/schedule pair as the real execution, and an abstract database/schedule pair which is at least as precise as the one in the abstract execution would be.

3 Rule Optimisation using Partial Evaluation

Partial evaluation [13, 12] aims to improve program efficiency by producing specialised versions of the program for specific input values. Here we show how partial evaluation can be applied to the problem of optimising a given set of active rules. A sequence of rewriting steps is performed on the definition of *execSched*, preserving its semantics but resulting in a specialisation of it for each form of rule action. Each specialisation is then optimised *for that particular form of rule action*. The rewriting steps are as follows:
Step 1: Produce an equation defining *execSched* for each possible form of action a_1, ..., a_n appearing in the current rule set:

$$execSched\ (db, a_1 : s) = execSched \circ schedRules\ (exec\ (a_1, db), a_1 : s)$$
$$execSched\ (db, a_2 : s) = execSched \circ schedRules\ (exec\ (a_2, db), a_2 : s)\ ...$$

By "form of action" we mean whether the action is an insertion or deletion, and with respect to which relation. Thus, for the rule set given in Example 1, the forms of action are $Ins\ R_0\ q$, $Del\ R_1\ q$, $Ins\ R_5\ q$, $Del\ R_4\ q$ and $Ins\ R_9\ q$, and one specialisation will be produced for *execSched* for each of these.
Step 2: Also produce an equation defining $schedRules(db, a : s)$ for each form of action:

$$\begin{aligned} schedRules\ (db, a_1 : s) = {} & let\ (db, pre, suf)\ = \\ & \quad fold\ schedRule\ (db, [\,], [\,])\ (triggers\ a_1) \\ & in\ (db, pre ++ s ++ suf) \\ schedRules\ (db, a_2 : s) = {} & let\ (db, pre, suf)\ = \\ & \quad fold\ schedRule\ (db, [\,], [\,])\ (triggers\ a_2) \\ & in\ (db, pre ++ s ++ suf)\ ... \end{aligned}$$

Step 3: Replace each call to $triggers\ a_i$ above by the specific list of rule identifiers triggered by the action a_i, and unfold the applications of $fold$. At this point it is useful to switch to a concrete example rule set. Considering again the rule set of Example 1, the specialisations of $schedRules$ for this are:

$$
\begin{aligned}
schedRules\ (db, Ins\ R_0\ q : s) &= let\ (db, pre, suf)\ = \\
&\quad\quad schedRule\ (schedRule\ (db, [\,], [\,])\ 2)\ 3 \\
&\quad in\ (db, pre ++ s ++ suf) \\
schedRules\ (db, Del\ R_1\ q : s) &= let\ (db, pre, suf)\ =\ (db, [\,], [\,]) \\
&\quad in\ (db, pre ++ s ++ suf) \\
schedRules\ (db, Ins\ R_5\ q : s) &= let\ (db, pre, suf)\ = \\
&\quad\quad schedRule\ (schedRule\ (db, [\,], [\,])\ 4)\ 5 \\
&\quad in\ (db, pre ++ s ++ suf) \\
schedRules\ (db, Del\ R_4\ q : s) &= let\ (db, pre, suf)\ =\ (db, [\,], [\,]) \\
&\quad in\ (db, pre ++ s ++ suf) \\
schedRules\ (db, Ins\ R_9\ q : s) &= let\ (db, pre, suf)\ = \\
&\quad\quad schedRule\ (db, [\,], [\,])\ 1 \\
&\quad in\ (db, pre ++ s ++ suf)
\end{aligned}
$$

Step 4: Unfold the calls to $schedRule$ — we just develop the first equation from now on:

$$
\begin{aligned}
schedRules\ (db, Ins\ R_0\ q : s) = let\ \ & (db, pre, suf)\ = \\
& if\ (eval\ (ecq\ 2)\ db) = \{\} \\
& then\ if\ (eval\ (ecq\ 3)\ db) = \{\} \\
& \quad then\ (db, [\,], [\,]) \\
& \quad else\ (db, [\,] ++ (bind\ (actions\ 3)\ db), [\,]) \\
& else\ if\ (eval\ (ecq\ 3)\ db) = \{\} \\
& \quad then\ (db, [\,] ++ (bind\ (actions\ 2)\ db), [\,]) \\
& \quad else\ (db, [\,] ++ (bind\ (actions\ 2)\ db) ++ \\
& \quad\quad\quad (bind\ (actions\ 3)\ db), [\,]) \\
in\ \ & (db, pre ++ s ++ suf)
\end{aligned}
$$

