A Framework for Implementing Hypothetical Queries

Timothy Griffin
Bell Laboratories, Lucent Technologies
griffin@research.bell-labs.com

Richard Hull
Bell Laboratories, Lucent Technologies
hull@research.bell-labs.com

Abstract

Previous approaches to supporting hypothetical queries have been "eager": some representation of the hypothetical state (or the corresponding delta) is materialized, and query evaluation is filtered through that representation. This paper develops a framework for evaluating hypothetical queries using a "lazy" approach, or using a hybrid of eager and lazy approaches.

We focus on queries having the form "Q when $\{\!\{U\}\!\}$" where Q is a relational algebra query and U is an update expression. The value assigned to this query in state DB is the value that Q would return in the state resulting from executing U on DB. Nesting of the keyword when is permitted, and U may involve a sequence of several atomic updates.

We present an equational theory for queries involving when that can be used as a basis for optimization. This theory is very different from traditional rules for the relational algebra, because the semantics of when is unlike the semantics of the algebra operators. Our theory is based on the observation that hypothetical states can be represented as substitutions, similar to those arising in functional and logic programming. Furthermore, hypothetical queries of the form Q when $\{\!\{U\}\!\}$ can be thought of as representing the *suspended* application of a substitution. Using the equational theory we develop an approach to optimizing the evaluation of hypothetical queries that uses deltas in the sense of Heraclitus, and permits a range of evaluation strategies from lazy to eager.

1 Introduction

The need for accessing hypothetical database states arises in many application areas. This includes version management, decision support, active databases (where rules may access the deltas and potential future states specified by proposed updates), and integrity maintenance.

We focus on hypothetical queries with form Q when $\{\!\{U\}\!\}$ where Q is essentially a relational algebra query (which may involve nested when's) and $\{\!\{U\}\!\}$ represents a *hypothetical state*. Here U is an update expression, expressed in a language that mimics the update expressions of SQL, which may include a sequence of atomic updates. The value assigned to this query in state DB is the value that Q would return in the state that is reached from DB by executing update U. Such queries are fundamental to all applications that involve hypothetical database states.

Previous approaches to supporting queries against hypothetical states have been *eager*: some representation of the hypothetical physical state (or the corresponding delta) is materialized, and then query evaluation is "filtered" through that materialized representation. This includes version management systems, the Heraclitus database programming language prototypes [GHJ96, DHR96], and implementations of hypothetical relations (e.g., see [WS83]). The eager approach makes sense for applications where many queries are made against a single hypothetical state.

However, effective optimization of queries requires the development of a rich, systematic space of evaluation strategies. The first of two contributions of this paper is to present a wide spectrum of evaluation strategies for hypothetical queries, ranging from *eager* evaluation of hypothetical states to *lazy* evaluation of hypothetical states. An evaluation strategy for a hypothetical query, Q when $\{\!\{U\}\!\}$, is *lazy* if it first reformulates Q when $\{\!\{U\}\!\}$ into an equivalent, non-hypothetical query Q', and then evaluates Q' using conventional techniques. Our framework provides a means of moving along the entire spectrum between eager and lazy evaluation, allowing evaluation strategies that are eager, lazy, or hybrid.

The main difficulty encountered in developing this framework was the lack of an equational theory to bridge the gap between the relational algebra and expressions involving hypothetical states. The second contribution of this paper is to fill this gap. The central observation underlying our theory is that hypothetical states can be represented as *substitutions*, similar to those arising in functional and logic programming. Using this insight, we construct a Hypothetical Query Language, called HQL, that incorporates substitutions as *explicit syntactic constructs*. These provide an intermediate representation of hypothetical update expressions, that can be manipulated without going outside of the language.

We now briefly describe the language HQL. Hypothetical queries are generalized expressions of the form Q when η, where η is a hypothetical-state expression that can be an update expression, $\{\!\{U\}\!\}$; an explicit substitution of the form $\{\!\{Q_1/R_1, \ldots, Q_n/R_n\}\!\}$ (which means, intuitively, that relation R_i is to be replaced by the query Q_i); or an expression constructed by an operation that composes substitutions.

Permission to make digital/hard copy of part or all this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD '97 AZ,USA
© 1997 ACM 0-89791-911-4/97/0005...$3.50

The construct Q when η can be viewed as the *suspended* application of the substitution represented by η. An equational theory for HQL can be developed as a combination of (1) a conventional equational theory for the relational algebra, (2) an equational theory of explicit substitutions, and (3) a set of equations that relate hypothetical states of the form $\{\!|U|\!\}$ to explicit substitutions.

The fully lazy evaluation strategy is achieved by transforming each hypothetical-state expression into an equivalent explicit substitution, and then applying these substitutions to queries in their scopes, to obtain a pure relational algebra query. The fully eager evaluation strategy is achieved by transforming each hypothetical-state expression into an explicit substitution, materializing a full or partial representation of this substitution and storing it in an environment, and filtering the evaluation of queries in its scope through this environment. Hybrid strategies arise by combining these two approaches in the evaluation of a hypothetical query. The framework can be used in conjunction with existing techniques for the physical representation of hypothetical states, e.g., the deltas of the Heraclitus languages.

The framework developed here can provide the basis for a full optimizer of hypothetical queries. We leave, as future work, the development of cost estimation techniques for execution plans of hypothetical queries, and heuristics for reducing the space of reasonable plans.

Related work. The framework developed here is inspired in part by the Heraclitus languages, which support hypothetical queries and the keyword when. HQL is more general because hypothetical-state expressions do not have to be implemented using deltas. Furthermore, the use of explicit substitutions reveals optimization opportunities not previously explored.

Explicit substitutions were first presented in [ACCL91] where they are used to model λ-reduction and to serve as a formal intermediate language between functional programming languages and closure-based implementations. Another interesting application of explicit substitutions is the reduction of higher-order unification to first-order equational unification [DHK95]. For a survey of calculi of explicit substitutions for λ-terms, see [Les94].

Hypothetical queries are closely related to the familiar notion of *weakest preconditions* [Dij75, Dij76, Gri81]. In the database context, a formula β is a *weakest pre-condition* for formula α with respect to update U if for each database state DB, β holds in DB if and only if α holds in the result of applying U to DB. Several algorithms for constructing preconditions β have been presented (see for example, [Qia90, Qia91]) that rewrite α based on the form of U. Evaluating β corresponds to the lazy strategy in the sense that no hypothetical states are materialized. In our approach, β is simply expressed as α when $\{\!|U|\!\}$, and we are then free to rewrite this formula into any number of forms, for either lazy, eager, or hybrid evaluation.

Hypothetical queries have been studied in the logic programming community, e.g., see [GR84, GR85], and the datalog community, e.g., Hypothetical Datalog [Bon90, Bon95]. In the database context, the combination of recursion and hypotheticals can lead to an increase in expressive power and/or data complexity. For example, [Bon90] shows that datalog (without negation) extended by hypothetical insertions and deletions has EXPTIME data complexity, an increase over the PTIME data complexity of datalog. The proofs of some of these complexity results use algorithms that involve eager, lazy, and hybrid evaluation of hypothetical states in the course of query evaluation. However, the emphasis of this work is on expressiveness and complexity, rather than optimization.

Outline. Section 2 introduces examples that motivate the need for an equational theory of hypothetical queries. Sections 3 and 4 develop the formal basis for our work. In particular, a simple language called HQL is introduced. This language includes constructs for building and manipulating hypothetical states, and for accessing them using when. The connection between hypothetical state expressions and substitutions is worked out in detail. A key result in this development gives the mapping of hypothetical queries into pure relational algebra queries. This mapping provides one method for evaluating HQL queries — translate HQL queries to the relational algebra and then use standard optimization techniques.

Section 5 develops a more general framework for optimizing and evaluating hypothetical queries, that incorporates both eager and lazy evaluation of hypothetical-state expressions. This is based on an equational theory of explicit substitutions that can be used together a conventional equational theory for the relational algebra to rewrite hypothetical queries into an optimized form. Section 6 concludes with general research questions.

Due to space limitations, the exposition here is very terse. More details, and proofs of theoretical results, may be found in [GH97].

2 Simple query and update languages

This section presents several examples illustrating different strategies for evaluating hypothetical queries, and indicating how our approach provides a unifying framework for generating these strategies.

Example 2.1 illustrates the traditional "eager" approach to processing hypothetical queries, a "lazy" approach based on the use of substitutions, and a hybrid in between these. Examples 2.2 and 2.3 show useful optimizations stemming from the perspective of substitutions. Finally, Example 2.4 shows one pitfall of the purely lazy approach.

Example 2.1: [Hypothetical queries using alternatives] Consider an application in which queries against many hypothetical states are to be considered and compared. A tree of potential updates might be constructed, where each edge is labeled with an hypothetical update expression. A node v in the tree corresponds to the hypothetical state produced by applying all of the hypothetical update expressions in the path from the root to node v. One might ask the query

$$Q' \equiv ((Q_1 \text{ when } \eta_1) - (Q_2 \text{ when } \eta_2)) \text{ when } \eta_3$$

where Q_1, Q_2 are relational algebra queries, η_3 is the hypothetical update expression associated with a path from the root to some node v, and η_1 and η_2 are the hypothetical update expressions associated with two possible extensions of this path. Intuitively, Q' is asking for the difference of the value of Q_1 under hypothetical update η_1 vs. the value of Q_2 under hypothetical update η_2, assuming already the hypothetical update η_3.

