Analysis and Optimisation of Active Database Rules Using Abstract Interpretation and Partial Evaluation

James Bailey[1] and Alexandra Poulovassilis[2]

[1] Dept. of Computer Science, King's College London, Strand, London WC2R 2LS, james@dcs.kcl.ac.uk
[2] Dept. of Computer Science, Birkbeck College, University of London, Malet Street, London WC1E 7HX, ap@dcs.bbk.ac.uk

Abstract

Active databases provide reactive functionality by supporting *event-condition-action rules* (also known as *triggers*). Two key issues in active databases are analysis and optimisation of such rules. In this paper we describe how abstract interpretation and partial evaluation can be applied to these tasks, demonstrating that they provide a useful framework that both encompasses various existing database analysis and optimisation methods, and also aids in the development of new ones.

We specify a generic functional semantics for rule execution. From this we derive an abstract execution semantics and show how it can be used for rule termination analysis. We also show how applying partial evaluation to the functional semantics can be used to optimise rule execution.

Our application of both abstract interpretation and partial evaluation is tailored towards event-condition-action rules. Thus, the abstractions and specialisations that we develop are novel to our approach. Our methods have wider applicability to reactive data-intensive systems in general. We thus envisage them as being of interest to both the database and the programming language communities [1].

[1] This work is supported by EPSRC Grant No. GR/L26872.

1 INTRODUCTION

Active databases provide reactive functionality by supporting event-condition-action (ECA) rules, of the form *on* event *if* condition *do* action. [14] gives a comprehensive account of this area, including descriptions of several active database systems. Two key issues in active databases are analysis and optimisation of the run-time behaviour of ECA rules. Given their data-intensive nature, much of the research into these areas has adapted existing database techniques. In contrast, in this paper we show how the programming language techniques of abstract interpretation and partial evaluation can be applied to these tasks.

We specify a functional semantics for ECA rule execution which is independent of any particular database system or rule definition language (Section 2). From this semantics we derive an abstract execution semantics and show how this can be used for rule termination analysis (Section 3). We then show how applying partial evaluation to the semantics can be used to optimise rule execution by facilitating optimisations such as abstraction of common sub-expressions and incremental evaluation (Section 4). We also discuss how more sophisticated program specialisations can be derived from the rule triggering graph and by abstract interpretation (Section 5). We give our concluding remarks in Section 6.

2 THE RULE EXECUTION SEMANTICS

We specify our rule execution semantics by a function, $execSched$, which takes a database state and a schedule of rules, and executes the schedule, updating the database and schedule according to rules that fire along the way. Rules are identified by natural numbers, and schedules are lists of rule identifiers:

```
RuleId   == Nat
Schedule == List(RuleId)
```

A rule is said to be *triggered* if its event occurs. Executing the action of one rule may cause one or more other rules to be triggered. The list of rules triggered by each rule is represented by an extensionally-defined function:

```
triggers : RuleId -> List(RuleId)
```

An equation $triggers\ r = [r_1, \ldots, r_n]$ (where $n \geq 0$), means that the execution of rule r's action causes the event part of each rule r_i to occur, for all $1 \leq i \leq n$. The order of the r_i indicates their priority and their actions will be scheduled for execution in this order [2]. A rule may trigger itself i.e. r may be one of the r_i.

A rule *fires* if it is triggered and its condition part is true. We combine the event and condition part of each rule into a single *event-condition query*, so that the rule fires if this query is non-empty. Rules' event-condition queries and actions are represented by two extensionally defined functions:

[2] Thus, rule firing is assumed to be deterministic, as for example in the SQL3 proposal [11]. Extending our framework to allow non-deterministic firings of rules with the same priority is an area of future work.

```
ecq    : RuleId -> Query
action : RuleId -> Action
```

We do not specify the types $Query$ and $Action$ here as they depend on the particular active database system e.g. relational, object-oriented etc.

The $execSched$ function is listed below. It is initially called with arguments $(db_0, s_0, [\,])$ where db_0 is the current database state and s_0 is the initial list of triggered rules. $execSched$ accumulates the list of rules that are executed, and the changes they make to the database state, in its third parameter (the log variable). Note that the types $DBState$ and $Change$ depend on the specific database system. Accumulating the log is not actually needed for the purposes of calculating the final database state but, as we will see in Section 4, it is useful for optimising rule execution.