Step 5: Finally, in the RHS of the resulting equations, unfold the calls to ecq and simplify applications of ++ to the empty list:

$$
\begin{aligned}
& let\ (db, pre, suf) = \\
& \quad if\ (eval\ (\Delta R_0 \times (R_2 \cup (R_3 \bowtie R_4)))\ db) = \{\} \\
& \quad then\ if\ (eval\ (\Delta R_0 \times ((R_3 \bowtie R_4) - R_5))\ db) = \{\} \\
& \quad\quad then\ (db, [\,], [\,]) \\
& \quad\quad else\ (db, (bind\ (actions\ 3)\ db), [\,])
\end{aligned}
$$

$$
\begin{array}{l}
\quad\quad else\ if\ (eval\ (\Delta R_0 \times ((R_3 \bowtie R_4) - R_5))\ db) = \{\} \\
\quad\quad\quad then\ (db, (bind\ (actions\ 2)\ db), [\,]) \\
\quad\quad\quad else\ (db, (bind\ (actions\ 2)\ db) + + \\
\quad\quad\quad\quad\quad (bind\ (actions\ 3)\ db), [\,]) \\
in\ \ (db, pre + + s + + suf)
\end{array}
$$

The above transformations have brought together all of the event-condition query evaluations that will result from the execution of a specific form of rule action (it is possible to use a standard partial evaluator to implement these transformations). It is now possible to apply standard optimisation techniques to each resulting equation of *schedRules*. For example, common sub-queries can be abstracted from the event-condition queries so that they are evaluated at most once e.g. the sub-queries ΔR_0 and $R_3 \bowtie R_4$ from the event-condition queries of rules 2 and 3 above. It is also possible for *eval* to use previous values of queries with respect to past database states to incrementally evaluate these queries with respect to the current database state [21, 9, 4, 23, 10].

Finally, abstract execution can be "mixed into" the partially evaluated rule execution code, and hence can further optimise it, in the same way as for the original rule execution code in Section 2.3.

4 Specialising for possible sequences of actions

In Section 3 our specialisations of *execSched* were for single rule actions. Suppose we can determine that a *sequence* of actions $[a_1, ..., a_n]$ may appear on the schedule without any action a_i, $1 \le i < n$, triggering any rule, so that the values of *pre* and *suf* returned by *schedRules* are known to be empty for this sequence of actions. Then *execSched* can be specialised for such sequences of actions also. Doing this presents the opportunity to optimise such sequences of actions (note that these specialisations are additional to the single-action specialisations already generated by the treatment described in the previous section).

Such sequences of actions can be derived from the abstract execution traces obtained by running $execSched^*$ once for each possible initial singleton schedule, in each case with an initial abstract database state in which all queries have an *Unknown* value c.f. our test for rule termination described in Section 2.2. Doing this for the rule set of Example 1, we find that the actions of rules 2 and 3 may be placed consecutively on the schedule by the execution of rule 1's action and that rule 2's action can never fire any other rule. Denoting by a_2 and a_3 the actions of rules 2 and 3 ($a_2 = Del\ R_1\ (R_7 \times R_8)$, $a_3 = Ins\ R_5\ (R_3 \cup (R_7 \times R_8))$), we can therefore generate this additional specialisation for *execSched*:

$$execSched\ (db, a_2 : a_3 : s) = execSched \circ schedRules\ (exec\ (a_2, db), a_2 : a_3 : s)$$

The RHS of this equation reduces to

$$execSched \circ schedRules\ (exec\ (a_3, exec\ (a_2, db)), a_3 : s)$$

Standard update optimisation techniques can now be applied to this RHS. For example, in the query parts of actions a_2 and a_3 notice the common sub-query $R_7 \times R_8$ whose value is independent of the effect of action a_2. This common sub-query can be abstracted out from the two applications of *exec* and evaluated only once. Incremental evaluation of action queries, or sub-queries thereof, using their previous values is also possible.

Note that such sequences of actions do not *definitely* have to appear on the schedule, only that there is the *possibility* that they may appear. If a sequence does not appear then this specialisation of *execSched* will simply not be invoked. For example, rules 2 and 3 have different condition queries, so it may be the case that only one fires after some execution of rule 1's action rather than both of them. In such a case the individual specialisation of *execSched* matching a_2 or a_3 would be invoked rather than the specialisation for the sequence a_2, a_3.