(a) [Eager approach] Most of the existing literature on hypothetical queries advocates an "eager" approach to materializing hypothetical states. The basic idea is to evaluate expressions of form Q when η in two steps: (i) (part of) the hypothetical state specified by η is materialized, and (ii) the evaluation of Q is "filtered" through it. If nested when's are

present, then we must be careful about (iii) how the evaluation steps are ordered. We illustrate in terms of the prototype implementation of the relational Heraclitus described in [GHJ96]; a similar approach is taken with hypothetical relations (e.g., [WS83]).

With regards to (i), Heraclitus uses deltas to represent partially materialized hypothetical states. A delta holds the sets of tuples to be inserted and deleted from each relation affected by the hypothetical state. For (ii), special-purpose generalizations of the relational algebra operators are used to filter query evaluation against a delta (see Subsection 5.5).

We now consider (iii). The Heraclitus implementation uses a "run-time when stack". At any point in the computation, the top of this stack essentially holds the materialized delta reflecting the hypothetical state that is currently being used for filtering. We briefly indicate how this is used to evaluate Q' above. First, a delta Δ_3 corresponding to η_3 is computed. Then a delta Δ_1 corresponding to η_1 is computed, by evaluating η_1 filtered against the current top of the stack. A new delta Δ_1' is formed, that captures the net effect of Δ_3 and Δ_1, and pushed onto the stack. Now Q_1 is evaluated, filtered against the top of the stack. The stack is popped and again pushed for evaluation of Q_2, and finally the results of the two queries are differenced.

In general, a materialized representation of a hypothetical state will become out-of-date if the underlying database state is modified.

(b) [Lazy approach] An alternative approach, which is *lazy* with regards to the evaluation of hypothetical states, is to build relational algebra queries

$$\begin{array}{rcl} Q_1' & \equiv & (Q_1 \text{ when } \eta_1) \text{ when } \eta_3 \\ Q_2' & \equiv & (Q_2 \text{ when } \eta_2) \text{ when } \eta_3 \end{array}$$

and then evaluate $Q_1' - Q_2'$. The queries Q_1' and Q_2' can be found using the framework developed in this paper.

We illustrate this using the following query, where R and S have the same arity, $Q_1 = Q_2 = R \bowtie S$, and where the join condition is not specified.

$$\begin{array}{l} [\ ((R \bowtie S) \text{ when } \{\!\{ins(R, \sigma_{A>30}(S))\}\!\}) - \\ \quad ((R \bowtie S) \text{ when } \{\!\{ins(R, \sigma_{A>50}(S))\}\!\})\ ] \\ \qquad \text{when } \{\!\{del(S, \sigma_{A<60}(S))\}\!\} \end{array} \quad (1)$$

Note that in the absence of the outer hypothetical update, we would expect the query to yield the non-empty result $\sigma_{30<A\leq 50}(S) \bowtie S$.

We consider now the construction of Q_1'. We begin with the query

$$\begin{array}{l} ((R \bowtie S) \text{ when } \{\!\{ins(R, \sigma_{A>30}(S))\}\!\}) \\ \quad \text{when } \{\!\{del(S, \sigma_{A<60}(S))\}\!\}. \end{array}$$

We replace the two hypothetical-state expressions using "explicit substitutions" (see Section 3).

$$\begin{array}{l} ((R \bowtie S) \text{ when } \{\!\{R \cup \sigma_{A>30}(S)/R\}\!\}) \\ \quad \text{when } \{\!\{S - \sigma_{A<60}(S)/S\}\!\}. \end{array}$$

We now "apply" the inner substitution to the query:

$$((R \cup \sigma_{A>30}(S)) \bowtie S) \text{ when } \{\!\{S - \sigma_{A<60}(S)/S\}\!\}.$$

Applying the second substitution we have

$$\begin{array}{rcl} & & (R \cup \sigma_{A>30}(S - \sigma_{A<60}(S))) \bowtie (S - \sigma_{A<60}(S)) \\ & \equiv & (R \cup \sigma_{A>30}(\sigma_{A\geq 60}(S))) \bowtie (\sigma_{A\geq 60}(S)) \\ & \equiv & (R \cup \sigma_{A\geq 60}(S)) \bowtie (\sigma_{A\geq 60}(S)) \end{array}$$

Analogously, query Q_2' can be transformed into

$$(R \cup \sigma_{A\geq 60}(S)) \bowtie (\sigma_{A\geq 60}(S)).$$

It follows that the overall query (1) is equivalent to the empty query ϕ. Importantly, this analysis can be performed without reference to the underlying database state.

(c) [Hybrid] In some cases, hybrids of the eager and lazy approaches may also useful. Suppose in query Q' that the hypothetical state of η_3 gives new values for relations R and S, and that the subqueries Q_1 and Q_2 involve many occurrences of R and S. Suppose further that, for $i = 1, 2$, the relation names affected by η_i occur only once or twice in Q_i. In this case it may make sense to use an evaluation strategy guided by

$$(Q_1'' - Q_2'') \text{ when } \eta_3,$$

where $Q_i'' \equiv (Q_i \text{ when } \eta_i)$. In this case, η_3 is evaluated eagerly, and η_1 and η_2 are evaluated lazily. □

The next two examples describe optimizations based on the fact that hypothetical states can be viewed as substitutions. The first uses the classical operation of composing substitutions.

Example 2.2: [Families of hypothetical queries] In some applications multiple queries might be asked against the same hypothetical state.

(a) [Using composition] As a simple illustration, suppose that

$$\begin{array}{l} (\widehat{Q} \text{ when } \{\!\{ins(R, \sigma_{A>30}(S))\}\!\}) \\ \quad \text{when } \{\!\{del(S, \sigma_{A<60}(S))\}\!\} \end{array}$$

will be asked for many queries $\widehat{Q}$.

As in Example 1(b), we could evaluate such queries by converting to explicit substitutions

$$\begin{array}{l} (\widehat{Q} \text{ when } \{\!\{R \cup \sigma_{A>30}(S)/R\}\!\}) \\ \quad \text{when } \{\!\{S - \sigma_{A<60}(S)/S\}\!\} \end{array}$$

and then applying the two substitutions into $\widehat{Q}$. However, it might reduce work to compute the *composition* of the two substitutions, and use that composition repeatedly. As developed in Sections 3 and 5 below, we continue the transformation by first using '#' to represent the composition of hypothetical-state expressions,

$$\widehat{Q} \text{ when } [\ \{\!\{S - \sigma_{A<60}(S)/S\}\!\} \# \{\!\{R \cup \sigma_{A>30}(S)/R\}\!\}\ ],$$

then expanding the definition of composition to obtain

$$\widehat{Q} \text{ when } \{\!\{\sigma_{A\geq 60}(S)/S,\ R \cup \sigma_{A>30}(S - \sigma_{A<60}(S))/R\}\!\},$$

and finally using algebraic simplification to arrive at

$$\widehat{Q} \text{ when } \{\!\{\sigma_{A\geq 60}(S)/S,\ R \cup \sigma_{A\geq 60}(S)/R\}\!\}.$$

Under a lazy strategy, the new substitution can be applied to each of the queries $\widehat{Q}$. This substitution remains valid even if the underlying database state is changed. Under the eager strategy, the substitution can be (partially) materialized, and used to filter evaluation of the $\widehat{Q}$'s.

(b) [Another hybrid] To generalize a bit, suppose now that in a single state DB the expression

$$((\widehat{Q_1} \text{ when } \eta_1) - (\widehat{Q_2} \text{ when } \widehat{\eta_2})) \text{ when } \eta_3,$$

is to be evaluated with many different values for $\widehat{Q_1}$, $\widehat{Q_2}$, and $\widehat{\eta_2}$. In this case it might make sense to materialize part of η_3 for use with $\widehat{Q_1}$ when $\widehat{\eta_1}$, and to materialize part of $\eta_3 \# \eta_2$ for use with $\widehat{Q_2}$. □

The next example uses the classical technique of binding removal in substitutions.

Example 2.3: [Binding removal] Suppose that many queries of the form

$$\widehat{Q} \text{ when } \{\!\{ ins(R, \sigma_p(S)); del(S, \sigma_q(R)); ins(T, \pi_{\vec{A}} R) \}\!\}$$

are to be answered for different $\widehat{Q}$'s. Direct application of the technique of Example 2.2 suggests that the substitution

$$\{R \cup \sigma_p(S)/R\} \# \{S - \sigma_q(R)/S\} \# \{T \cup \pi_{\vec{A}}(R)/T\}$$

should be computed. Suppose however that none of the queries $\widehat{Q}$ will involve relation name S. This permits us to avoid computation of the full composition. In particular, the middle substitution can be dropped from consideration. This stems from a classical equivalence for substitutions, namely

$$sub(E, \sigma) \equiv sub(E, [\sigma - \{t/v\}])$$

where σ is a substitution, t/v is a binding in σ, and variable v is not free in expression E.

In the case of eager evaluation this will reduce work on the underlying data, and in the case of lazy evaluation this will reduce work in the optimizer. □

The final example shows that transforming to a fully lazy equivalent of an hypothetical query Q can lead to an exponential blow-up. This is one motivation for exploring both lazy and eager forms of evaluation of hypothetical queries. We also illustrate two ways that the blow-up may be avoided in practice.

