March 1992

Structural Recursion as a Query Language

Val Tannen
University of Pennsylvania, val@cis.upenn.edu

Peter Buneman
University of Pennsylvania

Shamim Naqvi
Bellcore

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-92-17.

Structural Recursion as a Query Language

Abstract

We propose a programming paradigm that tries to get close to both the semantic simplicity of relational algebra, and the expressive power of unrestricted programming languages. Its main computational engine is *structural recursion on sets.* All programming is done within a "nicely" typed lambda calculus, as in Machiavelli [OBB89]. A guiding principle is that *how* queries are implemented is as important as whether they can be implemented. As in relational algebra, the meaning of any relation transformer is guaranteed to be a total map taking finite relations to finite relations. A naturally restricted class of programs written with structural recursion has precisely the expressive power of the relational algebra. The same programming paradigm scales up, yielding query languages for the complex-object model [AB89]. Beyond that, there are, for example, efficient programs for transitive closure and we are also able to write programs that move out of sets, and then perhaps back to sets, as long as we stay within a (quite flexible) type system. The uniform paradigm of the language suggests positive expectations for the optimization problem. In fact, structural recursion yields *finer* grain programming. Therefore we expect that lower-level and therefore better optimizations will be feasible.

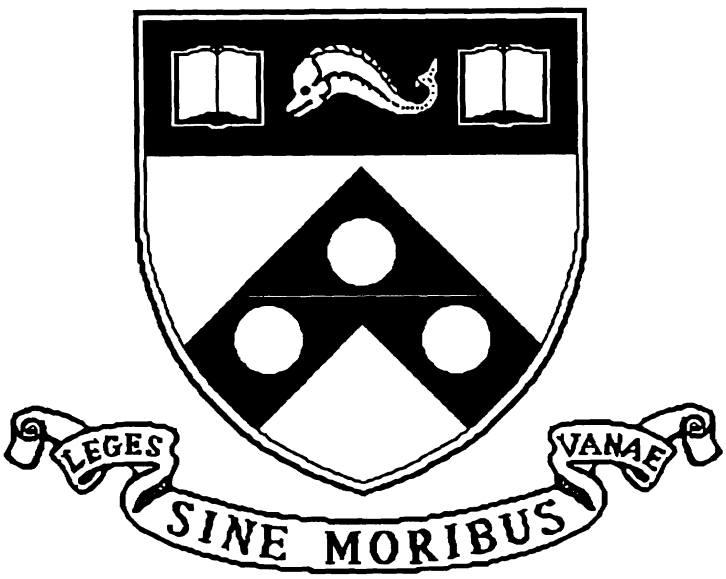
Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-92-17.

Structural Recursion As A Query Language

MS-CIS-92-17
LOGIC & COMPUTATION 46

Val Breazu-Tannen
Peter Buneman
Shamim Naqvi



University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

March 1992

Structural Recursion as a Query Language †

Val Breazu-Tannen, Peter Buneman
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104–6389, USA
val, peter @cis.upenn.edu

Shamim Naqvi
BELLCORE
445 South St.
Morristown, NJ 07960–1910, USA
shamim@bellcore.com

Abstract

We propose a programming paradigm that tries to get close to both the semantic simplicity of relational algebra, and the expressive power of unrestricted programming languages. Its main computational engine is *structural recursion on sets*. All programming is done within a "nicely" typed lambda calculus, as in Machiavelli [OBB89]. A guiding principle is that *how* queries are implemented is as important as *whether* they can be implemented. As in relational algebra, the meaning of any relation transformer is guaranteed to be a total map taking finite relations to finite relations. A naturally restricted class of programs written with structural recursion has precisely the expressive power of the relational algebra. The same programming paradigm scales up, yielding query languages for the complex-object model [AB89]. Beyond that, there are, for example, efficient programs for transitive closure and we are also able to write programs that move out of sets, and then perhaps back to sets, as long as we stay within a (quite flexible) type system. The uniform paradigm of the language suggests positive expectations for the optimization problem. In fact, structural recursion yields *finer grain* programming therefore we expect that lower-level, and therefore better optimizations will be feasible.

1 Introduction

Apart from its simplicity, the major selling point of the relational model is the availability of simple query languages with well-understood properties. Realistic database programming calls, however, for more flexibility, and more expressiveness, than relational algebra (say) offers. To meet these needs, practical query languages add somewhat ad-hoc features (certain aggregate operators, "group-by") and when even this fails, programmers use query languages embedded in general-purpose ones.