We note that with the rule set of Example 1, simple inspection of the triggering graph would also have derived a_2, a_3 as a possible execution sequence, since rule 2 triggers no other rule. However, our abstract interpretation approach can yield more possibilities for optimisation than simple analysis of the triggering graph. For example, consider Example 1 again, this time with the addition of an extra rule Rule 6:on $\bigtriangledown R_1$ if $R_7 \times R_8$ do $Del\ R_9\ R_7 \times R_8$, having immediate coupling mode. This new rule is triggered by rule 2 and does not trigger any rule. If rule 1 is the first rule triggered, then the abstract execution trace is:

Iteration	c_1	c_2	c_3	c_4	c_5	c_6	Schedule
1	U	U	U	U	U	U	$[a_1]$
2	F	U	U	U	U	U	$[a_2, a_3]$
3	F	U	U	U	U	U	$[a_6, a_3]$
4	F	U	U	U	U	U	$[a_3]$
5	F	U	U	U	U	U	$[a_4, a_5]$
6	F	U	U	U	U	U	$[a_5]$

We see that a_6, a_3 is a possible sequence of actions on the schedule, and that a_6 can never fire any other rule. We can therefore generate this additional specialisation for *execSched*:

$$execSched\ (db, a_6 : a_3 : s) = execSched \circ schedRules\ (exec\ (a_6, db), a_6 : a_3 : s)$$

The RHS of this equation reduces to

$$execSched \circ schedRules\ (exec\ (a_3, exec\ (a_6, db)), a_3 : s)$$

and standard update optimisation techniques such as abstraction of common sub-expressions can now be applied to this RHS.

5 Dynamic specialisation for definite sequences of actions

So far the optimisation techniques we have described have been static ones i.e. applicable at compile-time. If we know at run-time that certain action execution sequences will *definitely* be followed from the current database state and schedule, then we can use this knowledge to dynamically perform further unfoldings of $execSched$, and thereby create further opportunities for update optimisation. This kind of definite execution information can be obtained by using a modified version of $execSched^*$ that halts if $eval^*$ returns $Unknown$ (see the Observation in Section 2.2). To illustrate, consider again the set of rules in Example 1, this time with the input abstract database state shown below and with rule 1 just having been triggered. The abstract execution trace using $execSched^*$ is:

Iteration	c_1	c_2	c_3	c_4	c_5	Schedule
1	T	T	T	T	U	$[a_1]$
2	F	T	T	T	U	$[a_2, a_3]$
3	F	T	T	T	U	$[a_3]$
4	F	T	U	T	U	$[a_4]$ or $[a_4, a_5]$

At iterations 1-3, $eval^*$ returns True or False when applied to the event-condition queries of rules 2 and 3. At iteration 4, $eval^*$ returns Unknown for rule 5's event-condition query and we halt the analysis. We conclude that a_1, a_2, a_3, a_4 is a definite execution sequence from this execution state. We can therefore dynamically generate an equation for $execSched(db, a_1 : s)$ which performs four unfoldings of $execSched$:

$$
\begin{aligned}
&execSched\ (db, a_1 : s)\ = \\
&\quad execSched \circ schedRules\ (exec\ (a_4, exec\ (a_3, exec\ (a_2, exec\ (a_1, db)))), a_4 : s)
\end{aligned}
$$

We can then optimise the RHS of this equation.

Note that this dynamically generated code is applicable only to the given execution state. After it has been executed, this specialisation becomes invalid and the default specialisation for a_1 generated at compile time is the only one that can correctly be used without further dynamic analysis.

Abstract execution can again be "mixed into" the dynamically specialised code. Note that this will now just retrace the already computed abstract execution trace up to the $Unknown$ event-condition query, and will therefore not need to be updated by the concrete execution until that point.

The overall rule execution cycle using dynamic specialisation is as follows:

```
repeat
   from the current concrete and abstract states (db,s), (db*,s*)
        execute execSched* till Unknown is encountered;
   generate specialised execSchedM;
   execute specialised execSchedM;
until s = []
```

We note that these dynamically generated specialisations subsume the specialisations generated statically in Sections 3 and 4. The precise trade-off between the cost of dynamically generating the specialised $execSchedM$ code versus the gain of using this rather than the statically generated default specialisations needs to be determined empirically, and this is an area of ongoing work.

6 Discussion

We have extended the PFL active database system [22] with some of the analysis and optimisation techniques described here. In particular, we have implemented the single-action specialisation described in Section 3, and the abstract interpretation approach to dynamic termination analysis described in [3] and Section 2.2 above. We have not yet amalgamated the abstract interpretation and partial evaluation techniques to obtain the mixed semantics and the multi-action specialisations, and this is an area of ongoing work.