If $execSched$ terminates, it outputs the final database state and log, and the final, empty, schedule. If it fails to terminate it produces no output. The function $exec$ executes the action part of the first rule on the schedule, r, with respect to the current database state, returning a new database state. $exec$ depends on the specific database language so we have not defined it here. The function $schedRule$ is applied to each rule triggered by r, in order of these rules' priority, thereby constructing the list of rules to be prefixed to the current schedule. $schedRule$ determines whether a given rule, t, fires by evaluating its event-condition query. If this is non-empty t's action is added to the end of the schedule prefix. As with $exec$, the function $empty$ depends on the particular database language so we have not defined it here.

```
execSched:(DBState,Schedule,List(RuleId,Change)) ->
          (DBState,Schedule,List(RuleId,Change))
execSched (db,s,log) =
    if (s = [])
    then (db,s,log) else execNonEmpty(db,s,log)

execNonEmpty:(DBState,Schedule,List(RuleId,Change)) ->
             (DBState,Schedule,List(RuleId,Change))
execNonEmpty (db,r:s,log) =
    let (db,c) == exec (action r) db;
        prefix == foldl (schedRule db) [] (triggers r) in
    execSched (db,prefix++s,log++[(r,c)])

schedRule:DBState -> Schedule -> RuleId -> Schedule
schedRule db prefix t =
    if empty (ecq t,db)
    then prefix else prefix ++ [t]
```

3 ABSTRACT INTERPRETATION FOR TERMINATION ANALYSIS

A particularly important type of analysis in active databases is *termination analysis* since it is possible to define ECA rules that may fire each other indefinitely. ECA rule termination is undecidable even for simple rule languages [3]. The solution to this problem currently adopted by commercial database products is to impose a fixed upper limit on the number of rule firings allowed e.g. in Oracle-8 this is 64. If this limit is reached, the actions of all the rules are rolled back. This has the drawback that rule execution sequences that would eventually terminate may exceed this predefined limit and be halted prematurely. Previous research, e.g. [2, 6], has predominantly dealt with developing simple sufficient conditions for ensuring rule termination, and can be overly conservative.

Abstract interpretation [1] has proven a useful tool in program analysis and has been applied in functional programming languages to problems such as strictness analysis [13]. Here, we apply it to the problem of determining whether a given set of ECA rules always terminates. The abstract counterpart of $execSched$ is $execSched^*$. This is identical to $execSched$ except that it operates on an abstract database state type, $DBState^*$, and on an abstract database change type, $Change^*$. Also, the base functions $empty$ and $exec$ are replaced by abstract counterparts $empty^*$ and $exec^*$. The definitions of $DBState^*$, $Change^*$, $exec^*$ and $empty^*$ will depend on both the database system and on the particular approximation being adopted (we consider an example approximation below).

An abstract database approximates a set of real databases, given by some concretisation function $conc : DBState^* \rightarrow \mathcal{P}(DBState)$. $execSched^*$ is a safe approximation of $execSched$ if, for all db^* and s:

$$\begin{array}{l} execSched^*(db^*, s, [\,]) = (dbOut^*, [\,], log^*) \\ \Rightarrow \forall db \in conc\ db^*.execSched(db, s, [\,]) = (dbOut, [\,], log) \wedge \\ \qquad\qquad\qquad\qquad (dbOut \in conc\ dbOut^*) \end{array}$$

Thus, if $execSched^*$ terminates then so does $execSched$.

It can be shown (see [5] for details) that sufficient conditions for this to hold are that, for all abstract databases db^*, event-condition queries q and actions a:

(i) $\{new\_db \mid db \leftarrow conc\ db^*; (new\_db, c) \leftarrow exec\ a\ db\} \subseteq$
$\{new\_db \mid (new\_db^*, c^*) \leftarrow exec^*\ a\ db^*; new\_db \leftarrow conc\ new\_db^*\}$, and

(ii) $empty^*\ q\ db^* = True \Rightarrow (\forall db \in conc\ db^*\ .\ empty\ q\ db = True)$.

Condition (i) ensures that the database states collectively generated at each step of the abstract execution subsume those generated at the same step of all concrete executions. Condition (ii) ensures that rules not scheduled during abstract execution would not be scheduled during any concrete execution either, so that abstract schedules are "super-schedules" of concrete schedules [3].

