On the Application of Partial Evaluation to Database Optimization

Scott Daniels and Bennet Vance
Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology

December 8, 1993

1 Introduction

The theory and practice of *partial evaluation* have attracted increasing interest in recent years, and the techniques it offers have been applied to a variety of problems in compilation and the design of algorithms. To our knowledge, however, little work has been done to investigate its possible applications to database systems. In this paper we describe some very preliminary work we have done with a view to remedying this gap in partial-evaluation research. Our specific focus of attention is *optimization* of database implementations.

We will discuss both actual partial-evaluation experiments we have already carried out and more speculative ideas that we have thought about but not implemented. The techniques we will consider have potential application both to relational and object-oriented database systems. However, most of our initial work is based completely on relational databases. We begin by reviewing basics of the relational algebra, then touch on the terminology of partial evaluation before proceeding to describe our research results and research ideas.

1.1 Quick sketch of relational algebra

The relational database model treats a database as a collection of named *relations*. A relation can be thought of as a labelled table, where the columns are called "attributes." The rows are called "tuples" in the relation, and are, conceptually at least, unordered and distinct. That is, a relation is an attribute list and a set of tuples that are described by that attribute

list. You can consider the attribute list to be a "type" for the relation tuples. Relations are the *values* in relational algebra, and there are a standard set of operators on those values, all of which produce relations as results: `select`(σ), `project`(π), `join`($\bowtie$), `cross_product`($\times$), `intersection`($\cap$), `union`($\cup$), and `difference`($-$).

A typical textbook relation [EN89]:

Product	PNAME	PNUMBER	PLOCATION	DNUM
	ProductX	1	Bellaire	5
	ProductY	2	Sugarland	5
	ProductZ	3	Houston	5
	Computerization	10	Stafford	4
	Reorganization	20	Houston	1
	Newbenefits	30	Stafford	4

The `select` operator takes two inputs, a parameter and an argument. The parameter is a predicate which is evaluated with respect to a tuple and returns either `true` or `false`. The argument is a relation, which supplies a set of tuples over which the predicate will be applied. The result of a `select` is a relation, with the same attribute list as the argument relation, containing only those tuples for which the predicate was `true`. As an example, `select`(DNUM $= 4$)(`Product`) is shown.

	PNAME	PNUMBER	PLOCATION	DNUM
	Computerization	10	Stafford	4
	Newbenefits	30	Stafford	4

The `project` operator also takes two inputs, a parameter and an argument. The parameter is a list of attributes (drawn from the attributes in its argument) The argument is a relation, which supplies data for the projection (and a set of attributes to draw on). The result of a `project` is a new relation, with an attribute list that matches the parameter, and containing tuples with the chosen attributes. For this example, we show `project`([DNUM, PLOCATION])(`Product`).

	DNUM	PLOCATION
	5	Bellaire
	5	Sugarland
	5	Houston
	4	Stafford
	1	Houston

The `join` operator takes three inputs, a parameter and two arguments. The parameter is a "join condition," a predicate on a pair of tuples (one from each of the two arguments) which identifies which pairs of tuples should be in the resulting relation. The attribute list of the result is the concatenation of the attribute lists of the two argument relations. The tuples in the result are formed by concatenating every pair of input tuples that satisfy the join condition. We will have to, of course, provide another relation for our second argument.

Dept_Locations	DNUMBER	DLOCATION
	1	Houston
	4	Stafford
	5	Bellaire
	5	Sugarland
	5	Houston

Here we show `join`(DNUM = DNUMBER)(`Product`, `Dept_Locations`).

PNAME	PNUMBER	PLOCATION	DNUM	DNUMBER	DLOCATION
ProductX	1	Bellaire	5	5	Bellaire
ProductX	1	Bellaire	5	5	Sugarland
ProductX	1	Bellaire	5	5	Houston
ProductY	2	Sugarland	5	5	Bellaire
ProductY	2	Sugarland	5	5	Sugarland
ProductY	2	Sugarland	5	5	Houston
ProductZ	3	Houston	5	5	Bellaire
ProductZ	3	Houston	5	5	Sugarland
ProductZ	3	Houston	5	5	Houston
Computerization	10	Stafford	4	4	Stafford
Reorganization	20	Houston	1	1	Houston
Newbenefits	30	Stafford	4	4	Stafford

The `cross_product` operator behaves like an unconstrained `join`. In fact, `cross_product` can be implemented simply as `join(true)`.

As we have said, the attributes lists can be considered a "type" for relational tuples. The `union`, `intersection`, and `difference` operators all behave quite similarly, in that they implement the obvious set operations, take no "parameters," and take two arguments, which must be relations with matching attribute lists.

The names of all the named relations in the database, along with their corresponding attribute lists, is called the database *schema*.

1.2 Partial Evaluation

It is beyond the scope of this paper to explain partial evaluation [Con90, CD93], nor is it necessary to understanding the paper to know a great deal about the subject. However, it will be helpful to introduce a few basic terms.

A partial-evaluator accepts as input a program P and a *portion* of the input data that P needs to execute, and produces another program as output. More precisely, let us say that the program P acts as a function of two arguments, the first of which is known, and the second of which is not; thus we invoke P as $P(s, d)$, where the value of s is known. Then the partial-evaluator, given P and s as input, generates a new program P_s. The new program P_s acts as a function of one argument, and the intent is that $P_s(d)$ should be equivalent to, but more efficient to evaluate than, $P(s, d)$. The argument s is known as *static* data, the argument d as *dynamic* data, and the program P_s as the *residual* program (or, frequently, the *residual*).

Several research partial-evaluators are presently in use; the work described here made use of Schism [Con92, Con93], which is built on top of the programming language Scheme.

2 An Experiment in Implementation

We began our investigation of the application of partial evaluation to database implementation without a specific, detailed agenda. We did have some ideas about the kinds of automatic transformations that might be possible. For example, we harbored hopes that through partial evaluation we might be able to achieve some query simplifications that are conventionally obtained through algebraic query optimization. The possibility of achieving such simplifications will be discussed further in later sections. But in our implementation experiments we decided against attempting to to design a program that would meet specific objectives when partial-evaluated. Instead, we decided to implement the simplest, most naive relational query processor imaginable, and to observe the ways in which partial evaluation was able to improve it.

2.1 The Naive Query Processor

For the purpose of this exercise, we implemented the standard relational operators `select`, `project`, and `join`, but we omitted `union`, `intersection`, `cross_product` and `difference`. We also ignored some fine points of the

definitions of the operators we did implement. For example, a `project` operation may map two distinct tuples in its argument to two result tuples that are identical, since in some cases the attribute or attributes that distinguished the input tuples will have been projected away. In theory, a relational processor should perform a duplicate-elimination step following a projection to suppress redundant tuples in the result; however, we skip this step. (In defense of our laxness on this point, we note that most commercial relational databases also skip the duplicate-elimination step unless the user specifically requests it.)

Another area of simplification is predicates; we provide only a couple of simple predicate forms. For selection operations, a predicate must have the form "ATTRIBUTE = *constant*", and for join operations the predicate must have the form "LEFT_ATTR = RIGHT_ATTR". In the latter case, LEFT_ATTR denotes an attribute from `join`'s first argument, and RIGHT_ATTR an attribute from its second. Thanks in part to these simplifications, our entire query processor is a Schism program of only about 300 lines. Nonetheless, it is capable of processing complex queries, accessing relation attributes by attribute name rather than numerical offset, and generating result relations annotated with new attribute lists.