We see at least two problems with embedded query languages. The first problem is the mismatch between data models and type systems (the infamous "impedance mismatch" problem). Good progress has been made in overcoming this problem (see for example, Pascal-R, Galileo [Sch77, ACO83]), and in fact, the present paper builds on Machiavelli [OBB89]. But there is also a second problem, which stems from the universality of the languages in which we try to integrate database manipulation primitives. Suppose we have already achevied a smooth integration of the data model in the type system, and consider a function whose inputs and output have the type of relations (*i.e.*, set of tuples, notation $\{\alpha_1 \times \cdots \times \alpha_n\}$). Thinking wishfully, call such a function a *relation transformer*. In relational algebra, the meaning of any relation transformer is guaranteed to be a total map taking finite relations to finite relations. Not so in the powerful languages we just mentioned, where potential non-termination will mess up the semantic picture. We can program *everything*, but proving/understanding properties of such programs becomes much more difficult, and while good optimization techniques that have been developed for relational algebra, it is safe to say that optimization techniques for relational algebra embedded in a general-purpose programming language are non-existent (one usually resorts to doing things like: if we are calling a relational expression from within an iteration, then factor out the call, *etc.*,). This is $X\alpha\rho\upsilon\beta\delta\iota\varsigma$ (Charybdis).

†This paper has appeared in the proceedings of the *3rd International Workshop on Database Programming Languages*, Naphlion, Greece, August 1991. Breazu-Tannen was partially supported by grants ONR NOOO-14-88-K-0634, NSF CCR-90-57570, and ARO DAAL03-89-C-0031PRIME. Buneman was partially supported by grants ONR NOOO-14-88-K-0634 and NSF IRI-86-10617, and by a UK SERC visiting fellowship at Imperial College, London.

Back to $\Sigma\kappa\upsilon\lambda\lambda\alpha$ (Scylla). Relational algebra has a simple and safe semantics, but where does it fall short? The main practical problem is that there is no way of moving outside flat relations; we cannot expect the relational algebra to produce a set of sets, an ordered list, or even an integer. In fact a lot of work, under the title of normalization theory, was done to end up with flat relations. The "band-aid" approach of throwing in *count, average, group-by,* etc., is unsatisfactory. On a more conceptual level, relational algebra does not "scale-up", and new primitives had to be invented to deal with the complex object model. Even so the main concern in the design of languages for the complex object model, has been absolute expressiveness [BK86, HS88, AB89]. The optimization techniques, and the concerns for correctness have yet to be discussed.

We will instead put forward a programming paradigm that tries to get close to both the semantic simplicity of relational algebra, and the expressive power of unrestricted programming languages. Its main computational engine is *structural recursion on sets.* All programming is done within a "nicely" typed lambda calculus, as in Machiavelli [OBB89]. A guiding principle is that *how* queries are implemented is as important as *if* they can be implemented. To summarize the main advantages:

- As in relational algebra, the meaning of any relation transformer is guaranteed to be a total map taking finite relations to finite relations. (see [BS91] for the denotational, and operational semantics of such languages, and for reasoning about such programs).
- Relational queries have a natural representation using structural recursion. In fact, as we shall see, a naturally restricted class of programs written with structural recursion has precisely the expressive power of the relational algebra/calculus (section 3).
- The same programming paradigm scales up, yielding query languages for the complex-object model [AB89]
- Beyond that, there are, for example, efficient programs for transitive closure (section 4). Moreover, we are also able to write programs that move out of sets, and then perhaps back to sets, as long as we stay within a (quite flexible) type system.
- The uniform paradigm of the language makes a strong suggestion that the optimization problem can be studied in such a language. The correspondence between restricted forms of structural recursion and the relational algebra indicates that we can import existing optimization techniques directly into the language. In fact, we expect that lower-level, and therefore better optimizations will be feasible. More general possibilites are suggested by lazy evaluation.
- The same programming paradigm scales up, yielding query languages for the complex-object model [AB89] (section 4).
- Finally, we believe that in this paradigm, particularly because of the good fit of the type system and the data model, the database manipulations can be better integrated with other programming features, even references and modules.