One possible approximation for termination analysis is to model each database object by an expression which is extended by $exec^*$ to reflect the history of updates applied to the object during rule execution. $empty^*$ is then a satisfiability test on these "history expressions". We explored this approximation for the PFL

[3]This is sufficient to guarantee that concrete schedules have no worse termination properties than abstract schedules provided the abstract execution fully represents all possible concrete executions. A simple way to guarantee this is to encode each rule's event-condition query within its action — this has no effect on the semantics of the rule. Thus, for the purposes of abstract execution, the action of rule i has the notional form $if\ \ (ecq\ i)\ \ then\ \ (action\ i)$.

active database system in [4], with a generic treatment given in [5]. In particular, [5] develops a more general framework for termination analysis than presented here, where rule binding and coupling modes are unrestricted, and rules may perform lists of actions rather than a single atomic action.

Here we present instead a new approximation that is useful both for termination analysis and, as we will see in Section 5, for deriving definite rule execution sequences that can be used for optimising rule execution. This approximation represents the truth-value of event-condition queries by $True$, $False$ or $Unknown$. $empty^*$ returns $True$ if the value of a query is currently $False$, otherwise it returns $False$. $exec^*$ uses a function $infer\_new$ to deduce new truth values for queries. $infer\_new$ takes a query q, a truth value val inferred for q w.r.t. a previous database state, and the sequence of actions as applied to the database since that inference, and returns a new truth value for q:

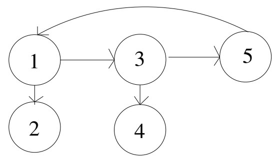
$$\begin{aligned} infer\_new\ q\ val\ as \quad &= \quad False,\ if\ ((val = False)\ and\ (still\_false\ q\ as))\ or \\ &\qquad\qquad ((satisfiable\ q\ as) = False) \\ &= \quad True,\ if\ ((val = True)\ and\ (still\_true\ q\ as))\ or \\ &\qquad\qquad ((satisfiable\ \neg q\ as) = False) \\ &= \quad Unknown,\ otherwise \end{aligned}$$

Here, $satisfiable$ determines if q is satisfiable after as have been applied, using a safe query satisfiability test such as those in [16, 7, 12]. $still\_false$ ($still\_true$) tests whether q remains false (true) after as are applied, using incremental techniques that determine the effect of updates on queries e.g. those in [9, 15]. Each of these three functions may return $Unknown$, since the properties being tested are undecidable in general. It is easy to verify that the resulting $exec^*$ and $empty^*$ functions satisfy conditions (i) and (ii) above.

Example. Let us assume a simple relational database system supporting two kinds of events: insertion into, and deletion from, relations. These events occur whenever an insertion or deletion statement is executed on a relation (irrespective of whether any change occurs to the value of the relation — see for example SQL3's statement-level triggers [11]). Consider the following five rules, where rule 2 has higher priority than rule 3, and rule 4 higher priority than rule 5.

$1:$	on $ins(R_9)$	if $R_1 - R_0$	do $ins\ R_0\ R_1$
$2:$	on $ins(R_0)$	if $R_2 \cup (R_3 \bowtie R_4)$	do $del\ R_1\ (\triangle R_0 \cup (R_7 \times R_8))$
$3:$	on $ins(R_0)$	if $(R_3 \bowtie R_4) - R_5$	do $ins\ R_5\ (R_7 \times R_8)$
$4:$	on $ins(R_5)$	if $R_2 \cup (R_3 \bowtie R_4)$	do $del\ R_4\ (R_5 \cup R_6)$
$5:$	on $ins(R_5)$	if $R_7 \times R_8$	do $ins\ R_9\ (R_7 \times R_8)$

The triggering graph of these rules is:



A trace of the abstract execution of the rules on each successive call to $exec$-$Sched^*$ is shown below for the initial singleton schedule [1]. We see that executing rule 1's action on iteration 1 causes its condition to become false. Thereafter it remains false. At iteration 5, it is evaluated again and its falsity means that rule 1 is not placed on the schedule. We therefore conclude that if rule 1 is the first rule triggered then rule execution will definitely terminate within 5 iterations.