Example 2.4: Consider the query

$$\begin{aligned} Q \;=\; & ((\ldots((\ R_0 \text{ when } \{\!\{ E_1(R_1)/R_0 \}\!\}) \\ & \quad \text{when } \{\!\{ E_2(R_2)/R_1 \}\!\}) \\ & \quad\quad \text{when } \cdots) \\ & \quad\quad\quad \text{when } \{\!\{ E_n(R_n)/R_{n-1} \}\!\}) \end{aligned}$$

where for each i, $E_i(R_i)$ is a relational algebra query whose only relation name is R_i. (In the examples below, the intended arities of relations can be inferred from the context.)

Note that expression 'Q' is linear in n, and that the fully lazy equivalent of Q is

$$\widehat{Q} \stackrel{\text{def}}{=} E_1(E_2(\ldots(E_n(R_n))\ldots)).$$

(a) [Exponential blow-up] Suppose that expression $E_i(R_i)$ is $R_i \times R_i$ for each i. Then $\widehat{Q}$ is exponential in n. Also, if the relations R_i are non-empty, then the value of $red(Q)$ is also exponential in n.

(b) [Relational algebra rewriting can help] Suppose that $E_i(R_i) = R_i \times R_i$ for each i except for one j, where $E_j(R_j)$ is $R_j - R_j$. Then the size of the query $\widehat{Q}$ is exponential in n, but its value is ϕ (the empty query). This equivalence can be found using the equational theory developed in Section 5. In particular, an algebra rewriting rule can be used to replace $\{\!\{ R_j - R_j/R_{j_1} \}\!\}$ by $\{\!\{ \phi/R_{j_1} \}\!\}$. Next, several applications of binding-removal (as in Example 2.3) will yield

$$\begin{aligned} Q \;\equiv\; & ((\ldots((\ R_0 \text{ when } \{\!\{ E_1(R_1)/R_0 \}\!\}) \\ & \quad \text{when } \{\!\{ E_2(R_2)/R_1 \}\!\}) \\ & \quad\quad \text{when } \cdots) \\ & \quad\quad\quad \text{when } \{\!\{ \phi/R_{j-1} \}\!\}) \end{aligned}$$

Now a series of compositions and binding-removals can now be used to obtain

$$\ldots \equiv R_0 \text{ when } \{\!\{ \phi/R_0 \}\!\} \equiv \phi$$

(c) [Eager evaluation may win] Suppose that each E_i has the form

$$\pi_{A_0 \cdots A_n}(\sigma_{A_{i-1} < 0} R_i) \cap \pi_{A_0 \cdots A_n}(\sigma_{A_{i-1} > 0} R_i)$$

If these intersections are small, then an eager evaluation strategy will be most efficient. □

3 A Language for Simple Hypothetical Queries

As mentioned, a key observation of this paper is that hypothetical states can be represented as substitutions. In this section, we introduce a very simple language of hypothetical states for the purpose of making this connection precise. The main points we want to develop here are that (1) substitutions can be treated as updates, that (2) updates can be treated as substitutions, and that (3) all instances of when can be eliminated by application of substitutions. In the next section, we will apply these ideas to the design of a more general language for hypothetical queries.

3.1 Syntax and Semantics

A *database schema* is a collection of relation names $\mathcal{Z} = \{Z_1, \ldots, Z_z\}$, each of a fixed arity. We assume that a database schema $\mathcal{Z}$ is fixed for this discussion. We will use R, R_j, S, S_j to range over the names of $\mathcal{Z}$.

We will define a language with queries Q, updates U, and hypothetical queries H. Hypothetical queries $H \in \mathcal{H}$ are defined by the grammar

$$H \;::=\; Q \text{ when } \{\!\{ U \}\!\}$$

In any database state, this construct represents the value that Q would have in the state reached by executing update U.

Relational algebra queries $Q \in \mathcal{RA}$ are generated by the grammar

Q	::=	R	base relation
	|	$\{t\}$	singleton set
	|	$\sigma_p(Q)$	selection
	|	$\pi_{\vec{A}}(Q)$	projection
	|	$Q \cup Q$	union
	|	$Q \cap Q$	intersection
	|	$Q \times Q$	cartesian product
	|	$Q \bowtie Q$	join
	|	$Q - Q$	difference

We have included some redundancy in the operators here, because that will be useful in the context of query optimization. In the formal language we use coordinate position rather than attribute name to identify fields of a tuple or relation. (However, in examples we may use attribute names.) We assume the usual typing rules concerning the arities of

query expressions (see [Ull88]). We also omit discussion of the particular syntax for specifying selection and projection conditions.

Simple updates $U \in \mathcal{U}$ are generated by the grammar

$$\begin{array}{rcll} U & ::= & ins(R,Q) & \text{insert the value of } Q \text{ into } R \\ & | & del(R,Q) & \text{delete the value of } Q \text{ from } R \\ & | & (U;U) & \text{sequence} \end{array}$$

The set of all relations (of arbitrary arity) is denoted by $\mathcal{R}$. A (*database*) *state* is a function DB mapping every relation name $S \in \mathcal{Z}$ to a relation $DB(S) \in \mathcal{R}$ of the appropriate arity. The set of all database states is denoted $\mathcal{DB}$.

Each syntactic category has an associated semantic function,

$$\begin{array}{rcl} [\![Q]\!] & : & \mathcal{DB} \to \mathcal{R}, \\ [\![U]\!] & : & \mathcal{DB} \to \mathcal{DB}, \\ [\![H]\!] & : & \mathcal{DB} \to \mathcal{R}. \end{array}$$

The semantic function $[\![Q]\!]$ is defined in the usual way. The function $[\![U]\!]$ is defined recursively as

$$\begin{array}{rcl} [\![ins(R,Q)]\!](DB) & = & DB[R \leftarrow [\![R \cup Q]\!](DB)] \\ [\![del(R,Q)]\!](DB) & = & DB[R \leftarrow [\![R - Q]\!](DB)] \\ [\![(U_1;U_2)]\!](DB) & = & [\![U_2]\!]([\![U_1]\!](DB)) \end{array}$$

Here the notation $DB[R \leftarrow V]$ denotes the database state DB' such that $DB'(R) = V$ and $DB'(S) = DB(S)$ when $R \neq S$. In practice, if a database is currently in state DB and update expression U is executed, then the database will move to state $[\![U]\!](DB)$.

The semantics of hypothetical queries is simply defined as

$$[\![Q \text{ when } \{\!|U|\!\}]\!](DB) = [\![Q]\!]([\![U]\!](DB)).$$

3.2 Substitutions in the abstract

As we indicated in Section 2, the query Q when $\{\!|U|\!\}$ can be simplified using substitutions. We now present an abstract theory of substitutions, similar to what can be found in [Llo87], that will be used to make this connection more precise.

A partial function $\mu : \mathcal{Z} \to \mathcal{RA}$ is called a *substitution* if for every $S \in dom(\mu)$ the arity of S is equal to the arity of $\mu(S)$. The set of all substitutions over $\mathcal{RA}$ is denoted by $\Sigma(\mathcal{RA})$. We will use the notation $\mu = \{Q_1/S_1, \cdots, Q_j/S_j\}$, to denote the substitution μ where $dom(\mu) = \{S_1, \cdots, S_j\}$ and $\mu(S_i) = Q_i$. Given a substitution μ and a query Q, the function $sub(Q,\mu)$ denotes the query that results from replacing every occurrence of a name S in Q with the query $\mu(S)$, for each $S \in dom(\mu)$. For any substitution μ, the query $sub(Q,\mu)$ is called a *substitution instance of* Q.

Example 3.1: Let $\mu = \{(S - R)/R, \ \sigma_q(R)/S\}$ and $Q = (\pi_{\vec{A}}(R \times S)) \cup V$. Then

$$sub(Q,\mu) = (\pi_{\vec{A}}((S - R) \times \sigma_q(R))) \cup V.$$

□

Given substitutions μ_1 and μ_2, we define the *composition of* μ_1 *and* μ_2, denoted $\mu_1 \# \mu_2$, to be the unique substitution μ_3 such that $dom(\mu_3) = dom(\mu_1) \cup dom(\mu_2)$, and for any $S \in dom(\mu_3)$,

$$\mu_3(S) = \begin{cases} sub(\mu_2(S), \mu_1) & S \in dom(\mu_2) \\ \mu_1(S) & \text{otherwise} \end{cases}$$

The use of '#' here is compatible with the use of this symbol in Heraclitus (see [GHJ96, GH97]). The condition that $dom(\mu_1 \# \mu_2) = dom(\mu_1) \cup dom(\mu_2)$ is required in order to guarantee that $\mu_1 \# \mu_2$ is unique, as a substitution. Without this condition we could always "pad out" the substitution with bindings of the form S/S. It is easily verified that

Lemma 3.2: For each substitutions μ_1, μ_2, and μ_3, and each query Q, we have

$$sub(Q, \mu_1 \# \mu_2) = sub(sub(Q, \mu_2), \mu_1),$$

and

$$\mu_1 \#(\mu_2 \# \mu_3) = (\mu_1 \# \mu_2) \# \mu_3.$$

Example 3.3: Suppose that

$$\begin{array}{rcl} \mu_1 & = & \{(S - R)/R, \ \sigma_q(R)/S\}, \\ \mu_2 & = & \{(\pi_{\vec{B}}(R \bowtie T))/S, \ \sigma_p(S)/V\}. \end{array}$$

Then $\mu_1 \# \mu_2$ is equal to

$$\{(S - R)/R, \ (\pi_{\vec{B}}((S - R) \bowtie T))/S, \ \sigma_p(\sigma_q(R))/V\}.$$