Concretely, we will use the following programming constructs.

$$\begin{array}{lrcl} \mathit{fun} & h(\emptyset) & = & e \\ \mid & h(\{x\}) & = & f(x) \\ \mid & h(S_1 \cup S_2) & = & u(h(S_1), h(S_2)) \end{array}$$

In combinator style, we will write $\Phi(e, f, u)$ for h. The typing is $\Phi(e, f, u) : \{\alpha\} \to \beta$ provided that $e : \beta$, $f : \alpha \to \beta$, and $u : \beta \times \beta \to \beta$.

$$\begin{array}{lrcl} \mathit{fun} & g(\emptyset) & = & e \\ \mid & g(\mathit{Insert}(x, S)) & = & i(x, g(S)) \end{array}$$

In combinator style, we will write $\Psi(e, i)$ for g. The typing is $\Psi(e, i) : \{\alpha\} \to \beta$ provided that $e : \beta$, and $i : \alpha \times \beta \to \beta$.

Intuitively, these definitions are based on the fact that all finite subsets of a set can be *generated* either by repeated unions starting from singleton sets (and add the empty set, which is also finite) or by repeated insertions of elements, starting from the empty set. In fact, these two kinds of definitions are not

independent, one can express each by the other (and the correspondence extends to the associated reasoning by structural induction) as shown in [BS91]. There is however a fundamental subtlety: the meaning of h is uniquely defined by either set of clauses above, but in order for this meaning to *exist*, we must require that the meanings of u and e form a commutative-idempotent monoid on the range of the meaning of $\Phi(e, f, u)$, respectively that on the range of the meaning of $\Psi(e, i)$, the meaning of i satisfies [BS91] (with a slight abuse of notation that confuses syntax and semantics)

$$i(x, i(y, S)) = i(y, i(x, S)) \quad (1)$$
$$i(x, i(x, S)) = i(x, S) \quad (2)$$

Using an obvious analogy, we will call condition (1) *commutativity*, and condition (2) *idempotence*. Such conditions make programming with sets more challenging, but also more interesting. Moreover, in our experience, verification of these conditions is often closely related to proving the correctness of the programs in question.

As an example, consider a program that takes a set and a linear order and returns a strictly ordered list of the elements of the set. We can write this with either one of the constructs presented above. With the first construct, the operation u is merging of strictly ordered lists; this is clearly associative, commutative and idempotent, and *nil* (the empty list) is an identity for merging. With the second construct, i is insertion in a strictly ordered list, which satisfies the equations (1) and (2). Both implementations use equality tests for the elements of the set, and thus are restricted to sets over equality types.

Throughout this paper we shall write our examples without type decorations/tags on primitive constructs, but one must assume that the decorations are there. These decorations can always be reconstructed from the context. For more clarity, we will occasionally specify the type of expressions. A more subtle problem is whether our primitives can be typed in a *polymorphic* manner (thus leaving the type decorations off). This is particularly challenging in the case of tuples/records (see [OB88] and references therein). One example of a rich polymorphic type system for relational databases is to be found in Machiavelli [OBB89], and its ideas can be equally successfully applied to the language we consider here. To improve readability we also occasionally write binary functions in infix notation (as we did with union).

2 First-order relational queries

We begin with the simple queries that are specified using disjunction, $F \vee G$ (provided F and G have exactly the same free variables), conjunction, $F \wedge G$ (provided F and G have no common free variables), and existential quantification $\exists x.F$. In relational algebra, these are implemented via *union*, *cartesian product* (as defined in relational algebra, not in set theory! also known as *disjoint join*) and *projection*.

Union is a primitive in our language too. To show how to implement the others with structural recursion we make the abbreviation $\Phi_\cup(f) = \Phi(\emptyset, f, \cup)$ where $f : \alpha \to \{\gamma\}$. This is correctly defined since $\cup$ and $\emptyset$ form a commutative-idempotent monoid. Moreover, we will use

$$\| : (\alpha_1 \times \cdots \times \alpha_m) \times (\beta_1 \times \cdots \times \beta_n) \to \alpha_1 \times \cdots \times \alpha_m \times \beta_1 \times \cdots \times \beta_n$$

for *tuple concatenation* and

$$\pi_I : \alpha_1 \times \cdots \times \alpha_n \to \alpha_{i_1} \times \cdots \times \alpha_{i_k} \qquad \text{where} \quad I = (i_1, \ldots, i_k)$$

for *tuple projection*.