Iteration	cond(1)	cond(2)	cond(3)	cond(4)	cond(5)	Schedule
1	U	U	U	U	U	$[1]$
2	F	U	U	U	U	$[2,3]$
3	F	U	U	U	U	$[3]$
4	F	U	U	U	U	$[4,5]$
5	F	U	U	U	U	$[5]$

In general, the termination test consists of running $execSched^*$ once for each possible initial singleton schedule, with an initial abstract database in which all relations have an $Unknown$ value. If all invocations of $execSched^*$ terminate, then definite termination of the set of ECA rules can be concluded. Otherwise, the set of rules is deemed possibly non-terminating. Of course in general $execSched^*$ itself may fail to terminate, and so we need a criterion for halting the abstract execution (concluding, in such a case, that the concrete rule execution is possibly non-terminating). A simple way is to choose *a priori* a bound on the number of iterations of $execSched^*$. Another strategy is to maintain a history of $(db^*, head(s))$ pairs for each invocation of $execSched^*$. Execution of $execSched^*$ can then be halted if a repeating argument (db^*, r) is detected, since in such a case $execSched^*$ would be non-terminating.

4 PARTIAL EVALUATION FOR RULE OPTIMISATION

Partial evaluation [10] aims to improve program efficiency by producing versions of the program specialised for specific input values. Here we apply partial evaluation to the problem of optimising a given set of active rules. Since the rules are known, the parameter $r : s$ passed to $execNonEmpty$ is static in r, which will take values in the range $1 \ldots n$, where n is the number of rules. We can thus produce an equation defining $execNonEmpty(db, i : s, log)$ for each $i \in 1 \ldots n$. The calls to $action\ i$ and $triggers\ i$ can be reduced to values of type $Action$ and $List(RuleId)$ and the applications of $foldl$ unfolded. At this point it is clearer to use a specific rule set, so we consider the rules given in the previous Example. For reasons of space we focus on the specialisation of $execNonEmpty$ to $1 : s$:

```
execNonEmpty (db,1:s,log) =
  let (db,c) == exec ( ins R0 R1 ) db;
      prefix == (schedRule db (schedRule db [] 2) 3) in
  execSched (db,prefix++s,log++[(1,c)])
```

In the clause defining $prefix$, we can now unfold the applications of $schedRule$ and reduce the functions ecq, $action$ and $++$:

```
prefix  ==  if empty (ins(R0) × (R2 ∪ (R3 ⋈ R4)), db)
            then if empty (ins(R0) × ((R3 ⋈ R4) − R5), db)
                 then [] else [3]
            else if empty (ins(R0) × ((R3 ⋈ R4) − R5), db)
                 then [2] else [2,3]
```

The above transformations bring together all of the event-condition query evaluations that will result from the execution of a given rule. We can now apply standard optimisation techniques to each resulting equation of $execNonEmpty$. Firstly, common sub-queries can be abstracted out from the event-condition queries so that they are evaluated at most once:

```
prefix  ==  let res1 == eval(ins(R0), db);
                res2 == eval(R3 ⋈ R4, db) in
            if empty (res1 × (R2 ∪ res2), db)
            then if empty (res1 × (res2 − R5), db)
                 then [] else [3]
            else if empty (res1 × (res2 − R5), db)
                 then [2] else [2,3]
```

Secondly, $eval$ and $empty$ can be memoised, and the values of queries with respect to previous database states can be used to incrementally infer their values with respect to the current database state. Clearly, it is not feasible to store actual database states in the memo table. Fortunately, they are not needed for the purposes of incremental inferencing and it is sufficient to identify successive database states by the number of rules executed so far. This is given by the length of the log parameter, which can be passed as an extra argument from $execNonEmpty$ through to $schedRule$, $eval$ and $empty$. The event-condition queries passed to $empty$ may have been transformed by the abstraction of common sub-queries, but they can be reconstructed from their rule number and this too can be passed as an extra argument from $schedRule$ to $empty$. Thus:

```
prefix  ==  let res1 == eval(ins(R0), db, log);
                res2 == eval(R3 ⋈ R4, db, log) in
            if empty (res1 × (R2 ∪ res2), ecq 2, db, log)
            then if empty (res1 × (res2 − R5), ecq 3, db, log)
                 then [] else [3]
            else if empty (res1 × (res2 − R5), ecq 3, db, log)
                 then [2] else [2,3]
```