□

Remark 3.4: The standard mathematical notation for the composition of two functions, $g : A \to B$ and $f : B \to C$, is $f \circ g$, which denotes the unique function $h : A \to C$ such that for every $a \in A$ we have $h(a) = f(g(a))$. Our use of the phrase "the composition of μ_1 and μ_2" is consistent with this in the following sense. Let $\widetilde{\mu} = \lambda q.sub(q,\mu)$ be the substitution function from $\mathcal{RA}$ to $\mathcal{RA}$ that applies the substitution μ to its argument. Then we have

$$\widetilde{\mu_1 \# \mu_2} = \widetilde{\mu_1} \circ \widetilde{\mu_2}.$$

In other words, for each query Q, $(\widetilde{\mu_1 \# \mu_2})(Q) = \widetilde{\mu_1}(\widetilde{\mu_2}(Q))$. □

3.3 Substitutions as Updates

The substitution

$$\mu = \{Q_1/S_1, \cdots, Q_j/S_j\}$$

can can also be viewed as an "algebraic" representation of the update that simultaneously replaces table S_i with the value of query Q_i. Put another way, the update represented by μ executes the j operations $S_i := Q_i$ in parallel. To arrive at the value of $sub(Q,\mu)$ in state DB, we can either evaluate $sub(Q,\mu)$ in DB or we can "execute" the abstract update represented by μ and then evaluate Q in the resulting state.

To make this more precise, we simply define $apply(DB,\mu)$ to be the database state DB' such that

$$DB'(R) = \begin{cases} [\![\mu(R)]\!](DB) & R \in dom(\mu) \\ DB(R) & \text{otherwise} \end{cases}$$

Lemma 3.5: Let Q be any query and μ be any substitution. Then for every database state DB,

$$[\![sub(Q,\mu)]\!](DB) = [\![Q]\!](apply(DB,\mu)).$$

When substitutions are viewed as updates, then composition corresponds to sequential execution:

Lemma 3.6: Let μ_1 and μ_2 be substitutions. Then for every database state DB,

$$apply(DB, \mu_1 \# \mu_2) = apply(apply(DB,\mu_1),\mu_2).$$

Remark 3.7: There is a subtle difference between treating a substitution as a syntactic replacement and treating it as an update. This difference can easily lead to some confusion. Let $\widehat{\mu} = \lambda d.apply(d, \mu)$ be the *update function* from $\mathcal{DB}$ to $\mathcal{DB}$ that applies the substitution *as an update*. The preceding lemma is telling us that

$$\widehat{\mu_1 \# \mu_2} \;=\; \widehat{\mu_2} \circ \widehat{\mu_1}.$$

If we compare this with the equation of Remark 3.4,

$$\widetilde{\mu_1 \# \mu_2} \;=\; \widetilde{\mu_1} \circ \widetilde{\mu_2},$$

we see that when substitutions are viewed as updates, then the composition operator corresponds to composing updates functions in the *opposite* order that the corresponding substitution functions are applied. It helps to remember that the first equation defines how we can *realize* the transition to a new state in two steps, whereas the second equation is telling how to *anticipate* the effects of this transition *before* it has actually taken place. This difference can be expressed in the equation

$$[\![(\widetilde{\mu_1} \circ \widetilde{\mu_2})(Q)]\!](DB) \;=\; [\![Q]\!]((\widehat{\mu_2} \circ \widehat{\mu_1})(DB)),$$

which clearly illustrates the reversal. □

3.4 Updates as Substitutions

We now go in the other direction and show that updates can be transformed into substitutions. Given an update U, we shall define the substitution $slice(U)$ that has the same effect as U when it is treated as an "abstract" update. We use the name "slice" because it represents an operation that is similar in many ways to *program slicing* [HR92]. If $R \in dom(slice(U))$, then we can think of the query $Q = (slice(U))(R)$ as representing the "slice" of U that is relevant to R. Put another way, as far as R is concerned, the update U has the same effect as executing $R{:=}Q$.

The substitution $slice(U)$ is defined recursively as

$$\begin{array}{rcl} slice(ins(R,Q)) & = & \{(R \cup Q)/R\} \\ slice(del(R,Q)) & = & \{(R - Q)/R\} \\ slice((U_1;U_2)) & = & slice(U_1) \# slice(U_2) \end{array}$$

Example 3.8: Let $U = (ins(R, Q_1); del(S, \sigma_p(R)))$. Then

$$\begin{array}{rcl} slice(U) & = & slice(ins(R,Q_1)) \# slice(del(S,\sigma_p(R))) \\ & = & \{(R \cup Q_1)/R\} \# \{(S - \sigma_p(R))/S\} \\ & = & \{(R \cup Q_1)/R,\ \ (S - \sigma_p(R \cup Q_1))/S\}. \end{array}$$

□

Using Lemma 3.6 we obtain the following.

Lemma 3.9: For each update U and each database state DB, $apply(DB, slice(U)) = [\![U]\!](DB)$.

This lemma and Lemma 3.5 allow us to show that a hypothetical query Q when $\{\!\{U\}\!\}$ *can be evaluated in the current state* since it is equivalent to a substitution instance of Q.

Theorem 3.10: Let Q be a query and U be an update. For each database state the $\mathcal{RA}$ query $sub(Q, slice(U))$ has the same value as the hypothetical query Q when $\{\!\{U\}\!\}$. That is, for each state DB,

$$[\![Q \text{ when } \{\!\{U\}\!\}]\!](DB) \;=\; [\![sub(Q, slice(U))]\!](DB).$$

□

Example 3.11: Let U be the update described in Example 3.8. If $Q = \pi_{\vec{A}}(S) \bowtie V$, then the corollary tells us that evaluating Q when $\{\!\{U\}\!\}$ will always be the same as evaluating

$$sub(Q, \mu) \;=\; \pi_{\vec{A}}(S - \sigma_p(R \cup Q_1)) \bowtie V.$$

Unlike the hypothetical query Q when $\{\!\{U\}\!\}$, this query could be optimized using standard techniques. □

4 HQL: Hypothetical Query Language

The previous section demonstrates how (1) substitutions can be treated as updates, how (2) updates can be expressed as substitutions, and how (3) hypothetical queries can be expressed as substitution instances. We now introduce syntactic constructs into the query language that represent substitutions and the application of *suspended* substitutions. That is, we take the *meta-level* functions of the previous section and add them to the language as *explicit syntactic objects*. This is similar in spirit to the work of [ACCL91], where explicit substitutions are used to represent suspended λ-reductions.

4.1 Syntax of HQL

HQL will include a grammar for hypothetical-state expressions η and for explicit substitutions ϵ. We extend the grammar of $\mathcal{RA}$ queries to that of $\mathcal{RA}^{\text{hyp}}$:

$$\begin{array}{rcll} Q & ::= & \cdots & \\ & | & Q \text{ when } \eta & \text{hypothetical query} \end{array}$$

That is, hypothetical queries can now occur at any nesting level within queries. Hypothetical-state expressions $\eta \in \mathcal{HS}$ are defined by the rule

$$\begin{array}{rcll} \eta & ::= & \epsilon & \text{explicit substitution} \\ & | & \{\!\{U\}\!\} & \text{hypothetical state reached by } U \\ & | & \eta \,\#\, \eta & \text{composition} \end{array}$$

while explicit substitutions $\epsilon \in \mathcal{E}$ are defined by the grammar

$$\epsilon \;::=\; \{\!\{Q_1/S_1, \cdots, Q_j/S_j\}\!\} \qquad j \geq 0$$

where the Q_i are $\mathcal{RA}^{\text{hyp}}$ queries and the S_i are distinct names from $\mathcal{Z}$.

4.2 Direct Semantics for HQL

The direct semantics of HQL is defined by the functions

$$\begin{array}{rcl} [\![Q]\!] & : & \mathcal{DB} \to \mathcal{R}, \\ [\![\eta]\!] & : & \mathcal{DB} \to \mathcal{DB}, \\ [\![\epsilon]\!] & : & \mathcal{DB} \to \mathcal{DB}. \end{array}$$

The function $[\![Q]\!]$ is simply obtained by extending the semantic function defined in Section 3 with the equation

$$[\![Q \text{ when } \eta]\!](DB) \;=\; [\![Q]\!]([\![\eta]\!](DB)).$$

The function $[\![\eta]\!](DB)$ is defined as

$$\begin{array}{rcl} [\![\cdots, Q_j/S_j, \cdots]\!](DB) & = & DB[\cdots, S_j \leftarrow [\![Q_j]\!](DB), \cdots] \\ [\![\{\!\{U\}\!\}]\!](DB) & = & [\![U]\!](DB) \\ [\![\eta_1 \,\#\, \eta_2]\!](DB) & = & [\![\eta_2]\!]([\![\eta_1]\!](DB)) \end{array}$$

This reflects the direct computational interpretation of a hypothetical-state expression. We shall now see that any η can be transformed into an equivalent abstract substitution.