Nothing in the implementation of the query processor was especially difficult or complicated, but the correct handling of attribute names did require some care. At issue is the following: In the interest of efficient use of storage space, relational databases do not store information about attribute names in individual tuples; this is true of commercial relational systems, and we faithfully adhered to the same convention in our miniature query processor. Thus, relations are stored in much the same way in the database as they appear in the examples in section 1.1, with a list of attribute names on top (or off to the side somewhere), and a two-dimensional matrix of attribute values in another location. Actually, it is conventional to store the attribute lists of all of the named relations together in a single data structure, which, as mentioned earlier, is called the database *schema*; we adhere to this convention as well.

2.2 The Problem of Attribute Look-ups

Now consider what is involved in evaluating a query such as the query `select`(DNUM = 4)(`Product`) illustrated above in section 1.1. For each tuple in the `Product` relation, we must consult the DNUM attribute and compare it for equality with 4; if equality holds, the tuple becomes part of the output

relation, otherwise we move on to the next tuple. Thus, for each *row* of the relation, viewing it as a matrix, we must consult the *fourth column*, since the attribute list for `Product` is (PNAME PNUMBER PLOC DNUM).

But what if the argument of `select` were not `Product`, but instead were some complex subquery possibly involving joins? What column should we look in to find DNUM in that case? In the result relation produced by the sample query `join`(DNUM = DNUMBER)(`Product`, `Dept_Locations`) from section 1.1, the DNUM attribute is again in the fourth column. If, however, the arguments are reversed, the query becomes `join`(DNUMBER = DNUM)(`Dept_Locations`, `Product`); then the `Dept_Locations` attributes precede the `Product` attributes in the result, and DNUM falls in the *sixth* column. Moreover, in general the attribute lists of join results do not appear in the database schema—they are *synthesized* attribute lists—and so there is no schema entry we can consult to find the column number associated with DNUM.

2.3 A Crude Look-up Technique

To give our operators sufficient information to evaluate predicates and join conditions, we let their arguments be pairs of attribute lists and relations rather than just relations; and similarly they produce a pair consisting of an attribute list and a relation as their result. In particular, the join operator must synthesize a new attribute list as well as calculate a result relation.

With the attribute lists passed along in the arguments to the operators, it is easy to find the column number corresponding to an attribute. In `select`, for example, we iterate over the tuples of the argument relation, as described above, and for each tuple, we loop *in lock-step* through both the attribute list and the tuple columns until we find the attribute name we are looking for. We then know we are positioned at the correct column in the tuple to compare against the constant in the selection predicate. This code is horribly inefficient—not least because we loop through the attribute list once for each tuple—but convenient to write, because we never have to bother with column numbers. By scanning the attribute list and tuple columns in tandem, we obtain a simple, reliable search.

One can imagine gradations of improvement over the crude attribute loop-up technique just described. At the first gradation, one would compute a column number corresponding to the attribute name once and for all before commencing iteration over the tuples; then the appropriate attribute could be fetched on a per-tuple basis by a simple array access, us-

ing the column number as an index. At a higher gradation, one might consider a pre-analysis of the entire query for the purpose of resolving attribute names to column numbers prior to evaluation of the query. Thus, the query `select`(DNUM = 4)(`Product`) would be transcribed as something along the lines of `select`([*column 3*] = 4)(`Product`), with all attribute indexes pre-computed. Note that with the attribute indexes pre-computed, it is no longer necessary to pass around attribute lists in the arguments to the query operators. Simple matrices of values are then sufficient; consideration of attribute names is entirely removed from actual query evaluation.

However, we did not attempt any such improvements in coding our naive query processor. Such improvements increase the complexity of a program, risk the introduction of bookkeeping errors, and can entail substantial labor. Consequently, we stuck with the simplest attribute look-up technique we could think of—the lock-step scan described above.

2.4 Partial-Evaluation of the Naive Query Processor

In partial-evaluating the naive query processor, we found it most fruitful to supply the schema and query as static data, and leave the relations themselves dynamic. Thus, we provided to the partial-evaluator exactly the same information that would be available to a conventional query optimizer—the schema and query. Given this combination of static data, the partial-evaluator produced residuals with a remarkable property: though they mention attribute names because we specified that the result of a query should consist of an attribute list as well as a relation, they make no use whatsoever of attribute names in actual query evaluation. In other words, the partial-evaluator automatically generated the highest gradation of improvement over the crude attribute look-up technique described in the previous section.

We imagine that most readers will not be interested in the actual Schism code that exhibits these effects. However, for those who are, we have provided several appendices. Appendix A (page 25) gives the Schism source for the naive query processor. Appendices B–D illustrate several specializations of that source code produced by the Schism partial-evaluator. Appendix B (page 30) shows an example where *all* the data is static, including the relations; in this case, the residual code has the query result embedded within it, and performs no work when evaluated. Appendix C (page 32) shows a classic example of the elimination of references to attribute names. The function `choose-rows.2` is an unfolded and specialized variant of `select`

that knows exactly where to look in a tuple for the selection attribute: the expression `(equal? 'f (nth (car rows) 6))` compares column number 6 of the current tuple (`(car rows)`) against the constant `'f` given in the selection predicate of the query supplied as static data. Of course, 6 is the position of the SEX attribute in the `employee` attribute list (FNAME MINIT LNAME SSN BDATE ADDR SEX SALARY SUPERSSN DNO) in the static schema at the top of page 30. The experiment in Appendix D (page 33) is similar in spirit, but more complex; in particular, it shows that partial evaluation is unfazed by the synthesized attribute lists that arise in join subqueries.

3 Conventional Query Optimization

In the previous discussion we showed how partial evaluation could relieve a database implementor of some of the tedious and error-prone coding that goes into making a database perform well. That is, the implementor could partial-evaluate a more straightforward implementation to obtain many of the same improvements that are traditionally obtained by hand. But the improvements we have discussed up to this point are not of the variety that are ordinarily considered in the literature on database query optimization. That literature is less concerned with low-level implementation details than with high-level, algebraic transformation of queries.

For example, one simple transformation allows a composition of selections to be rewritten as a single selection:

$$\texttt{select}(q)(\texttt{select}(p)(R)) \Rightarrow \texttt{select}(p \wedge q)(R) \tag{1}$$

The left-hand expression first takes a pass over R to select the elements that satisfy predicate p, and then takes another pass over those that do—this time to select the ones that also satisfy predicate q. The rule says that we may achieve the same effect by taking a single pass over R and selecting the elements that satisfy both p and q. Aside from its simplicity, this example is entirely characteristic of the kind of transformation that arises in algebraic query optimization. So it is interesting to ask whether there is any way to achieve a transformation of this kind within the framework of partial evaluation. We will return to this question presently.

But first we note that the foregoing example, while characteristic of query optimization from a strictly *algebraic* viewpoint, fails to bring out one of the central concerns in query optimization: cost estimation. Our sample transformation is virtually guaranteed to be beneficial when carried

out in the direction of the arrow. In other words, the right-hand expression is so likely to be less costly to evaluate than the left-hand expression, that one may treat the rule as a reliable heuristic, without bothering to analyze whether it will yield savings in a particular instance.

However, many algebraic transformations in database query optimization do not share this trait; for example, the rule

$$\texttt{select}(q)(\texttt{join}(p)(R,S)) \Rightarrow \texttt{join}(p)(\texttt{select}(q)(R),S) \tag{2}$$

is usually beneficial when applicable, but not always. The idea behind this transformation is that by performing a selection on the *input* to the join operator, rather than on its *output*, one reduces the amount of data that the join operator must process, and hence the cost of the operator. But this intuition can mislead in some situations. As an extreme example, suppose R is extremely large, and S is expected to be empty; then the result of the join may also be expected to be empty. In this situation, the cost of evaluating the left-hand expression will be essentially the same as the cost of evaluating the expression R, whereas the cost of evaluating the right-hand expression will be essentially the same as the cost of evaluating $\texttt{select}(q)(R)$, which may be significantly greater.