In our implementation of the single-action specialisations, active rules are "compiled" into one 0-ary scheduling function for each rule action. These scheduling functions correspond to the specialised equations of $schedRules^*$ in Section 3. The scheduling functions are defined in PFL itself and so can be optimised using the same query optimiser as for other PFL queries/functions e.g. to perform common sub-expression abstraction. During rule execution, after an action has been executed its scheduling function is evaluated to determine which rules have fired, and how the schedule needs to be updated as a result. The costs incurred by our techniques are low. For undertaking the analysis/optimisation they are:

(i) Deriving possible/definite execution sequences using abstract interpretation. Given n rule actions, to statically derive all 'possible' sequences, $execSched^*$ needs to be run at most n times. To dynamically derive a definite execution sequence from some execution state, $execSched^*$ needs to be run once. The cost of the abstract inferencing itself is negligible, being based on simple query rewriting.
(ii) Generating the specialisations. In the worst case this is $\mathcal{O}(s \times n)$ for s specialisations and n rules (for a 'complete' triggering graph where each action fires all the rules).

The costs incurred during rule execution are:

(iii) Matching schedule prefixes with respect to specialisations. This retrieves a scheduling function from the database using a hash index on the function name and hence has $\mathcal{O}(1)$ cost.
(iv) Performing abstract inferencing as part of the "mixed" execution. Again, the cost of this is negligible, being based on simple query rewriting.

A key question is what kinds of rule sets are likely to benefit from our partial evaluation approach? An important feature of our technique is that it presents an opportunity to abstract common sub-expressions from conditions and action queries of rules that are evaluated as part of the same execution sequence. Consequently, it will be particularly effective for rule sets where conditions and/or action queries are significantly overlapping e.g. a set of rules that incrementally maintains the contents of a collection of similar views, such as a data cube, in response to updates on the underlying base relations.

The benefit of performing mixed abstract and actual execution depends on the precision of the abstract inferencing (but as we have noted above, this is cheap to carry out and so always worth doing). The precision of the abstract inferencing depends on the complexity of the conditions and action queries. If these are relatively simple (e.g. for simple alerter triggers or for triggers performing log updates) then inferencing will more often produce definite information about the truth/falsity of conditions, and so performing mixed execution will give commensurately greater speed-ups.

6.1 Related Work

There has been much work on local optimisation of the condition parts of active rules: [11] proposes discrimination networks for optimising repetitive evaluations of single rules, and strategies used by other systems are reviewed in [24, 19]. There has been less research, however, on global rule optimisation. The two main papers containing relevant work in this area are [16] and [7].

[16] generates alternative versions of triggers according to the different ways in which they can be invoked from a top-level transaction. Differences from our work are that: (a) The optimisations are not couched in the framework of partial evaluation and it is consequently more difficult to see the broad relationships and how they can be extended. (b) The complex behaviour of chained rule execution is not the focus for generating specialised versions of triggers. Instead, triggers are generated according to the way in which they are initially invoked from the host transaction. (c) The integration of analysis information is not made explicit and the notions of definite and possible execution sequences are not a feature.

[7] discusses optimisation of multiple rules and an optimisation is identified in the case when multiple rules are triggered by an action and none of them can trigger further rules. The main difference from our work is that the method for identifying multiple rules whose behaviour can be globally optimised is based on a simpler analysis of the triggering graph. Consequently, it does not take into account information about the current database state, and possible/definite execution sequences from it, as derived by our analysis.

A third paper, [8] looks at optimisation in the context of large numbers of triggers. However, the techniques proposed are not meant for triggers that can have arbitrary relational conditions/actions, and it is assumed that many of the triggers will be identical except for constant values. Finally, one of the few other uses of abstract interpretation in a database setting is [5], which discusses the use of abstract interpretation for queries over graph schemata.

7 Conclusions

We have described how abstract interpretation and partial evaluation can be used for optimising active database rules. Abstract interpretation can be "mixed into" the rule execution to avoid query evaluation by making use of cheap incremental inferencing techniques. Partial evaluation can be applied at compile-time to yield a specialised version of the rule execution semantics for each possible rule action. This brings together into one expression all of the event-condition query evaluations that will arise after the execution of that action. Standard query optimisation techniques can then be applied to this expression, for example abstraction of common sub-expressions and incremental evaluation.

Abstraction of common sub-expressions and incremental evaluation have been proposed before for active rules [16, 7]. The key difference between this work and our partial evaluation approach is that our optimisations are automatically derived using general principles. This places rule optimisation on a sound theoretical footing, encompassing many previous optimisation approaches, and also providing the opportunity to discover new ones. For example, we have shown how it is possible to use the abstract execution traces to produce specialised code for possible or definite sequences of actions, and such multi-action specialisations have not been proposed before.