4.3 Reduction : A Substitution Semantics for HQL

We define a function $red(\eta)$ (called *reduce*) that maps any partial state expression η to an abstract substitution $red(\eta) \in \Sigma(\mathcal{RA})$. In addition, we define a function $red(Q)$ that transforms any $Q \in \mathcal{RA}^{\text{hyp}}$ into a query $red(Q) \in \mathcal{RA}$. These functions are defined in a mutually recursive manner as follows. (Here u_op denotes any unary algebraic operator, and b_op denotes any binary algebraic operator.)

$$\begin{aligned} red(\cdots, Q_j/S_j, \cdots) &= \{\cdots, red(Q_j)/S_j, \cdots\} \\ red(\{\!|U|\!\}) &= slice(U) \\ red(\eta_1 \ \# \ \eta_2) &= red(\eta_1) \# red(\eta_2) \end{aligned}$$

and

$$\begin{aligned} red(R) &= R \\ red(\{t\}) &= \{t\} \\ red(\text{u\_op}(Q)) &= \text{u\_op}(red(Q)) \\ red(Q_1 \ \text{b\_op} \ Q_2) &= (red(Q_1)) \ \text{b\_op} \ (red(Q_2)) \\ red(Q \ \textbf{when} \ \eta) &= sub(red(Q), red(\eta)) \end{aligned}$$

Note that this transformation makes explicit the relationship between the *meta-level* and *object-level* constructs. It clearly illustrates the fact that hypothetical state expressions can be viewed as substitutions and that in an hypothetical query 'Q when η', the expression '**when** η' can be viewed as a *suspended* substitution. Of course these remarks assume that the direct semantics and the reduction semantics are in agreement, which is proved by the following theorem.

Theorem 4.1: If $Q \in \mathcal{RA}^{\text{hyp}}$, then $red(Q) \in \mathcal{RA}$. Furthermore, for each database state DB, each query $Q \in \mathcal{RA}^{\text{hyp}}$, and each hypothetical-state expression η

1. $[\![Q]\!](DB) = [\![red(Q)]\!](DB)$,
2. $[\![\eta]\!](DB) = apply(DB, red(\eta))$.

Note that this is a generalization of Theorem 3.10.

5 Optimization

The reduction semantics presented in the last section provides one strategy for evaluating HQL queries — a query $Q \in \mathcal{RA}^{\text{hyp}}$ can be *reduced* to the query $red(Q) \in \mathcal{RA}$. This allows hypothetical queries to be optimized using standard techniques developed for $\mathcal{RA}$. It also suggests one approach to implementing HQL on top of standard relational database systems.

This section develops a more general framework for optimizing and evaluating hypothetical queries, that incorporates both eager and lazy evaluation of hypothetical-state expressions. A central component of this framework is an equational theory for HQL, which is a combination of (1) a conventional equational theory for the relational algebra, (2) an equational theory of explicit substitutions, and (3) a set of equations that relate hypothetical states of the form $\{\!|U|\!\}$ to explicit substitutions. The equational theory is considered in Subsection 5.1.

Another ingredient of the optimization strategy is the notion of Evaluable Normal Form (ENF). An arbitrary HQL query Q can be converted into one or more equivalent ENF queries Q' using the equational theory as a set of rewriting rules. ENF is defined in Subsection 5.2.

The syntax tree of an ENF query Q' can be used to guide a systematic execution of the operators in Q'. Each explicit substitution present in Q' will be materialized during this execution, and subqueries within the scope of the explicit substitution will be "filtered" against that materialization. Thus, for a given HQL query Q, the choice of an equivalent ENF query Q' is in effect the choice of how eager or lazy the evaluation of Q.

Subsection 5.3. introduces the notion of "explicit substitution value" or "xsub-value". These are physical values that can be associated with explicit substitutions. Two operators involving xsub-values are also introduced.

Subsection 5.4 presents two algorithms for evaluating ENF queries, which are based on the use of xsub-values. The first performs a systematic depth-first traversal of an ENF syntax tree. The second permits some clustered evaluation of individual algebraic operations (e.g., combining a join with selects and projects immediately above or below it).

If the hypothetical updates in an HQL query Q change only a small portion of the data, then it is not efficient to compute the full value of the associated explicit substitutions. Subsection 5.5 considers a generalization of one of the query evaluation algorithms to use delta values, in the sense of Heraclitus.

5.1 An equational theory for HQL

This subsection presents a family of equivalences that can be used for rewriting HQL queries. Some of the equivalences stem from properties of substitutions (e.g., see [HS86, Llo87]), and others stem from the interaction of **when** with the relational algebra operators.

The family of equivalences involving constructs from HQL is called EQUIV_{when}, and is shown in Figure 1. In this figure, u_op (b_op) denotes an arbitrary unary (binary) relational algebra operator. Some definitions are needed for the rules in Figure 1 called "substitution-simplification" and "commute-hypotheticals". If ϵ is an explicit substitution and R a relation name, then $\epsilon|_{-R}$ denotes the explicit substitution obtained from ϵ by deleting the binding with right-side 'R' (if any). The mutually recursive definition of functions *free* and *dom* is presented in Figure 2. Intuitively, $free(E)$ is the set of relation names occurring "free" in E. For hypothetical-state and update expressions E, $dom(E)$ is the set of relation names that are "defined" by E. In essence, these functions articulate the scoping rules for the **when** construct.

It can be verified that the equations of EQUIV_{when} are sound (proof omitted). Soundness continues to hold if equations for the relational algebra are incorporated into the system.

5.2 Evaluable Normal Form (ENF)

This subsection defines a normal form for HQL queries, and describes a family of syntax trees that we use to represent queries in this form.

An HQL query is in *evaluable normal form* (ENF) if it does not use composition (#) or expressions of the form $\{\!|U|\!\}$. This means that all hypothetical-state expressions are in the form of explicit substitutions. Note that for an explicit substitution of form $\{\!|\ldots, Q/R, \ldots|\!\}$, the **when** construct may occur within Q.

We use specialized syntax trees to represent ENF queries. Each node v of the syntax tree for an ENF query has a label, denoted as $label(v)$, that is either a relational operator, or a relation name, or '**when**', or '$\{\!|\ |\!\}$'. These trees differ from conventional relational algebra trees in that (a) the **when**

$$
\begin{array}{rcl}
R \text{ when } \varepsilon & \equiv & Q, \text{ if } \varepsilon \in \mathcal{E}_{ra} \text{ and } Q/R \in \varepsilon \\
R \text{ when } \varepsilon & \equiv & R, \text{ if } \varepsilon \in \mathcal{E}_{ra} \text{ and } R \text{ has no binding in } \varepsilon \\
\{t\} \text{ when } \eta & \equiv & \{t\} \\
(\text{u\_op}(Q)) \text{ when } \eta & \equiv & \text{u\_op}(Q_1 \text{ when } \eta) \\
(Q_1 \text{ b\_op } Q_2) \text{ when } \eta & \equiv & (Q_1 \text{ when } \eta) \text{ b\_op } (Q_2 \text{ when } \eta)
\end{array}
$$

push-when-into-algebra-expressions

$$
\begin{array}{rcl}
\{\!\{ins(R,Q)\}\!\} & \equiv & \{(R \cup Q)/R\} \\
\{\!\{del(R,Q)\}\!\} & \equiv & \{(R - Q)/R\} \\
\{\!\{(U_1;U_2)\}\!\} & \equiv & \{\!\{U_1\}\!\} * \{\!\{U_2\}\!\}
\end{array}
$$

convert-to-explicit-substitutions

$$(Q \text{ when } \eta_1) \text{ when } \eta_2 \equiv Q \text{ when } (\eta_2 \# \eta_1)$$

replace-nested-when

$$(\eta_1 * \eta_2) * \eta_3 \equiv \eta_1 * (\eta_2 * \eta_3)$$

associativity

$$
\begin{array}{rcl}
\varepsilon_1 * \varepsilon_2 & \equiv & \{\!\{P_1 \text{ when } \varepsilon_1/S_1, \ldots, P_m \text{ when } \varepsilon_1/S_m, \\
 & & \quad Q_{i_1}/R_{i_1}, \ldots, Q_{i_k}/R_{i_k}\}\!\}
\end{array}
$$

$$
\begin{array}{rcl}
\text{where} \quad \varepsilon_1 & = & \{\!\{Q_1/R_1, \ldots, Q_n/R_n\}\!\} \\
\varepsilon_2 & = & \{\!\{P_1/S_1, \ldots, P_m/S_m\}\!\} \\
\{R_{i_1}, \ldots, R_{i_k}\} & = & dom(\varepsilon_1) - dom(\varepsilon_2)
\end{array}
$$

compute-composition

$$
\begin{array}{rcll}
Q \text{ when } \varepsilon & \equiv & Q \text{ when } \varepsilon|_{-R}, & \text{if } R \notin free(Q) \\
Q \text{ when } \varepsilon & \equiv & Q \text{ when } \varepsilon|_{-R}, & \text{if } (R/R) \in \varepsilon \\
Q \text{ when } \{\!\{\,\}\!\} & \equiv & Q &
\end{array}
$$

substitution-simplification

$$(Q \text{ when } \eta_1) \text{ when } \eta_2 \equiv (Q \text{ when } \eta_2) \text{ when } \eta_1$$

$$
\begin{array}{rcll}
\text{if } dom(\eta_1) \cap dom(\eta_2) & = & dom(\eta_1) \cap free(\eta_2) & \\
 & = & dom(\eta_2) \cap free(\eta_1) & = \emptyset
\end{array}
$$

commute-hypotheticals

Figure 1: The family EQUIV_{when} of equivalences for HQL expressions

$$
\begin{array}{rcl}
free(Q) & = & \text{all relation names in } Q \\
 & & \text{if } Q \in \mathcal{RA} \\
free(Q \text{ when } \eta) & = & free(\eta) \cup (free(Q) - dom(\eta)) \\
 & & Q \in \mathcal{RA}^+ \text{and } \eta \in \mathcal{HS} \\
free(ins(R,Q)) & = & free(Q) \\
free(del(R,Q)) & = & free(Q) \\
free((U_1;U_2)) & = & free(U_1) \cup (free(U_2) - dom(U_1)) \\
free(\{Q_1/R_1, \ldots, Q_n/R_n\}) & = & \cup\{free(Q_i) \mid i \in [1,n]\} \\
free(\{\!\{U\}\!\}) & = & free(U) \\
free(\eta_1 \# \eta_2) & = & free(\eta_1) \cup (free(\eta_2) - dom(\eta_1))
\end{array}
$$

$$
\begin{array}{rcl}
dom(Q) = \emptyset & & \text{if } Q \text{ a query expression} \\
dom(ins(R,Q)) = R & & \\
dom(del(R,Q)) = R & & \\
dom((U_1;U_2)) = dom(U_1) \cup dom(U_2) & & \\
dom(\{Q_1/R_1, \ldots, Q_n/R_n\}) = \{R_1, \ldots, R_n\} & & \\
dom(\eta_1 \# \eta_2) = dom(\eta_1) \cup dom(\eta_2) & & \\
dom(\{\!\{U\}\!\}) = dom(U) & &
\end{array}
$$