The memo table will contain triples of the form $(query, value, db\_id)$. Calls of the form $eval(q, db, log)$ can be replaced by $memo\_eval(q, q, db, log)$, and calls $empty(pq, q, db, log)$ by $memo\_eval(pq, q, db, log) = \{\}$, where:

```
memo_eval (pq,q,db,log) =
   let j == (length log) in
   if in_memo_table q
   then let (val,i)   == lookup_table q;
            (q,new,j) == insert_table (q,inc_eval(pq,q,val,log,i),j)
        in  new
   else let (q,new,j) == insert_table (q,full_eval(pq,db),j)
        in  new
```

Here q is a query and pq the same query but possibly with some sub-queries already evaluated (due to the abstraction of common sub-queries). inc_eval performs an incremental evaluation of q based on its previous value and the updates that have occurred since this was computed, using entries from $i+1$ onwards in the log (see [15, 9] for specific techniques). In the absence of any information regarding q's previous value, $full\_eval$ evaluates pq w.r.t. the current database state. Only the latest value of q need be maintained, so $insert\_eval$ can overwrite any previous entry in the memo table with the same first component. Thus, the size of the memo table is bounded by the number of different queries that are passed as arguments to $eval$.

5 MORE SOPHISTICATED PROGRAM SPECIALISATION

If we can statically determine that a sequence of rules $[r_1, ..., r_n]$ may appear on the schedule without any rule r_i, $1 \leq i < n$, triggering any other rule (so that the value of $prefix$ is known to be empty for this sequence of rules) then we can specialise $execNonEmpty$ for such sequences of rules, in addition to the single-rule specialisations already generated above. For example, looking at the five rules in Section 3, we see that rules 2 and 3 may be placed consecutively on the schedule by the execution of rule 1 and that moreover rule 2 does not trigger any other rule. Thus:

```
execNonEmpty (db,2:3:s,log) =
    let (db,c) == exec (action 2) db;
        []     == foldl (schedRule db) [] [] in
    execSched (db,[]++3:s,log++[(2,c)])
```

which further unfolds to:

```
execNonEmpty (db,2:3:s,log) =
    let (db,c)  == exec (action 2) db;
        (db,c') == exec (action 3) db;
        prefix  == foldl (schedRule db) [] (triggers 3) in
    execSched (db,prefix++s,log++[(2,c),(3,c')])
```

Such sequences of rules can be derived from the triggering graph: for each equation $triggers\ r = [r_1, \ldots, r_n]$ we can take prefixes $[r_1, \ldots, r_k]$ such that $triggers\ r_j = [\,]$ for all $1 \leq j < k$. This presents the opportunity to optimise sequences of actions (in addition to optimising event-condition queries as discussed in the previous section). For example, $action\ 2 = del\ R_1\ (\Delta R_0 \cup (R_7 \times R_8))$ and $action\ 3 = ins\ R_5\ (R_7 \times R_8)$, so $(R_7 \times R_8)$ can be abstracted out and evaluated only once, using $eval$. Incremental evaluation using $memo\_eval$ is also possible.

Note that such sequences of rules do not definitely have to appear on the schedule, only that there is the possibility that they may appear. If they do not appear then this specialisation of $execNonEmpty$ will simply not be invoked.

In contrast, if we know that certain rule execution sequences will *definitely* be followed, we can use this knowledge to further unfold $execNonEmpty$. This kind of definite execution information can be obtained by using a modified version of $execSched^*$ that halts if $empty^*$ is passed an $Unknown$ query. Also, in addition to condition (ii) of Section 3, $empty^*$ must satisfy the following condition for all abstract databases db^* and event-condition queries q:

$$empty^*\ q\ db^* = False \Rightarrow (\forall db \in conc\ db^*\ .\ empty\ q\ db = False)$$

(the $empty^*$ function presented in Section 3 satisfies this condition). Then, given initial input $(db^*, [r], [])$, $execSched^*$ will accumulate in its third parameter the longest derivable definite execution sequence arising from rule r for any input database $db \in conc\ db^*$.