Although for simplicity's sake we have used an example that may be somewhat pathological, it illustrates an effect that is not confined to pathological situations. In general there is no definite direction in which a given algebraic transformation rule should be applied, and consequently modern query optimizers rely on cost estimation strategies to choose a low-cost expression from among the different formulations of a query that can be obtained through a succession of transformations. If we want to obtain the same kind of optimization behavior through partial evaluation, we must find a way to incorporate a cost model and cost-based decision-making strategy into the code that will be partially evaluated.

There is a further complication lurking in the example just given. We noted that transformation rule 2, carried out in the direction of the arrow, is usually beneficial *when applicable*. What is this applicability constraint? The answer is that a `select` may be "pushed inside of" a `join` in the manner expressed by rule 2 only when the selection predicate q refers exclusively to attributes of R, and not to any attributes of S. Thus, if we wish to achieve this transformation through partial evaluation, we must ensure that the applicability constraint is satisfied.

In the remainder of this section, we examine the ways in which partial evaluation might be helpful in simplifying some aspects of conventional query

optimization, and the reasons why it appears difficult to apply it to other aspects.

3.1 On-the-fly Generation of Algebraic Laws

It seems at first a tantalizing possibility that one might be able to construct a query evaluator in such a way that partial-evaluating it with respect to a query involving a select of a select, as described above, would automatically yield a residual in which only a single select was used. (Of course, in the residual that single select might not appear explicitly as an invocation of the `select` function, but would probably itself be unfolded and further optimized. The point is that one might hope that a pair of loops, implicit in the original code, would be reduced to a single loop in the residual.) In effect, the partial evaluator would be discovering and applying an algebraic transformation on the fly.

This approach to transforming queries would be a radical departure from the conventional approach, which maintains a collection of algebraic laws that have been recorded by the database implementor. The approach based on on-the-fly transformations might seem promising with respect to implementation effort and reliability, because it obviates the construction of a list of algebraic laws. However, on closer inspection it appears that approach based on on-the-fly transformations is neither desirable nor feasible. It is not desirable because it apparently provides no way to consider estimated evaluation cost in the application of algebraic laws. The impediments to feasibility are discussed below.

Despite this pessimistic assessment of on-the-fly transformations, it is interesting to consider what would be involved in achieving them through partial evaluation. Considering this question may help provide techniques for assuring the correctness of algebraic laws, and may also shed some light on the limitations of current partial-evaluation technology.

3.1.1 Simplifying the Problem

We wish to construct an example of code that can be partial-evaluated to obtain the effect of an algebraic transformation from query optimization. To make our task as easy as possible, we will confine our attention to the simple transformation discussed above, in which two selects are reduced to one. To further simplify the problem, we will assume that relations are *bags* of tuples, not sets. Also, we will write our selection operations in terms of

a more general operator that has proved convenient in expressing queries [BTBN91]. We will call this operator Φ.

Like `select`, Φ is a parameterized operator. But `select` has a single parameter, which is a predicate, and its argument is a relation (whose elements are tuples); whereas Φ has *three* parameters, and its argument is a *bag*—it need not be a bag of tuples. A few examples will give a feeling for the way Φ works:

$$\begin{aligned} \Phi(0, \texttt{square}, +)\{\!\{\}\!\} &= 0 \\ \Phi(0, \texttt{square}, +)\{\!\{3\}\!\} &= 9 \end{aligned}$$

The first example illustrates what Φ does if its argument is the empty bag: it simply returns its first parameter—zero in this instance. The second example illustrates what Φ does if its argument is a *singleton* bag: the second parameter of Φ is a function—`square` in this instance—and this function is applied to the contents of the singleton bag. Thus, $\texttt{square}(3) = 9$.

Now what about bags that are neither empty nor singleton bags? In the following example, we abbreviate $\Phi(0, \texttt{square}, +)$ as just Ψ:

$$\begin{aligned} \Psi\{\!\{1,3,5\}\!\} &= \Psi\{\!\{1\}\!\} + \Psi\{\!\{3\}\!\} + \Psi\{\!\{5\}\!\} \\ &= \texttt{square}(1) + \texttt{square}(3) + \texttt{square}(5) \\ &= 1 + 9 + 25 \\ &= 35 \end{aligned}$$

The argument to Ψ has been broken down into singleton bags, and Ψ separately applied to each of them. The results have then been combined using Φ's *third* parameter—which in this instance is $+$.

It may not be immediately apparent what Φ has to do with selection. However, selection may easily be expressed with Φ. For example, the expression

$$\Phi(\{\!\{\}\!\}, \lambda x.\textbf{if } \texttt{even}(x) \textbf{ then } \{\!\{x\}\!\} \textbf{ else } \{\!\{\}\!\}, \uplus)$$

selects just the even elements of a bag. All the complexity is in the second parameter, so let us first consider singletons as example arguments; for readability we again use Ψ to abbreviate Φ with bound parameters (in this case $\Phi(\{\!\{\}\!\}, \lambda x.\textbf{if } \texttt{even}(x) \textbf{ then } \{\!\{x\}\!\} \textbf{ else } \{\!\{\}\!\}, \uplus)$).

$$\begin{aligned} \Psi\{\!\{2\}\!\} &= \textbf{if } \texttt{even}(2) \textbf{ then } \{\!\{2\}\!\} \textbf{ else } \{\!\{\}\!\} \\ &= \{\!\{2\}\!\} \\ \Psi\{\!\{3\}\!\} &= \textbf{if } \texttt{even}(3) \textbf{ then } \{\!\{3\}\!\} \textbf{ else } \{\!\{\}\!\} \\ &= \{\!\{\}\!\} \end{aligned}$$

So we see that this Ψ correctly performs selection on singletons. It also correctly performs selection on the empty bag, since in that case it simply returns the first parameter to Φ, which is now the empty bag. For bags with multiple elements, selection is performed on the singleton constituents, and the results are combined with the bag union $\uplus$. Thus,

$$\begin{aligned}\Psi\{\!\{2,3,4,5,6\}\!\} &= \{\!\{2\}\!\} \uplus \{\!\{\}\!\} \uplus \{\!\{4\}\!\} \uplus \{\!\{\}\!\} \uplus \{\!\{6\}\!\} \\ &= \{\!\{2,4,6\}\!\}.\end{aligned}$$

In the following we consider the operator `select` no further, since its functionality is subsumed by Φ.

We now give a general, inductive definition for Φ. The definition is straightforward; indeed, it is very similar to some of the examples given above.

$$\Phi(e,f,\oplus)\{\!\{\}\!\} = e \tag{3}$$

$$\Phi(e,f,\oplus)\{\!\{x\}\!\} = f(x) \tag{4}$$

$$\Phi(e,f,\oplus)(X \uplus Y) = \Phi(e,f,\oplus)(X) \oplus \Phi(e,f,\oplus)(Y) \tag{5}$$

3.1.2 The Promotion Theorem for Φ

The operator Φ obeys a law that has been referred to as the *promotion theorem*:

$$\textit{Let } \Psi = \Phi(e,g,\otimes);\ \textit{then } \Psi \circ \Phi(\{\!\{\}\!\},f,\uplus) \equiv \Phi(e,\Psi \circ f,\otimes) \tag{6}$$

When Φ is parameterized to perform selection, this theorem gives us the result we want: it says that performing two selections in succession is equivalent to a single selection involving a more complicated predicate.[1] Consequently, if we are able to automate the transformation expressed by this theorem, we will have achieved our goal. As it turns out, we can use partial evaluation to automate the transformation in part, but not completely.[2]

[1] Readers seeking a justification for this interpretation of the promotion theorem may profit from the following observations. Recall that when Φ is parameterized for selection, f will be a function of the form $\lambda x.$**if** $p(x)$ **then** $\{\!\{x\}\!\}$ **else** $\{\!\{\}\!\}$ for some predicate p. Thus the composition $\Psi \circ f$ may be written as $\lambda x.$**if** $p(x)$ **then** $\Psi\{\!\{x\}\!\}$ **else** $\Psi\{\!\{\}\!\}$, which simplifies to $\lambda x.$**if** $p(x)$ **then** $g(x)$ **else** e. We then see that with this simplification, the right-hand side of the promotion theorem contains only a single occurrence of the looping operators Φ and Ψ.