Define

$$\begin{aligned} \mathit{map}\ f &= \Phi_\cup(\lambda x.\{f(x)\}) \qquad \mathit{map} : (\alpha \to \beta) \to \{\alpha\} \to \{\beta\} \\ \mathit{pairwith}\ S\ r &= \mathit{map}\ (\lambda s.r\|s)\ S \end{aligned}$$

$$\mathit{pairwith} : \{\beta_1 \times \cdots \times \beta_n\} \to \alpha_1 \times \cdots \times \alpha_m \to \{\alpha_1 \times \cdots \times \alpha_m \times \beta_1 \times \cdots \times \beta_n\}$$

$$\begin{aligned} R \times S &= \Phi_\cup(\mathit{pairwith}\ S)\ R \\ \Pi_I(R) &= \mathit{map}\ \pi_I\ R \end{aligned}$$

Looking at query specification from the vantage point provided by a language like the one we are considering, we naturally take a more computational view of query *safety*. For example, since we do not

want to rule out working with databases of algorithms, *i.e.*, sets of tuples of functions, we regard equality as domain specific. This is why we have isolated above the disjoint join, which does not require equality on the domain. The query specifications we have considered so far do not allow constructions like $R(x,x)$, $R(x,3)$ and $R(x,y) \wedge R(y,z)$. Note, however, that these formulas are equivalent to $R(x,y) \wedge x = y$, $\exists y.R(x,y) \wedge y = 3$, and $R(x,y) \wedge R(w,z) \wedge w = y$. We take the view that "variable coincidence" which is so easily expressed in logical languages, but can be—depending on the domain—nontrivial computationally, is treated explicitly via equality in the domain. We consider formulas like $y = 3$, and $w = y$ to be domain specific, and we will perform a conceptual *domain abstraction*, by encapsulating everything that is specific to it in formulas of a *domain logic.* [1] We do not care really what form this logic takes, but we care that only formulas defining *computable* predicates on the domain be used. If F is a query specification, φ is a formula in the associated domain logic, and all the free variables of φ are already in F (are *limited*), then $F \wedge \varphi$ is a query specification. In relational algebra this corresponds to *selection*, and in our language selection it is implemented by

$$\begin{array}{rcl} \mathit{filter}\ p & = & \Phi_\cup(\lambda r.\ \mathit{if}\ p(r)\ \mathit{then}\ \{r\}\ \mathit{else}\ \emptyset) \\ \sigma_p(R) & = & \mathit{filter}\ p\ R \end{array} \qquad \mathit{filter} \ : \ (\alpha \to \mathit{bool}) \to \{\alpha\} \to \{\alpha\}$$

where R implements F and p is the implementation of the domain specific selection formula φ.

One can conceive of more flexible ways to limit the variables of the selection formula. For example, $\exists y.R(x,y) \wedge y > 1 \wedge z = y - 1$ is perfectly safe. We note however, that this only amounts to a little additional computation on the query $R(x,y) \wedge y > 1$, computation that is easily expressible in our language.

What about negation (corresponding in relational algebra to *difference*)? Implementing it requires membership testing, therefore equality. The ability to specify negation in queries is therefore domain specific (we can't, for example, do it in a database of algorithms.) To emphasize this point, we would like to see the specification of negation as a particular case of the domain specific selection that we just mentioned. This follows, if we regard membership as an *aggregate operator.* In general, aggregate operators map relations to other types, related to the domain. Hence, aggregate operators are a kind of "feedback" to selection, allowing the relations to contribute to the domain specific logic. With this

$$R \setminus S \ = \ \sigma_{\lambda r.\neg(\mathit{member}\ S\ r)}(R)$$

To implement $\mathit{member} : \{\alpha\} \to (\alpha \to \mathit{bool})$ we use the abbreviation $\Phi_\vee(Q) = \Phi(\mathit{false}, Q, \vee)$ where $Q : \alpha \to \mathit{bool}$. This is again correctly defined since $\vee$ and *false* form a commutative-idempotent monoid. Then

$$\mathit{member}\ S\ r \ = \ \Phi_\vee(\lambda s.s = r)\ S$$

3 Structural recursion and the relational algebra

We have seen in the previous section that only a limited use of structural recursion was needed to implement the operators of the relational algebra. The empty set, the union, and the *relation variables* (*i.e.*, variables of type set of tuples) are common to the two formalisms. The implementation of (disjoint) join and projection used only $\Phi_\cup$ and concatenation and projection of tuples. Selection used in addition *if-then-else*, and finally difference used $\Phi_\vee$ and equality of tuples.