To illustrate, consider again our example set of rules, this time with the specific input database state shown at iteration 1. The abstract execution trace is:

Iteration	cond(1)	cond(2)	cond(3)	cond(4)	cond(5)	Log
1	any	T	T	T	U	$[1]$
2	F	T	T	T	U	$[1,2,3]$
3	F	T	T	T	U	$[1,2,3]$
4	F	T	U	T	U	$[1,2,3,4,5]$ or [1,2,3,4]

At iterations 1-3, the value of the condition being tested is either $True$ or $False$. At iteration 4, the value of rule 5's condition is $Unknown$ so we halt the analysis. We conclude that 1,2,3,4 is a definite execution sequence if rule 1 is triggered with this input database state. We can therefore modify the equation for $execNonEmpty(db, 1 : s, log)$ to perform three unfoldings of $execSched$ if the current database satisfies the above input condition.

6 CONCLUDING REMARKS

This paper has described how abstract interpretation and partial evaluation can be applied to termination analysis and optimisation of active database rules.

There is a large body of existing work dealing with these issues and overview presentations can be found in [14]. On the termination analysis side, [2] and [6] present techniques based on graphs constructed by considering relationships between pairs of rules. However, these techniques do not "execute" the rule set and so are less informative than our abstract interpretation approach. Also, the demonstration of their correctness does not rest upon generic conditions, but upon a detailed consideration of the behaviour of each specific technique.

In the area of optimisation, incremental evaluation and abstraction of common sub-expressions have been used in [8]. Once again, the key difference between this and our partial evaluation approach is that the former was developed in an *ad hoc* fashion, whereas our optimisations are automatically derived using general principles. Indeed, the specialisations we have presented in Section 5 have not previously appeared in the active database literature.

Our analysis and optimisation techniques can be applied to more sophisticated execution semantics (e.g. different rule binding and coupling modes and non-atomic actions [5]) and it is an interesting area of future research to apply them to SQL3 [11], which supports both statement-level and row-level triggers.

REFERENCES

[1] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.

[2] A. Aiken, J. Widom, and J. M. Hellerstein. Static analysis techniques for predicting the behavior of active database rules. *ACM Transactions on Database Systems*, 20(1):3–41, 1995.

[3] J. Bailey, G. Dong, and K. Ramamohanarao. Decidability and undecidability results for the termination problem of active database rules. In *Proceedings of the 17th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 264–273, Seattle, Washington, 1998.

[4] J. Bailey and A. Poulovassilis. Abstract interpretation for termination analysis in functional active databases. *Journal of Intelligent Information Systems, Special Issue on Functional Approach to Intelligent Information Systems*, 12(2/3):243–273, 1999.

[5] J. Bailey and A. Poulovassilis. An abstract interpretation framework for termination analysis of active rules. In *Proceedings of the 7th International Workshop on Database Programming Languages*, Kinloch Rannoch, Scotland, September 1999.

[6] E. Baralis, Ceri. S., and S. Paraboschi. Compile-time and runtime analysis of active behaviors. *IEEE Transactions on Knowledge and Data Engineering*, 10(3):353–370, 1998.

[7] E. Baralis and J. Widom. An algebraic approach to rule analysis in expert database systems. In *Proceedings of the 20th International Conference on Very Large Databases*, pages 475–486, Santiago, Chile, 1994.

[8] F. Fabret, M. Regnier, and E. Simon. An adaptive algorithm for incremental evaluation of production rules in databases. In *Proceedings of the 19th International Conference on Very Large Databases*, pages 455–467, Dublin, Ireland, 1993.

[9] 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.

[10] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[11] K. Kulkarni, N. Mattos, and R. Cochrane. Active database features in SQL3. In N. Paton, editor, *Active Rules in Database Systems*, pages 197–219. Springer–Verlag, 1999.

[12] A. Levy and Y. Sagiv. Queries independent of updates. In *Proceedings of 19th International Conference on Very Large Databases*, pages 171–181, Dublin, Ireland, 1993.

[13] A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. PhD thesis, Department of Computer Science, University of Edinburgh, December 1981.

[14] N. Paton, editor. *Active Rules in Database Systems*. Springer-Verlag, 1999.

[15] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Trans. on Knowledge and Data Engineering*, 3(3):337–341, 1991.

[16] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM*, 27(4):633–655, 1980.