[2] Since this paper was first written, it has come to the authors' attention that the use of partial evaluation for very similar transformations has previously been investigated by Lawall [Law93].

To understand the approach one might take to automating the transformation, it may help to examine a simple proof of the promotion theorem. The proof is by induction, and proceeds by case analysis of the three different kinds of bags that appear in the definition of Φ above: empty bags, singleton bags, and composite bags. For each kind of bag B, we wish to show that applying the left-hand side of (6) to B, gives the same result as applying the right-hand side to B.

Throughout the following proof, we use the abbreviation $\Psi \equiv \Phi(e, g, \otimes)$.

Case $B = \{\!\{\}\!\}$:

$$\begin{array}{lrcl}
\text{LHS:} & (\Psi \circ \Phi(\{\!\{\}\!\}, f, \uplus))\{\!\{\}\!\} & = & \Psi(\Phi(\{\!\{\}\!\}, f, \uplus)\{\!\{\}\!\}) \\
 & & = & \Psi\{\!\{\}\!\} \\
 & & = & \Phi(e, g, \otimes)\{\!\{\}\!\} \\
 & & = & e \\
\text{RHS:} & \Phi(e, \Psi \circ f, \otimes)\{\!\{\}\!\} & = & e
\end{array}$$

Case $B = \{\!\{x\}\!\}$:

$$\begin{array}{lrcl}
\text{LHS:} & (\Psi \circ \Phi(\{\!\{\}\!\}, f, \uplus))\{\!\{x\}\!\} & = & \Psi(\Phi(\{\!\{\}\!\}, f, \uplus)\{\!\{x\}\!\}) \\
 & & = & \Psi(f(x)) \\
\text{RHS:} & \Phi(e, \Psi \circ f, \otimes)\{\!\{x\}\!\} & = & (\Psi \circ f)(x) \\
 & & = & \Psi(f(x))
\end{array}$$

Case $B = (X \uplus Y)$:

Induction hypothesis:

$(\Psi \circ \Phi(\{\!\{\}\!\}, f, \uplus))(X) \equiv \Phi(e, \Psi \circ f, \otimes)(X)$ and
$(\Psi \circ \Phi(\{\!\{\}\!\}, f, \uplus))(Y) \equiv \Phi(e, \Psi \circ f, \otimes)(Y)$.

Then we have

$$\begin{array}{lrcl}
\text{LHS:} & (\Psi \circ \Phi(\{\!\{\}\!\}, f, \uplus))(X \uplus Y) & & \\
 & & = & \Psi(\Phi(\{\!\{\}\!\}, f, \uplus)(X \uplus Y)) \\
 & & = & \Psi(\Phi(\{\!\{\}\!\}, f, \uplus)(X) \uplus \Phi(\{\!\{\}\!\}, f, \uplus)(Y)) \\
 & & = & \Psi(\Phi(\{\!\{\}\!\}, f, \uplus)(X)) \otimes \\
 & & & \qquad \Psi(\Phi(\{\!\{\}\!\}, f, \uplus)(Y)) \\
 & & = & (\Psi \circ \Phi(\{\!\{\}\!\}, f, \uplus))(X) \otimes \\
 & & & \qquad (\Psi \circ \Phi(\{\!\{\}\!\}, f, \uplus))(Y) \\
 & & = & \Phi(e, \Psi \circ f, \otimes)(X) \otimes \Phi(e, \Psi \circ f, \otimes)(Y) \\
 & & & \qquad \text{by induction hypothesis} \\
\text{RHS:} & \Phi(e, \Psi \circ f, \otimes)(X \uplus Y) & & \\
 & & = & \Phi(e, \Psi \circ f, \otimes)(X) \otimes \Phi(e, \Psi \circ f, \otimes)(Y)
\end{array}$$

With the exception of the invocation of the induction hypothesis in the case $B = (X \uplus Y)$, every step of the proof is either a simple application—or in the terminology of partial evaluation, a simple *unfolding*—of one of the equations in the definition of Φ, or else an expansion of the definition of composition (i.e., $(g \circ f)(x) \equiv g(f(x))$).

We may imagine transcribing the definition of Φ as a function in a Schism program. Then partial-evaluating expressions of the form $(\Psi \circ \Phi(\{\!\{\}\!\}, f, \uplus))\{\!\{\}\!\}$ or $(\Psi \circ \Phi(\{\!\{\}\!\}, f, \uplus))\{\!\{x\}\!\}$ should yield the same residuals that would have been obtained from the expressions $\Phi(e, \Psi \circ f, \otimes)\{\!\{\}\!\}$ or $\Phi(e, \Psi \circ f, \otimes)\{\!\{x\}\!\}$, which is exactly the effect we want. That is, partial evaluation may be expected to expand and simplify these expressions in just the way we did in our proof above. Starting with the left-hand side of the promotion theorem, we will not arrive precisely at the right-hand side of the theorem; but the result we obtain will presumably be no more costly to evaluate than if we actually obtained the right-hand side.

Unfortunately, when the argument is of the form $(X \uplus Y)$, partial evaluation will not be quite so successful, because invocation of the induction hypothesis does not correspond to anything that we know of in partial evaluation. Use of the induction hypothesis *does* correspond to the *folding* step of the Burstall-Darlington program transformation paradigm [BD77], so a rewrite step of this kind is not categorically beyond the reach of automation; but it may not fit into the partial-evaluation framework.

There is an another respect in which current partial-evaluation technology may not be up to the task of automating a transformation such as that expressed by the promotion theorem. Above we considered the application of $\Psi \circ \Phi(\{\!\{\}\!\}, f, \uplus)$—let us call this expression E—to particular bag constructs, *viz.*, $\{\!\{\}\!\}$, $\{\!\{x\}\!\}$, and $(X \uplus Y)$. But what if E were applied to an arbitrary argument B whose structure was completely unknown? We would still like to obtain the benefits offered by the promotion theorem, which holds for arbitrary arguments. However, if B is dynamic, no such benefits will be obtained by partial-evaluating the application $E(B)$.

A simple rewrite of $E(B)$ can overcome this obstacle. We may write $E(B)$ as

$$\begin{array}{l} \textbf{case } B \textbf{ of} \\ \quad \begin{array}{rcl} \{\!\{\}\!\} & \Rightarrow & E(\{\!\{\}\!\}) \\ \{\!\{x\}\!\} & \Rightarrow & E(\{\!\{x\}\!\}) \\ (X \uplus Y) & \Rightarrow & E(X \uplus Y). \end{array} \end{array}$$

Now the argument to E is partially static in each instance, and enough

information is available to unfold Φ in the manner we illustrated above. But it would be tedious to have to perform this rewriting of $E(B)$ by hand. It would be preferable if the partial-evaluator carried out rewrites of this kind automatically prior to unfolding and specialization.[3] Moreover, as we shall discuss in section 4, there are other situations in which automatically inserting code that separates different cases may be beneficial. There is a danger, however, that indiscriminate rewriting of this kind could do more harm than good, since its immediate effect is to increase both code size and execution time. Sophisticated analysis might be required to determine whether and where to insert case discrimination constructs.