Figure 2: Definition of functions $free(\cdot)$ and $dom(\cdot)$

keyword is present, as an infix binary operator, and (b) an explicit substitution $\{\!\{Q_1/R_1, \ldots, Q_n/R_n\}\!\}$ is represented as follows: the parent node is labeled by $\{\!\{\,\}\!\}$, and has n children. For each i there a child corresponding to query Q_i. Also, the edge to that child has label R_i.

5.3 Explicit substitution values

Evaluation of an ENF query involves the construction of relations, and and of physical values corresponding to explicit substitutions. This subsection introduces the notion of "explicit substitution value", and describes algebraic operators that use it. In particular, we develop an equation that describes how explicit substitution values should be combined when evaluating ENF queries with nested when's.

In general, we are interested in evaluating an ENF query in the context of a fixed database state, DB. As defined in Section 3, the semantics of an explicit substitution ε is a function from $\mathcal{DB}$ to $\mathcal{DB}$. Thus, the value of $\varepsilon(DB)$ is a full database state. In practice, if a relation name $R \in \mathcal{Z}$ is not in $dom(\varepsilon)$ there is no point in replicating the value of $DB(R)$ when constructing the value of $\varepsilon(DB)$. An *explicit substitution value*, or *xsub-value*, is a (partial) function E that maps relation names in $\mathcal{Z}$ into relations, where relation $E(R)$ has the same arity as relation name R for each $R \in dom(E)$. We often denote an xsub-value as a set $E = \{J_1/R_1, \ldots, J_n/R_n\}$ of binding pairs, where each J_i is a physical relation. The family of xsub-values is denoted by $\mathcal{ESV}$.

One basic operator involving xsub-values is *apply*, that maps database states and xsub-values to database states. This is defined by

$$apply(DB,E)(R) = \begin{cases} E(R) & \text{if } R \in dom(E) \\ DB(R) & \text{otherwise} \end{cases}$$

There is a natural mapping $[\![\cdot]\!]_{x\_val}$ such that for each explicit substitution ε we have

$$[\![\varepsilon]\!]_{x\_val} : \mathcal{DB} \to \mathcal{ESV}.$$

This is defined by

$$
\begin{array}{l}
[\![\{\!\{Q_1/R_1, \ldots, Q_n/R_n\}\!\}]\!]_{x\_val}(DB) \\
\quad = \{\, [\![Q_1]\!](DB)/R_1, \ldots, [\![Q_n]\!](DB)/R_n \,\}
\end{array}
$$

It is clear that for each query Q, explicit substitution ε and state DB,

$$apply(DB, [\![\varepsilon]\!]_{x\_val}(DB)) = [\![\varepsilon]\!](DB)$$

We now develop an operator for combining xsub-values, which is closely related to the compose operator for explicit substitutions. We call this operator "smash", denoted '!', because it is analogous to the smash operator for delta values in Heraclitus. The *smash* of xsub-values E_1 and E_2 has domain $dom(E_1) \cup dom(E_2)$, and is defined by

$$(E_1 \,!\, E_2)(R) = \begin{cases} E_2(R) & \text{if } R \in dom(E_2) \\ E_1(R) & \text{if } R \in dom(E_1) - dom(E_2) \end{cases}$$

$$
\begin{aligned}
filter_1(R, E) &= \begin{cases} E(R) & R \in dom(E) \\ DB(R) & R \notin dom(E) \end{cases} \\
filter_1(\text{u\_op}(Q), E) &= \text{u\_op}(filter_1(Q, E)) \\
filter_1(Q_1 \text{ b\_op } Q_2, E) &= filter_1(Q_1, E) \text{ b\_op } filter_1(Q_2, E) \\
filter_1(\{\!|Q_1/R_1, \ldots, Q_n/R_n|\!\}, E) &= \{filter_1(Q_1, E)/R_1, \ldots, filter_1(Q_n, E)/R_n\} \\
filter_1(Q \text{ when } \varepsilon, E) &= filter_1(Q,\ E\ !\ filter_1(\varepsilon, E))
\end{aligned}
$$

Figure 3: The function $filter_1$

It is straightforward to verify for explicit substitutions ε_1 and ε_2 and state DB that

$$
\begin{aligned}
&[\![\varepsilon_1 \ \# \ \varepsilon_2]\!]_{x\_val}(DB) \\
&\quad = [\![\varepsilon_1]\!]_{x\_val}(DB)\ !\ [\![\varepsilon_2]\!]_{x\_val}(apply(DB, [\![\varepsilon_1]\!]_{x\_val}(DB))).
\end{aligned}
$$

The following equation describes how smash can be used to combine nested xsub-values during the evaluation of ENF queries. For each query Q we have:

$$
\begin{aligned}
&[\![(Q \text{ when } \varepsilon_2) \text{ when } \varepsilon_1]\!](DB) \\
&\quad = [\![Q \text{ when } (\varepsilon_1 \ \# \ \varepsilon_2)]\!](DB) \\
&\quad = [\![Q]\!](apply(DB, [\![\varepsilon_1]\!]_{x\_val}(DB)\ ! \\
&\qquad\qquad [\![\varepsilon_2]\!]_{x\_val} apply(DB, [\![\varepsilon_1]\!]_{x\_val}(DB)))).
\end{aligned}
$$

5.4 Evaluation of ENF queries using xsub-values

A variety of algorithms can be developed for evaluating ENF queries using xsub-values. This subsection presents two such algorithms. This first is relatively straightforward, and exposes the basic approach to "filtering" the evaluation of a query by an xsub-value. The second improves the first, by permitting clustered evaluation of relational algebra operators (e.g., combining a join with a select). Additional algorithms are presented in [GH97].

For the first algorithm we shall use a function $filter_1$ with signature

$$filter_1 : \text{HQL expressions} \times \mathcal{ESV} \times \mathcal{DB} \rightarrow (\mathcal{R} \cup \mathcal{ESV}).$$

We generally assume that the database state is given by the context, and so omit the third argument when mentioning this function. The recursive definition of $filter_1$ is given in Figure 3. In that figure, on the left side of the equality the symbols u_op and b_op are in the syntax of the relational algebra, and on the right side they indicate the application of algebraic operators to relations.

The function $filter_1$ is constructed so that evaluating $filter_1(Q, E)$ in state DB yields $[\![Q]\!](apply(DB, E))$. Intuitively, the function $filter_1$ does a depth-first traversal of the syntax tree of an ENF query Q. When encountering a when node, the right child is processed before the left child. The definition of $filter_1$ on individual relation names performs the actual "filtering" by an xsub-value. The action of $filter_1$ on the keyword 'when' has the effect of smashing together all of the xsub-values that affect a given subquery. (This corresponds to the behavior of the "run-time when stack" in the implementation of Heraclitus described in [GHJ+93].)

Algorithm HQL-1. Let Q be an HQL query in ENF, and DB a state. The value Q in DB is obtained by evaluating $filter_1(Q, \{\ \})$, i.e., $filter_1$ applied to Q and the empty xsub-value.

Proposition 5.1: Algorithm HQL-1 is correct.

A significant weakness of Algorithm HQL-1 is that it does not permit grouping of relational algebra operators into single physical operations, as frequently occurs in traditional optimizers for relational algebra queries. For example, the algorithm would not permit the grouping of a join and a select in a query such as $R \bowtie \sigma(S)$. We now modify the algorithm to permit some grouping.

Suppose that Q is an ENF query and T its syntax tree. The basic idea of the modified algorithm is to identify and "collapse" certain subtrees of T that correspond to pure relational algebra queries, and to permit the use of an optimized query processing algorithm on those queries.