We are currently implementing the mixed semantics and the multi-action specialisations within our PFL prototype. The next step will be to investigate the cost/benefit of dynamic versus static specialisation. Further work involves extending our optimisation techniques to row-level triggers. Our techniques for generating single-action specialisations can be re-used for these. In [3] we discussed how our abstract semantics can be modified to analyse row-level triggers also. This was in the context of termination analysis, but we believe that the same approach can be used for optimising row-level triggers and in particular for generating definite/possible execution sequences and hence multi-action specialisations — this is an area of ongoing investigation.

One avenue of further research is to investigate more closely the relationships between the abstractions and partial evaluation methods used in this paper, and the long-standing use of these techniques in functional and logic programming [1, 13, 17, 12]. Another promising area is to investigate the use of more sophisticated abstract domains in order to gain higher precision, and to evaluate the tradeoff between domain complexity and efficiency of the abstract semantics.

References

1. S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.
2. J. Bailey and A. Poulovassilis. An abstract interpretation framework for termination analysis of active rules. In *Proc. 7th Int. Workshop on Database Programming Languages LNCS 1949*, pages 249–266, Scotland, 1999.
3. J. Bailey, A. Poulovassilis, and P. Newson. A dynamic approach to termination analysis for active database rules. In *Proc. 1st Int. Conf. on Computational Logic (DOOD'2000 stream), LNCS 1861*, pages 1106–1120, London, 2000.
4. E. Baralis and J. Widom. Using delta relations to optimize condition evaluation in active databases. In *Proc. 2nd Int. Workshop on Rules in Database Systems (RIDS-95), LNCS 985*, pages 292–308, Athens, 1995.

5. A. Cortesi, A. Dovier, E. Quintarelli, and L. Tanca. Operational and abstract semantics of a query language for semi-structured information. In *Proc. 6th Int. Workshop on Deductive Databases and Logic Programming (DDLP'98)*, pages 127–140. GMD Report 22, ISSN 1435-2702, 1998.
6. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic Programming*, 13(2&3):103–179, 1992.
7. A. Dinn, N. W. Paton, and M. Howard Williams. Active rule analysis and optimisation in the Rock and Roll deductive object oriented database. *Information Systems*, 24(4):327–352, 1999.
8. Eric N. Hanson et al. Scalable trigger processing. In *Proc. 15th ICDE*, pages 266–275, Sydney, 1999.
9. F. Fabret, M. Regnier, and E. Simon. An adaptive algorithm for incremental evaluation of production rules in databases. In *Proc. 19th VLDB*, pages 455–467, Dublin, Ireland, 1993.
10. T. Griffin, L. Libkin, and H. Trickey. A correction to "Incremental recomputation of active relational expressions" by Qian and Wiederhold. *IEEE Trans. on Knowledge and Data Engineering*, 9(3):508–511, 1997.
11. E. Hanson. Rule condition testing and action execution in Ariel. In *Proc. SIGMOD 1992*, pages 49–58, San Diego, 1992.
12. N Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, 1996.
13. N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
14. A. Kotz-Dittrich and E. Simon. Active database systems: Expectations, Commercial Experience and Beyond. In Paton [19], pages 367–404.
15. K. Kulkarni, N. Mattos, and R. Cochrane. Active database features in SQL3. In Paton [19], pages 197–219.
16. F. Llirbat, F. Fabret, and E. Simon. Eliminating costly redundant computations from SQL trigger executions. In *Proc. SIGMOD 1997*, pages 428–439, 1997.
17. K. Marriott, H. Søndergaard, and N. D. Jones. Denotational abstract interpretation of logic programs. *ACM TOPLAS*, 16(3):607–648, 1994.
18. F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
19. N. Paton, editor. *Active Rules in Database Systems*. Springer-Verlag, 1999.
20. S. Peyton Jones. *The implementation of functional programming languages*. Prentice-Hall, 1987.
21. X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Trans. on Knowledge and Data Engineering*, 3(3):337–341, 1991.
22. S. Reddi, A. Poulovassilis, and C. Small. PFL: An active functional DBPL. In Paton [19], pages 297–308.
23. Martin Sköld and Tore Risch. Using partial differencing for efficient monitoring of deferred complex rule conditions. In Stanley Y. W. Su, editor, *Proc. 12th ICDE*, pages 392–401, 1996.
24. J. Widom and S. Ceri. *Active Database Systems*. Morgan-Kaufmann, San Mateo, California, 1995.