3.1.3 Summary

There is a tantalizing possibility that partial evaluation can be made to automatically carry out query optimizations that are conventionally achieved through application of rewrite rules. However, even for the simple optimization we considered in depth in this section, current partial-evaluation technology is inadequate to automate the process completely. For more complicated transformations, such as the example involving join we gave in the introduction to this section, the difficulties are greater, and the prospects of overcoming them are dimmer.

3.2 An Optimizing Query Evaluator

On one level, the conclusions of the previous section are discouraging, in that the prospects of using partial evaluation to perform on-the-fly generation of transformations are not bright. But the idea of generating transformations on the fly may not be the best application of partial evaluation to query optimization anyway. If one were to succeed in achieving such transformations, then new problems would be created that might be even more difficult to surmount. Probably the biggest of these is the problem of integrating query evaluation cost estimation and cost-based decision-making into the partial-evaluation framework.

In this section we consider a less radical departure from the conventions of query optimization, and suggest a use of partial evaluation that allows us

[3] Lawall [Law93] has shown how one can coax a partial-evaluator into performing this kind of rewriting automatically by first inserting calls to a *copy* function at strategic points in the code to be expanded. However, the insertion of these calls to *copy* apparently must still be carried out manually.

to continue to use cost estimation to control an optimizer search. In current query processing systems, optimization is a completely separate step from query execution. Thus, a query algebra expression is optimized to produce a query plan, and subsequently the query execution engine is invoked to carry out the instructions in the query plan. The alternative we will now consider is to combine these two steps into a single query-evaluation step that accepts an algebraic query as input, and produces the query result as output. However, our intent is not that this query evaluator should proceed in such a naive manner as the evaluator of section 2. Rather, we intend that it should incorporate within it a rule-based optimization search to decide the best way to execute the query; but rather than emitting a query plan, it may immediately proceed to perform the execution strategy it has decided on.

Now if we make available as static data all the information that would have been available to a conventional optimizer—in other words, if we supply as static data an algebraic query as well as the database statistics that serve as the basis for cost estimation—then partial evaluation of our query evaluator should yield a residual in which optimization decisions have been pre-computed. Thus, the residual plays the same role as a query plan in a more conventional approach. The partial-evaluation step corresponds to running the optimizer, and executing the residual corresponds to running a query execution engine with the query plan as input.

One may ask what we have gained by combining the optimizer and execution engine into a single monolithic program. By keeping them separate, as is conventional, one would appear to have better modularity; and at first glance, it seems that at best partial evaluation allows us to get back to our starting point—it achieves the separation of the optimization and execution phases of query processing. But this initial impression is misleading.

First of all, combining the optimizer and execution engine does not mean they have to be bundled together in a huge mass of spaghetti code. The source code for these two components may still be nicely modularized, and the interface between them may still be kept narrow. What has changed is that when optimization and execution are separate phases, the interface is a data structure (*viz.*, a query plan), whereas when the two phases are combined into one, the interface between them is a function-call interface.

Second, and more important, is that use of partial evaluation to mimic the effect of the optimization phase is only one of *many* possible outcomes of applying partial evaluation to the combined program. This particular outcome is of interest because it shows that we have not lost functionality

in taking the partial-evaluation approach—the partial-evaluation approach is at least as general as that of separating optimization and execution. But there are also other outcomes of interest, which may be obtained by varying the choice of static data supplied to the partial-evaluator. If some fragment of the data needed for optimization is made dynamic, then some optimization decisions will be postponed until execution time. Such a scenario corresponds to the provision of *dynamic query plans* in the Volcano query processing system [GW89]. That is, sometimes one cannot reliably make good choices among the alternative execution strategies for some subexpression of a query on the basis of the data available at optimization time; the best choice may depend on the amount of memory available when the query executes, and other factors that the optimizer cannot predict. In such cases the query plan may incorporate a special operator that allows the optimization decision to be made at execution time. Implementing this provision involves writing code to generate the special operator (in the optimizer) and additional code to recognize and interpret the operator (in the execution engine), as well as duplication of some code to perform cost-based decision-making, which occurs both in the optimizer and the execution engine. None of this extra coding is required if the generation of dynamic plans is achieved as a by-product of partial evaluation.

However, there are indeed respects in which combining the optimizer and execution engine is likely to be detrimental. One serious drawback of partial-evaluating a query evaluator as a substitute for an explicit optimization phase producing a query plan as output is that the result of partial evaluation (i.e., the residual) will potentially be a very large piece of code, in contrast to a query plan, which is usually very small.

3.3 Using Partial Evaluation for Cost Estimation

As noted above, partial evaluation does not appear to be well-suited to applying transformations on queries conditionally, depending on the estimated evaluation costs of the pre-transformation and post-transformation versions of the query. Thus, it is probably best to stick with an optimization model in which the transformation steps are under the control of an optimizer that computes costs and incorporates them into its search strategy. There remains, however, the possibility that partial evaluation could help with the cost estimation itself, by way of an intriguing if slightly outlandish implementation technique.

It should be borne in mind that cost estimation for query optimization

is a poorly developed art. Cost estimates produced and used by current optimizers are crude to say the least, but what is perhaps worse is that considerable coding effort is required to obtain even these crude estimates. Optimizer generators create optimizers automatically, given a file of transformation rules that the optimizer should consider when optimizing a query [GM93]. But optimizer generators also require the database implementor to supply *support functions* that are difficult to generate automatically. Among the support functions that must be coded by hand are cost estimation functions for each operator that may appear in query plans. These cost estimation functions, in turn, rely partly on *size* estimation functions that estimate the amount of data that each operator in a query plan will produce as output. The size estimation functions are also difficult to generate automatically, and so these, too, must be hand-coded by the database implementor.

Thus, it is certainly of interest to explore ways of automating the process of cost estimation as well as that of size estimation. One of the remarkable characteristics of partial evaluation is that it allows the insertion of extra layers of interpretation in a system for essentially no cost. So we imagine proceeding as follows. Rather than writing our query execution engine in C or some other conventional implementation language, we might write it in a language of our own devising—let us call this language BUNBURY [Wil95]. BUNBURY will have much the expressiveness of any other language, but we will keep it simple enough so that we may easily write an interpreter for it. Then by partial-evaluating the interpreter, taking our query execution engine as static data, we obtain object code for the execution engine that is essentially as good as compiled code. (It is as good *in principle*; if our object code is in T, it will have trouble competing with C code, but here we are more concerned with concepts than with short-term practicability.) So far we have not really changed anything—we have just added a layer of interpretation, which the partial-evaluator then obligingly removes again for us.

But now we are in a position to be devious in ways that were not possible before. Without changing the query execution engine, we may start making small changes to the BUNBURY interpreter that will point us in the direction of automatic cost estimation. The first step is to change the interpretation of disk reads and disk writes so that we not only perform these operations, but we also *count* how many of them we have done. (We could also count up approximately how much CPU time we use, if we wished to develop a more precise cost model.) We may provide a means for the count to be examined

on completion of query execution, so that executing a query supplies us not only with the query result, but also with the cost in disk I/O of executing the query. Thus, to estimate the cost of a query (or subquery), we may simply pass the query to the execution engine, discard the query result, and retain the computed cost. Needless to say, there is a defect in this technique: to estimate the cost of a query, we had to *execute* the query; obviously we do not want to do that.