Example 5.2: Let T be the syntax tree of query

$$Q = (Q_1 \text{ when } \varepsilon_1) \bowtie (R \bowtie \sigma(Q_2 \text{ when } \varepsilon_2)).$$

Using the *collapse* operator defined below, tree $collapse(T)$ will have a root v labeled by $S_1 \bowtie (R \bowtie \sigma(S_2))$ that has three children. The first child will be the root of (the tree of) Q_1 when ε_1, the second child will be root of Q_2 when ε_2, and the third child will have label R. □

We now define the operator *collapse* that maps ENF syntax trees into a variant of ENF syntax trees. In this variant, nodes can be labeled by relation names in $\mathcal{Z}$, by 'when', by '$\{\!|\ |\!\}$', and by relational algebra queries of the form $Q[S_1, \ldots, S_k, R_1, \ldots, R_m]$, where $R_1, \ldots, R_m$ are distinct relation names in $\mathcal{Z}$ and $S_1, \ldots, S_k$ are "new" relation names. A node labeled $Q[S_1, \ldots, S_k, R_1, \ldots, R_m]$ will have $k + m$ children. Below the first k children will be subtrees with root node labeled by 'when', and the remaining m children will be leaves, labeled by $R_1, \ldots, R_m$, respectively.

The definition of *collapse* is recursive on the structure of ENF syntax trees. Suppose first that ENF syntax tree T has structure 'T_0 when $\{\!|T_1/R_1, \ldots, T_n/R_n|\!\}$'. Then $collapse(T)$ is defined to be

$$collapse(T_0) \text{ when } \{\!|collapse(T_1)/R_1, \ldots, collapse(T_n)/R_n|\!\}.$$

If ENF syntax tree T is a single node labeled by $R \in \mathcal{Z}$, then $collapse(T) = T$.

Finally, suppose that T is an ENF syntax tree with root node v, which is labeled by a relational algebra operator. Let $u_1, \ldots, u_k$ be the nodes below v whose labels are relation names or when, such that each node between v and u_i is labeled by a relational operator. Let $T_1, \ldots, T_m$ be the subtrees below the nodes among $u_1, \ldots, u_n$ that have label when, and let $R_1, \ldots, R_k$ be the relation names occurring as labels of $u_1, \ldots, u_n$. Let $Q_v[S_1, \ldots, S_m, R_1, \ldots, R_k]$ denote the relational algebra query that corresponds to the subtree of T with root v and leaves $u_1, \ldots, u_n$, where $S_1, \ldots, S_m$ are new relation names corresponding to subtrees $T_1, \ldots, T_m$. Then $collapse(T)$ is the tree that has root v, where the label of v is $Q_v[S_1, \ldots, S_m, R_1, \ldots, R_k]$, v has $m + k$ children, the first

m children are the roots of $collapse(T_1), \ldots, collapse(T_m)$, respectively, and the latter k children are leaves and are labeled by $R_1, \ldots, R_k$, respectively.

We shall also use the function $filter_2$, which is essentially the same as $filter_1$ but for collapsed trees. To describe $filter_2$, we view $filter_1$ as a mapping involving subtrees of ENF syntax trees (as opposed to expressions from HQL). Function $filter_2$ is identical to $filter_1$, except for its behavior on nodes with labels of form $Q_v(S_1, \ldots, S_m, R_1, \ldots, R_k)$. Suppose that v is such a label, with subtrees $T_1, \ldots, T_m$ below the first m children. For xsub-value E, we have

$$
\begin{array}{l}
filter_2(Q_v[S_1, \ldots, S_m, R_1, \ldots, R_k], E) = \\
\quad \textbf{let } S_1 \ = \ filter_2(T_1, E) \textbf{ in} \\
\quad \vdots \\
\quad \textbf{let } S_m \ = \ filter_2(T_m, E) \textbf{ in} \\
\quad eval\_filter\_x(Q[S_1, \ldots, S_m, R_1, \ldots, R_k], E)
\end{array}
$$

The function $eval\_filter\_x$ can be based on any conventional (optimized) algorithm for evaluating relational algebra queries, with the modification that each relation R_i mentioned explicitly in $Q[S_1, \ldots, S_m, R_1, \ldots, R_k]$ is filtered by xsub-value E (analogous to the action of $filter_1$ on individual relation names). Importantly, $eval\_filter\_x$ might cluster several relational operators together into a single physical operator.

Algorithm HQL-2. Let Q be an ENF query with syntax tree T. The value of Q in state DB is now computed as $filter_2(collapse(T), \{\})$.

Using Proposition 5.1, the following is easily verified.

Proposition 5.3: Algorithm HQL-2 is correct.

5.5 Generalization to use deltas

Frequently, the hypothetical changes to a database state specified by an hypothetical update $\{\!|U|\!\}$ will modify only a small fraction of the state. This observation is central to the optimizations studied in connection with both hypothetical relations [WS83] and Heraclitus [GHJ96]. We now consider how Algorithm HQL-2 can be generalized to permit the use of explict delta values, following the spirit of Heraclitus.

We begin by defining the notion of delta value, and illustrating how delta values can provide a substantial performance gain over xsub-values. A *delta value* is a (partial) function Δ that maps relation names in $\mathcal{Z}$ to pairs of relations, where the two relations in $\Delta(R)$ have the same arity as R, for each $R \in dom(\Delta)$. We often denote a delta value as a set $\Delta = \{\langle D_1, I_1\rangle/R_1, \ldots, \langle D_n, I_n\rangle/R_n\}$ of binding pairs, where each D_i and I_i is a physical relation. If $R \in dom(\Delta)$ and $\Delta(R) = \langle D, I\rangle$, we sometimes denote D by R^-_Δ and denote I by R^+_Δ. Also, if $R \notin dom(\Delta)$, then we define $R^-_\Delta = R^+_\Delta = \emptyset$. The family of delta values[1] is denoted by $\mathcal{DV}$.

One basic operator involving delta values is *apply*, that maps database states and delta values to database states. This is defined in analogy to *apply* for xsub-values, by

$$apply(DB, \Delta)(R) \ = \ ((DB(R) - R^-_\Delta) \cup R^+_\Delta).$$

[1] In Heraclitus [GHJ96], a delta value Δ is required to satisfy the condition that for each R, $R^-_\Delta \cap R^+_\Delta = \emptyset$; this condition is not required for the treatment here.

We now illustrate how the use of delta values can significantly improve performance over the use of xsub-values. Consider a simple hypothetical query

$$Q \ = \ (R \bowtie S) \textbf{ when } \{\!|U|\!\}.$$

Under Algorithm HQL-2, we might first transform the hypothetical update $\{\!|U|\!\}$ into an explicit substitution of the form $\epsilon = \{\!|Q_1/R, Q_2/S|\!\}$. Let DB be a database state, and suppose that the difference between $DB(R)$ and $[\![Q_1]\!](DB)$ is relatively small (e.g., about 1% of the size of $DB(R)$) and likewise for Q_2. Under Algorithm HQL-2 the complete values of $[\![Q_1]\!](DB)$ and $[\![Q_2]\!](DB)$ will be computed, and the join will be performed using these new relations.

Following Heraclitus, an efficient alternative can be developed by using special-purpose operators, that essentially combine delta application with relational algebra operators. For example, suppose that in state DB, the delta corresponding to ϵ is Δ. (We'll be more specific about computing Δ shortly.) To compute $R \bowtie S$ **when** ϵ we can compute $R \bowtie S$ "when" Δ using the `join_when` operator of Heraclitus. (We put 'when' in quotes, because the delta value Δ is a semantic object, rather than a syntactic object.) This algebraic operator takes six physical relations as operands. In particular, `join_when`$(DB(R), DB(S), R^-_\Delta, R^+_\Delta, S^-_\Delta, S^+_\Delta)$ computes the value of

$$[\ (DB(R) - R^-_\Delta) \cup R^+_\Delta\ ] \bowtie [\ (DB(S) - S^-_\Delta) \cup S^+_\Delta\ ].$$

But instead of computing this as most conventional SQL optimizers might, the `join_when` operator can be optimized for the case where the delta relations are relatively small. For example, one of the implementations of `join_when` described in [GHJ+93] is a variant of the sort-merge algorithm for join. In this variant, all six operands of `join_when` are sorted, and the merge simultaneously tests for (i) the join condition, (ii) membership in R "when" Δ, and (iii) membership in S "when" Δ. As discussed in [GHJ+93], for small deltas the cost of evaluating a join under a delta is only nominally more expensive than evaluating the join without the when. (As a general rule of thumb, if the delta has size x% of the base relations, then the `join_when` will take an additional $2x$% of time over the time for a join of the base relations.)

More generally, references [GHJ+93, GHJ96] describe how to use `join_when` and analogous operators to efficiently evaluate hypothetical queries of the form Q "when" Δ, where Q is an arbitrarily deep relational algebra query. We use $eval\_filter\_d(Q, \Delta)$ to denote the result of using such an algorithm. We assume that for each relational algebra query Q and state DB, the result of $eval\_filter\_d(Q, \Delta)$ in DB is equal to $[\![Q]\!](apply(DB, \Delta))$.