The next step, then, is to further revise our interpreter so that when it interprets an I/O operation, it counts it but does not actually do the I/O. This change saves time, but introduces a new difficulty. Since the I/O is not being performed, we lack the information we need to proceed with the computation. For this reason another change is necessitated: the interpreter must operate on and produce some kind of *estimated value*—corresponding to the size estimates in conventional query optimization—in place of actual values. When I/O is supposed to be performed, a crude approximation to the data to have been fetched may be constructed on the basis of database statistics, and used in place of the actual data. When a loop is encountered, probably the best interpretation strategy is to estimate the number of loop iterations and the effect of each iteration, and to extrapolate the effects from these pieces of information. Note that the domains for estimated values will not necessarily be the same as the domains for the values that they estimate. For example, it would probably make most sense to estimate a Boolean using a real value in the range $[0, 1]$; in other words, the estimate for a Boolean value would be a *probability*.

The ideas we have suggested in this section are extremely speculative, and it is not clear that they are workable. It is also unclear whether the effort in coding an interpreter that performs abstract interpretation over a domain of costs and estimated values, as we have described, would be smaller than the effort in coding cost and size estimation support functions as required by existing optimizer generators. But the current technology is hardly satisfactory, and promises to become even more unwieldy when applied to object-oriented databases. It therefore seems worthwhile to look into even relatively bizarre alternatives.

4 Code-splitting and Specialization to Quasi-constants

The previous sections focused mainly on the application of partial evaluation to *relational* databases. Our database research interests, however, lie in the realm of object-oriented, rather than relational, databases. While partial evaluation of code in an object-oriented database bears some resemblance to "normal" partial evaluation, there are some striking differences. When translating database queries, unlike when compiling normal programming languages, we (database researchers) treat CPU as being a nearly free resource, and concentrate on reducing input/output activity. This perspective gives a very different slant on optimization. It also results in quite some confusion when language and database optimization people talk. There is also a real difference in the environment in which we process code.

In a normal programming language environment, most of the data is unknown. Partial evaluation is used to embed the data that *is* known throughout the code, and thus speed up the whole program. Often there is an initial step in partial evaluation, called "binding time analysis."[JSS89] This analysis takes facts about what data *will be known* and is meant to speed up the incorporation of data once that data is provided.

In a database, query optimization is usually performed with the data close at hand. Most of the data is known, but some of it may change before the query is run. We are dealing with a embarrassment of riches as far as obtaining sample values for particular expressions. At first glance, "binding time analysis" seems the exact opposite of what we want. It would appear that the appropriate kind of specification would simply indicate which information is not certain to "hold still," facts about what data *will possibly differ* when we actually execute the query.

Another place in which databases differ from languages is in their cost models. The time to access data in a database so far outweighs the time spent computing with the data that the cost of computation in query optimization is typically ignored. There is some justification for this attitude. Using extremely rough figures for current architectures, a disk takes ten milliseconds to retrieve data, while RAM takes a hundred nanoseconds. This is a difference of five orders of magnitude, perhaps sufficient justification for ignoring CPU costs.

While database researchers typically call this disk access I/O, there is a big difference between this I/O and the typically programming language

treatment of I/O. The I/O operations that compiler researchers talk about are not, typically, optional. In a "normal" programming language, I/O is treated as the border between the program and the world in which it operates. Since the compiler has no control over (and little understanding of) that world, the compiler cannot make choices about what I/O to perform, nor even when the I/O may be re-ordered. Within a database, the devices and/or files that constitute the database system are "owned" by the database. It is assumed that both read and write access to this store is the sole province of the database system, so I/O re-ordering (and even elimination) is fair game.

In the *Self* system,[CU89] very little is known until run time. This system does not even really have a user-visible type system to guide their code selection and compilation. The authors use a technique called "selective code splitting" to produce specialization points where there were none before. Basically, they insert a (cheap) test for the expected case, and then copy the ensuing code, once for the most common case, and once for the other cases. This allows their code to work for all cases (by using the unchanged original source), while working efficiently for the most common case (by using the specialized source). In the case of object-oriented code, the specialization can resolve implementation questions, thus not only eliminating the need for a run-time method lookup, but providing the opportunity to unfold method bodies in-line. Unlike many optimization techniques, this one may be freely applied to any piece of code. The worst it can do to code cost is add a small constant overhead for the tests (and increase code size). Recall that we have already examined some case introductions in the preceding sections when dealing with the Φ operator, the inserted cases provide more static information for the partial-evaluator to work with.

What do we hope to gain? In unfolding messages, we hope to expose the contained code, resolve message-to-method bindings, and, at least in some cases, remove some of the data fetches from the run-time expression. Remember, there is a substantial bonus for removing a data reference. If the result of sending a message to an object can be determined by partial evaluation, the substitution of the appropriate constant may mean a reduction in total I/O. Because of the huge ratio between simple calculation and disk access, we expect visible performance benefits from such optimizations.

If we can propagate constants far enough, we may be able to pre-compute our way through user-implemented indices in bulk data structures, or even perform some selects at query optimization time. We must place our emphasis on eliminating data fetches; it will do us little good simply to reduce

computation on data once retrieved. When we can discern the value of a boolean which controls a conditional statement, however, incorporating that value into the expression yields a simpler expression. This simplification may be necessary to produce effective code analysis.

This technique of case-insertion can work for us in other ways as well. Because our environment is loaded with data, and because we can look at data at query optimization time if we so desire, we can use the values from the database itself to help provide reasonable alternatives for case insertion.

5 Conclusions

We have seen how partial evaluation can be extremely effective in carrying out low-level optimizations in a database implementation, and have discussed ways in which it might be applied to a variety of problems in higher-level optimization of both relational and object-oriented databases. We have covered a mixture of techniques ranging from the experimentally proven to the wildly speculative. There is plenty of room for work at both ends of the spectrum. Despite the remarkable results obtained for low-level optimizations, it is not yet clear that they are applicable to real database systems; the gulf between a toy query evaluator and an actual system is too great to allow definite conclusions to be drawn. But the possibility of obtaining similar results in a larger, more credible context is certainly alluring. There are many exciting possibilities in the application of partial evaluation to database optimization, and they have only begun to be explored.

Acknowledgments

We are indebted to Charles Consel for his generous assistance and advice. Thanks also to David Burke, Luke Hornof, and Barbara Moura for answering questions, and to Mike Ashley and Julia Lawall for providing improved versions of Schism.

This work was supported in part by NSF grant IRI 91 18360.

References

[BD77] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[BTBN91] Val Breazu-Tannen, Peter Buneman, and Shamim Naqvi. Structural recursion as a query language. In Paris Kanellakis and Joachim W. Schmidt, editors, *Database Programming Languages: Bulk Types & Persistent Data, The Third International Workshop*, pages 9–19, Nafplion, Greece, August 1991. Morgan Kaufmann.

[CD93] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proceedings of the Twentieth Annual SIGACT Symposium on Principles of Programming Languages*. ACM, January 1993.

[Con90] Charles Consel. Binding time analysis for higher order untyped functional languages. In *1990 ACM Conference on Lisp and Functional Programming*, pages 264–272. ACM, 1990.

[Con92] Charles Consel. *Report on Schism '92*. Pacific Software Research Center, Oregon Graduate Institute of Science & Technology, October 1992. Draft.

[Con93] Charles Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In PEPM-93 [PEP93], pages 145–154.

[CU89] Craig Chambers and David Ungar. Customization: Optimizing compiler technology for SELF, a dynamically–typed object–oriented programming language. *SIGPLAN Notices*, 24(7):146–160, 1989.

[EN89] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, 1989.

[GM93] Goetz Graefe and William J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the IEEE Conference on Data Engineering*, pages 209–218, Vienna, Austria, April 1993.

[GW89] Goetz Graefe and Karen Ward. Dynamic query evaluation plans. In James Clifford, Bruce Lindsay, and David Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, pages 358–366, Portland, Oregon, June 1989.

[JSS89] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, February 1989.

[Law93] Julia L. Lawall. Proofs by structural induction using partial evaluation. In PEPM-93 [PEP93], pages 155–166.

[PEP93] *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Copenhagen, Denmark, June 1993.

Appendix A — The Program

```
;;; Data Structures: (Types have an initial capital letter)
;  Schema =          List( Relation-Descriptor )
;  Relations =       List( Relation )
;  Query =  (select  Predicate Query)  |  (join  Query Query)
;        |  (project Attribute-List Query)  |  (Relation-Name)
;  Labeled-Relation= Labeled-Relation(Attribute-List Relation)
;  Relation-Descriptor =  ( Relation-Name Attribute-List )
;  Relation-Name =   Symbol
;  Attribute-List =  List( Attribute-Name )
;  Attribute-Name =  Symbol
;  Relation =        List( Row )
;  Row =             List( Attribute-Val )
;  Attribute-Val =   Value
;  Predicate =       Triple( "=" Attribute-Name Value )

; main: Schema x Relations x Query -> Relation
;                          Just pass things along to real code
(define (main schema relations query)
    (db schema relations query
    ))

; db: Schema x Relations x Query -> Relation
;                              Decompose query and evaluate ops
(define (db schema relations query)
   (if (atom? query)
       (findRelation query schema relations)
       (case (car query)
         [(join)    (db-join (cadr query)
                         (db schema relations (caddr query))
                         (db schema relations (cadddr query)))]
         [(select)  (db-select (cadr query)
                         (db schema relations (caddr query)))]
         [(project) (db-project (cadr query)
                         (db schema relations (caddr query)))]
         [else      '((bad query operator))]
       )
   ))
```

```
;findRelation: Symbol x Schema x Relations -> Labeled-Relation
(define (findRelation Relation-name schema relations)
  (if (null? schema)
     (Labeled-Relation '(bogus-labels) Relations)
     (if (equal? Relation-name (fst (hd schema)))
       (Labeled-Relation (snd (hd schema)) (hd relations))
       (findRelation Relation-name (tl schema) (tl relations))
     )
  ))

; db-select : Predicate x labeled-relation -> labeled-relation
(define (db-select pred lrel)
    (let ([(Labeled-Relation labels rows) lrel])

        (Labeled-Relation labels
                          (choose-rows pred labels rows '()))
    ))

;choose-rows: Predicate x Attribute-List x Rows x Rows -> Rows
(define (choose-rows pred labels rows accumulator)
    (if (null? rows)
        accumulator
        (choose-rows pred labels (tl rows)
                (if (pred-holds? pred labels (hd rows))
                        (cons (hd rows) accumulator)
                        accumulator)
        )
    ))

; pred-holds? : Predicate x Attribute-List x Row -> Boolean
(define (pred-holds? pred labels row)
   (equal? (caddr pred) (get-attribute (cadr pred) labels row)
   ))


; find-attribute : Name x Attribute-List x Row -> Value
;        Retrieve one column element ('no-such-attribute
;                                     if not in Attr-List)
(define (find-attribute name labels row)
```

```
    (if (null? labels)
        'no-such-attribute
        (if (equal? name (hd labels))
            (hd row)
            (find-attribute name (tl labels) (tl row)))
    ))

;db-project:List(attrs) x Labeled-Relation -> Labeled-relation
(define (db-project attrs lrel)
   (let ([(Labeled-Relation labels rows) lrel])

     (Labeled-Relation attrs (project-rows attrs labels rows))
   ))

; project-rows : List(attrs) x relation-tags x List(rows) ->
;                                              List(narrow-rows)
(define (project-rows attrs labels rows)
   (if (null? rows)
        '()
        (cons (trim-row attrs labels (hd rows))
            (project-rows attrs labels (tl rows)))
   ))

; trim-row : List(attrs) x relation-tags x row -> narrow-row
(define (trim-row attrs labels row)
   (if (null? attrs)
       '()
       (cons (get-attribute (hd attrs) labels row)
            (trim-row (tl attrs) labels row))
   ))

; db-join : join-pred x Labeled-relation x Labeled-relation ->
;                                         Labeled-relation
; join-pred = label x label
; that is, join-pred combines label from first relation and
; label from second relation, and specifies an equijoin.
; This implementation works as a natural join as well, i.e.,
; redundant attribute names in the joined tuples are removed
; (which of course is ; a bad thing if the join was not on
```

```
; the attribute in question...).
(define (db-join join-pred lrel1 lrel2)
  (let ([(Labeled-Relation labels1 rows1) lrel1]
        [(Labeled-Relation labels2 rows2) lrel2])
     (Labeled-Relation
       (ordered-set-union labels1 labels2)
       (outer-loop join-pred labels1 labels2 rows1 rows2 '())))
  ))

; outer-loop : join-pred x Attribute-List x Attribute-List x
;                   Relation x Relation x Relation -> Relation
; Does outer loop of the join --- a row at a time from rows1
(define (outer-loop join-pred attrs1 attrs2 rows1 rows2 accum)
    (if (null? rows1)
        accum
        (outer-loop join-pred attrs1 attrs2 (tl rows1) rows2
             (inner-loop join-pred attrs1 attrs2 (hd rows1)
                                                 rows2 accum))
    ))

; inner-loop : join-pred x Attribute-List x Attribute-List x
;                         Row x Relation x Relation -> Relation
; Does the inner loop of the join -- processing one row of the
; outer relation
;      Accomplished by looping a row at a time through rows2
(define (inner-loop join-pred attrs1 attrs2 row1 rows2 accum)
  (if (null? rows2)
     accum
     (inner-loop join-pred attrs1 attrs2 row1 (tl rows2)
        (match join-pred attrs1 attrs2 row1 (hd rows2) accum))
  ))

; match : join-pred x Attribute-List x Attribute-List x Row x
;                                     Row x Relation -> Relation
; The real guts of a join.  Combines two rows of a cross-
; product to add to a relation (or not).
(define (match join-pred attrs1 attrs2 row1 row2 accum)
    (if (equal? (get-attribute (car join-pred) attrs1 row1)
                (get-attribute (cadr join-pred) attrs2 row2))
```

```
        (cons (trim-row (ordered-set-union attrs1 attrs2)
                        (vdappend attrs1 attrs1 attrs2)
                        (vdappend attrs1 row1 row2))
            accum)
        accum
    ))

; ordered-set-union : Attribute-List x Attribute-List ->
;                                                 Attribute-List
; Combines a pair of attribute lists into the attribute list
; for a join of the two described relations.
(define (ordered-set-union set1 set2)
        (rev (back-ordered-set-union set2 (rev set1)))
    )

; back-ordered-set-union : Attribute-List x Attribute-List ->
;                                                 Attribute-List
; auxilliary function to ordered-set-union.  produces a result
;reversed in attribute order of its first arg.
(define (back-ordered-set-union set1 set2)
    (if (null? set1)
        set2
        (back-ordered-set-union (tl set1)
                (if (mem? (hd set1) set2)
                    set2
                    (cons (hd set1) set2)))
    ))

; vdappend: Attribute-List x Row x Other-Row -> Extended-Row
; vdappend -- Scott & Bennet's special append.  The Attribute-
; List describes the first Row.  This was needed to get 'nth's
; in result tail
(define (vdappend first-shape first-row second-row)
   (if (null? first-shape) second-row
      (cons (hd first-row)
        (vdappend (tl first-shape) (tl first-row) second-row))
   ))
```