There are some trade-offs in connection with the choice of a delta value Δ associated with an explicit substitution ϵ. One possibility, which is very precise, is that for each relation name $R \in dom(\epsilon)$ we set

$$
\begin{array}{rcl}
R^-_\Delta & = & DB(R) - [\![\epsilon(R)]\!](DB) \\
R^+_\Delta & = & [\![\epsilon(R)]\!](DB) - DB(R).
\end{array}
$$

This might be prohibitively expensive to evaluate. Fortunately, more "approximate" delta values can be used. We say that a delta value Δ *captures* xsub-value E in state DB if $apply(DB, \Delta) = apply(DB, E)$. As part of our generalization of Algorithm HQL-2, we describe below an approach for building deltas that capture explicit substitutions, which in many cases will be more efficient than the precise approach just described.

$$
\begin{aligned}
filter_3(Q[T_1,\ldots,T_m,R_1,\ldots,R_k],\Delta) &= \text{let } S_1 = filter_3(T_1,\Delta) \text{ in}\\
&\quad\vdots\\
&\quad \text{let } S_m = filter_3(T_m,\Delta) \text{ in}\\
&\quad eval\_filter\_d(Q[S_1,\ldots,S_m,R_1,\ldots,R_k],\Delta)\\
filter_3(\{\!|del(Q,R)|\!\},\Delta) &= \{\langle filter_3(Q,\Delta),\emptyset\rangle/R\}\\
filter_3(\{\!|ins(Q,R)|\!\},\Delta) &= \{\langle \emptyset, filter_3(Q,\Delta)\rangle/R\}\\
filter_3(\{\!|U;A|\!\},\Delta) &= filter_3(\{\!|U|\!\},\Delta)\,!\,filter_3(\{\!|A|\!\},\Delta\,!\,filter_3(\{\!|U|\!\},\Delta))\\
filter_3(Q \text{ when } \{\!|U|\!\},\Delta) &= filter_3(Q,\Delta\,!\,filter_3(\{\!|U|\!\},\Delta))
\end{aligned}
$$

Figure 4: The function $filter_3$

In analogy to the smash operator for xsub-values, we define a smash operator for delta values. The *smash* of delta values Δ_1 and Δ_2, denoted $\Delta_1!\Delta_2$, is that delta value Δ with domain equal to $dom(\Delta_1) \cup dom(\Delta_2)$, and such that

$$
\begin{aligned}
R^-_\Delta &= (R^-_{\Delta_1} - R^+_{\Delta_2}) \cup R^-_{\Delta_2}\\
R^+_\Delta &= (R^+_{\Delta_1} - R^-_{\Delta_2}) \cup R^+_{\Delta_2}
\end{aligned}
$$

for each R in $dom(\Delta)$. Let ε_1 and ε_2 be explicit substitutions and DB a state. If Δ_1 captures $[\![\varepsilon_1]\!]_{x\_val}(DB)$ in DB and Δ_2 captures $[\![\varepsilon_2]\!]_{x\_val}(apply(DB,\Delta_1))$ in $apply(DB,\Delta_1)$, then $\Delta_1!\Delta_2$ captures $[\![\varepsilon_1 \,\#\, \varepsilon_2]\!]_{x\_val}(DB)$ in state DB. Furthermore, for each relational algebra query Q we have

$$
\begin{aligned}
&[\![(Q \text{ when } \varepsilon_2) \text{ when } \varepsilon_1]\!](DB)\\
&\quad = [\![Q \text{ when } (\varepsilon_1 \,\#\, \varepsilon_2)]\!](DB)\\
&\quad = \text{result of } eval\_filter\_d(Q,\Delta_1!\Delta_2) \text{ in } DB.
\end{aligned}
$$

This is analogous to the equation presented in Subsection 5.3 for explicit substitutions that relates 'when' and smash, and describes how deltas should be combined when evaluating queries under nested when's.

In our generalization of Algorithm HQL-2 we shall use a method for computing delta values that is based on the sequence of atomic inserts and deletes that make up an hypothetical update. To this end, we modify the notion of ENF. An HQL query is in *modified ENF* (*mod-ENF*) if it satisfies the definition ENF, except that instead of explicit substitutions, all hypothetical updates present in the query have the form $\{\!|A_1;\ldots;A_n|\!\}$, where each A_i is an atomic insert or delete.

We now describe Algorithm HQL-3, a generalization of Algorithm HQL-2 that uses delta values. The new algorithm is based on a recursively defined function $filter_3$, which is defined in Figure 4. Operator $filter_3$ takes two arguments: the collapse of a mod-ENF tree and a delta value. In Figure 4, A ranges over atomic deletes and inserts, and U ranges over update expressions of the form $A_1,\ldots,A_n$, where each A_i is an atomic delete or insert.

Of particular interest is the treatment by $filter_3$ of hypothetical updates of form $\{\!|U|\!\}$, where $U = A_1;\ldots;A_n$ where each A_i is an atomic delete or insert. It can be verified that if Δ captures xsub-value E in state DB and ε' is an explicit substitution equivalent to $\{\!|U|\!\}$, then $filter_3(\{\!|U|\!\},\Delta)$ captures the xsub-value $[\![\varepsilon']\!]_{x\_val}(apply(DB,E))$ in state $apply(DB,E)$.

Algorithm HQL-3: Let Q be an mod-ENF query with syntax tree T. Construct $collapse(T)$. The value of Q in state DB is now computed as $filter_3(collapse(T),\{\})$, i.e., the application of $filter_3$ to $collapse(T)$ and the empty delta value.

Using Proposition 5.3 and properties of $eval\_filter\_d$ the following can be shown.

Proposition 5.4: Algorithm HQL-3 is correct.

6 Concluding Remarks

This paper develops a framework for optimizing hypothetical queries, that allows exploration of a broad spectrum of implementation strategies from the purely eager to the purely lazy. The core of the framework is an equational theory and family of rewriting rules, that is analogous to and compatible with the equational theory and rewriting rules used for optimizing relational algebra queries. The relationship between the two theories is explored further in [GH97].

The framework developed in this paper can be used as the basis for a complete optimization strategy for hypothetical queries. For this, several extensions will be needed, including techniques for estimating the cost of execution plans involving xsub-values and delta values; heuristics whereby optimizers can reduce the space of queries that will be considered for execution (e.g., see [GD87, Gra93]); and techniques for evaluating sequences of closely related hypothetical queries (as described in Examples 2.2.

Although we have not presented the details here, it is possible to extend our framework along several dimensions. First, the framework extends to query languages that include bags and aggregation. Second, the framework extends to update languages that include constructs such as conditionals, procedures, temporary tables, and aborting constructs. Such constructs don't extend the expressive power of the update language, but they do dramatically increase the conciseness with which programs can be written. Of course, there are limits to the direct application of our techniques. It may be, for some extensions to the update language, that Q when U is expressible in $\mathcal{RA}$, but not as a substitution instance of Q, or that it is simply not expressible in $\mathcal{RA}$ (for similar results in the context of preconditions, see [BGL96]).

This extended abstract has not treated several constructs found in the Heraclitus languages. For example, the keyword when can be applied to hypothetical-state expressions on the left, e.g., 'η_1 when η_2'. Incorporating such expressions illuminates subtleties concerning the semantics of hypothetical-states; these are explored in the full paper [GH97].

Acknowledgements. We would like to thank Latha Colby, Leonid Libkin, and Gang Zhou for their helpful comments on drafts of this paper.

References

[ACCL91] M. Abadi, L. Cardelli, P. Curien, and J. Levy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[BGL96] M. Benedikt, T. Griffin, and L. Libkin. Verifiable properties of database transactions. In *Proc. ACM Symp. on Principles of Database Systems*, pages 117–127, 1996.

[Bon90] A. J. Bonner. Hypothetical datalog: Complexity and expressibility. *Theoretical Computer Science*, 76:3–51, 1990.

[Bon95] A. J. Bonner. The logical semantics of hypothetical rulebases with deletion. *Journal of Logic Programming*, 1995.

[DHK95] G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions. In *IEEE Symposium on Logic in Computer Science*, pages 366–374, 1995.

[DHR96] M. Doherty, R. Hull, and M. Rupawalla. Structures for manipulating proposed updates in object-oriented databases. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 306–317, 1996.

[Dij75] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivations of programs. *Comm. ACM*, 18:453–457, 1975.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[GD87] G. Graefe and D. J. DeWitt. The EXODUS optimizer generator. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 160–172, 1987.

[GH97] T. Griffin and R. Hull. A framework for implementing hypothetical queries. Technical report, Bell Laboratories, Lucent Technologies, 1997. In preparation.

[GHJ+93] S. Ghandeharizadeh, R. Hull, D. Jacobs, et al. On implementing a language for specifying active database execution models. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 441–454, 1993.

[GHJ96] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus: Elevating deltas to be first-class citizens in a database programming language. *ACM Trans. on Database Systems*, 21(3):370–426, 1996.

[GR84] D. M. Gabbay and U. Reyle. N-Prolog: An extension of prolog with hypothetical implications. I. *Journal of Logic Programming*, 1(4):319–355, 1984.

[GR85] D. M. Gabbay and U. Reyle. N-Prolog: An extension of prolog with hypothetical implications. II: Logical foundations and negation as failure. *Journal of Logic Programming*, 2(4):251–283, 1985.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[Gri81] David Gries. *The Science of Programming.* Springer-Verlag, 1981.

[HR92] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *Proceedings of the 14th International Conference on Software Engineering*, 1992.

[HS86] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, 1986.

[Les94] P. Lescanne. From $\lambda\sigma$ to λv – a journey through calculi of explicit substitutions. In *ACM SIGPLAN-SIGACT Symposium in Principles of Programming Languages* (POPL), pages 60–69, 1994.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming (Second Edition).* Springer-Verlag, Berlin, 1987.

[Qia90] X. Qian. An axiom system for database transactions. *Information Processing Letters*, 36:183–189, 1990.

[Qia91] X. Qian. The expressive power of the bounded-iteration construct. *Acta Informatica*, 28(7):631–656, October 1991.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledgebase Systems, Vol. I.* Computer Science Press, Potomac, Maryland, 1988.

[WS83] J. Woodfill and M. Stonebraker. An implementation of hypothetical relations. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 157–165, September 1983.