Appendix B — Experiment 1: Fully Static Data

```
>> (schema, data, query) all static for Partial Evaluation
```

```
(  (employee     (fname minit lname ssn bdate addr sex
                                          salary superssn dno))
   (department   (dname dnumber mgrssn mgrstartdate))
   (dept_locations   (dnumber dlocation))
   (product      (pname pnumber plocation dnum))
   (works_on     (essn pno hours))
   (dependent    (essn dependent_name sex bdate relationship)))

( (; employee
    (John     B Smith   123456789 09-JAN-55
               "731 Fondren, Houston, TX" M 30000 333445555 5)
    (Franklin T Wong    333445555 08-DEC-45
              "638 Voss, Houston, TX"    M 40000 888665555 5 )
    (Alicia   J Zelaya  999887777 19-JUL-58
              "3321 Castle, Spring, TX"  F 25000 987654321 4 )
    (Jennifer S Wallace 987654321 20-JUN-31
              "291 Berry, Bellaire, TX"  F 43000 888665555 4 )
    (Ramesh   K Narayan 666884444 15-SEP-52
              "97 Fire Oak, Humble, TX"  M 38000 333445555 5 )
    (Joyce    A English 453453453 31-JUL-62
              "5631 Rice, Houston, TX"   F 25000 333445555 5 )
    (Ahmad    V Jabbar  987987987 29-MAR-59
              "980 Dallas, Houston, TX"  M 25000 987654321 4 )
    (James    E Borg    888665555 10-NOV-27
              "450 Stone, Houston, TX"   M 55000 null      1))

   (; department
        (Research         5 333445555 22-MAY-78)
        (Administration   4 987654321 01-JAN-85)
        (Headquarters     4 987654321 01-JAN-85))

   (; dept_locations
        (1 Houston)   (4 Stafford)   (5 Bellaire)
        (5 Sugarland) (5 Houston))
```

```
   (; product
        (ProductX         1 Bellaire    5)
        (ProductY         2 Sugarland   5)
        (ProductZ         3 Houston     5)
        (Computerization 10 Stafford    4)
        (Reorganization  20 Houston     1)
        (Newbenefits     30 Stafford    4))

  (; works_on
   (123456789  1 32.5) (123456789  2  7.5) (666884444  3 40.0)
   (123456789  1 32.5) (123456789  2  7.5) (666884444  3 40.0)
   (453453453  1 20.0) (453453453  2 20.0) (333445555  2 10.0)
   (333445555  3 10.0) (333445555 10 10.0) (333445555 20 10.0)
   (999887777 30 30.0) (999887777 10 10.0) (987987987 10 35.0)
   (987987987 30  5.0) (987654321 30 20.0) (987654321 20 15.0)
   (888665555 20 null))

  (; dependent
        (333445555 Alice        F 05-APR-76 daughter)
        (333445555 Theodore     M 25-OCT-73 son)
        (333445555 Joy          F 03-MAY-48 spouse)
        (987654321 Abner        M 29-FEB-32 spouse)
        (123456789 Michael      M 01-JAN-78 son)
        (123456789 Alice        F 31-DEC-78 daughter)
        (123456789 Elizabeth    F 05-MAY-57 spouse)))

(select (= dno 4)
        (project (fname dno)
                 (select (= sex f)
                         (join (essn ssn) works_on employee))))
```

```
>> Sugared Residual From Partial Evaluator (answer here)

   ((define (main.1)
       '((fname dno)
         ((jennifer 4) (jennifer 4) (alicia 4) (alicia 4)))))
```

Appendix C — Experiment 2: Relations Dynamic

```
>> (schema, query) static, data left dynamic for Partial Eval
>>                 Note: employee relation moved down in list

(  (department  (dname dnumber mgrssn mgrstartdate))
   (dept_locations  (dnumber dlocation))
   (product     (pname pnumber plocation dnum))
   (employee    (fname minit lname ssn bdate addr sex salary
                                                superssn dno))
   (works_on    (essn pno hours))
   (dependent   (essn dependent_name sex bdate relationship)))

DYNAMIC

(select (= sex f) employee)
```

```
>> Sugared Residual Produced by Partial Evaluator
>> (Attribute list known, all db accesses positional)

   (define (main.1 relations)
       (list '(fname minit lname ssn bdate addr sex salary
                                                superssn dno)
             (choose-rows.2
                (nth (list '(fname minit lname ssn bdate addr
                                        sex salary superssn dno)
                           (nth relations 3))
                     1)
                '())))

    (define (choose-rows.2 rows accumulator)
       (if (null? rows)
           accumulator
           (choose-rows.2
              (cdr rows)
              (if (equal? 'f (nth (car rows) 6))
                  (cons (car rows) accumulator)
                  accumulator))))
```

Appendix D — Experiment 3: Complex Query

```
>> (schema, query) static, data left dynamic for Partial Eval

(  (employee    (fname minit lname ssn bdate addr sex
                                          salary superssn dno))
   (department  (dname dnumber mgrssn mgrstartdate))
   (dept_locations   (dnumber dlocation))
   (product     (pname pnumber plocation dnum))
   (works_on    (essn pno hours))
   (dependent   (essn dependent_name sex bdate relationship)))

(  (department  (dname dnumber mgrssn mgrstartdate))
   (dept_locations  (dnumber dlocation))
   (product     (pname pnumber plocation dnum))
   (employee    (fname minit lname ssn bdate addr sex salary
                                                 superssn dno))
   (works_on    (essn pno hours))
   (dependent   (essn dependent_name sex bdate relationship)))

DYNAMIC

(select (= dno 4) (project (fname dno) (select (= sex f)
        (join (essn ssn) works_on employee))))
```

```
>> Sugared Residual Produced by Partial Evaluator
>> (Attribute list known, all db accesses positional,
>> Query structure discernible)

   (define (main.1 relations)
       (labeled-relation
          '(fname dno)
          (choose-rows.6
             (project-rows.5
                (choose-rows.4
                   (outer-loop.2 (nth relations 4)
                                 (car relations)
                                 '()))
```

```
              '()))
         '())))

(define (choose-rows.4 rows accumulator)
   (if (null? rows)
       accumulator
       (choose-rows.4
          (cdr rows)
          (if (equal? 'f (nth (car rows) 9))
              (cons (car rows) accumulator)
              accumulator))))

(define (choose-rows.6 rows accumulator)
   (if (null? rows)
       accumulator
       (choose-rows.6
          (cdr rows)
          (if (equal? '4 (nth (car rows) 1))
              (cons (car rows) accumulator)
              accumulator))))

(define (project-rows.5 rows)
   (if (null? rows)
       '()
       (cons (list (nth (car rows) 3) (nth (car rows) 12))
             (project-rows.5 (cdr rows)))))

(define (outer-loop.2 rows1 rows2 accum)
   (if (null? rows1)
       accum
       (outer-loop.2
          (cdr rows1)
          rows2
          (inner-loop.3 (car rows1) rows2 accum))))

(define (inner-loop.3 row1 rows2 accum)
   (if (null? rows2)
       accum
       (inner-loop.3
```

```
row1
(cdr rows2)
(if (equal? (car row1) (nth (car rows2) 3))
    (cons (list (car row1)
                (nth row1 1)
                (nth row1 2)
                (car (car rows2))
                (nth (car rows2) 1)
                (nth (car rows2) 2)
                (nth (car rows2) 3)
                (nth (car rows2) 4)
                (nth (car rows2) 5)
                (nth (car rows2) 6)
                (nth (car rows2) 7)
                (nth (car rows2) 8)
                (nth (car rows2) 9))
          accum)
    accum))))
```