We will show in this section that, in fact, this is a tight correspondence: the *project-join* algebra, the *positive* relational algebra (add selection) and the full relational algebra are, respectively, semantically equivalent to certain sublanguages of our language, organized around the operations mentioned above. In order to perform this translation uniformly we must augment the relational algebra with relation and tuple variables. Its syntax is given by:

$$\mathcal{A} ::= v \mid \emptyset \mid \{\tau\} \mid \mathcal{A} \cup \mathcal{A} \mid \Pi_I(\mathcal{A}) \mid \mathcal{A} \times \mathcal{A} \mid \sigma_{\lambda t.P}\mathcal{A} \mid \mathcal{A} \setminus \mathcal{A}$$

where v ranges over relation variables, and t ranges over tuple variables. The *project-join* algebra is the sublanguage of $\mathcal{A}$ generated without selection and difference; and the *positive* algebra is the sublanguage generated without difference.

[1] This leads to the interesting issue of the relationship between safety conditions on one hand, and parametric polymorphic data abstraction on the other hand.

First, we give the sublanguage of our typed calculus with "union-recursors", which corresponds to the project-join algebra. The syntax of tuple expressions is given by (t ranges over tuple variables)

$$\tau ::= t \mid \tau \| \tau \mid \pi_I(\tau)$$

The expressions in this first sublanguage are given by (v ranges over relation variables)

$$\mathcal{E} ::= v \mid \emptyset \mid \{\tau\} \mid \mathcal{E} \cup \mathcal{E} \mid \Phi_\cup(\lambda t.\mathcal{E})\ \mathcal{E}$$

All these expresions are assumed to be well-typed according to the conventions stated before.

We will now give a translation that associates to each expression $\mathcal{E}$ in the sublanguage an expression $\overline{\mathcal{E}}$ in the project-join algebra (enriched with singleton sets of tuple variables).

$$\begin{array}{rcl}
\overline{v} & = & v \\
\overline{\emptyset} & = & \emptyset \\
\overline{\{t\}} & = & \{t\} \\
\overline{\{\tau_1 \| \tau_2\}} & = & \overline{\{\tau_1\}} \times \overline{\{\tau_2\}} \\
\overline{\{\pi_i(\tau)\}} & = & \Pi_i(\overline{\{\tau\}}) \\
\overline{E_1 \cup E_2} & = & \overline{E_1} \cup \overline{E_2} \\
\overline{\Phi_\cup(\lambda t.E_1)\ E_2} & = & \overline{E_1}[\overline{E_2}/\{t\}]
\end{array}$$

In the last of these, $\overline{E_1}[\overline{E_2}/\{t\}]$ means substitute $\overline{E_2}$ for all occurences of the subexpression $\{t\}$ in $\overline{E_1}$. It is readily checked that the translation is well-defined, that expressions without free tuple variables are translated into expressions without free tuple variables, and that the free relation variables are exactly preserved by the translation.

An obvious property of this translation is that if there are no free tuple variables in E, then there are no tuple variables in $\overline{E}$, i.e. the result of translating a combinator (with respect to tuple variables) is an expression in what is normally regarded as relational algebra. To show that, in general, this translation preserves meaning we have to describe the meaning of expressions with free (relation or tuple) variables, so we take their meanings are functions from valuations (environments) for these variables to values. By thinking of the semantic domain as the domain of such functions on relation variables, we need not worry about these variables, since they cannot get bound in these restricted languages. However, because tuple variables get bound, we must, as usual, prove something about expressions with free tuple variables.

Lemma 3.1 *For any tuple variable environment ρ, and any expression E*

$$[\![E]\!]\rho = [\![\overline{E}]\!]\rho$$

Proof sketch. By induction on the structure of E. The only non-trivial step involves the translation of $\Phi_\cup(\lambda t.E_1)\ E_2$ and follows from two equalities. First, for any E_1 and E_2

$$[\![\Phi_\cup(\lambda t.E_1)\ E_2]\!]\rho = \bigcup\{\ [\![E_1]\!]\rho[r/t] \mid r \in [\![E_2]\!]\rho\ \}$$

which is shown by induction on E_2. Second, for any two expressions A_1, A_2 in the project-join algebra (enriched with singleton sets of tuple variables)

$$[\![A_1[A_2/\{t\}]]\!]\rho = \bigcup\{\ [\![A_1]\!]\rho[r/t] \mid r \in [\![A_2]\!]\rho\ \}S$$

and this is shown by induction on A_1. (Here, $\rho[r/t]$ is the environment which maps t to r and all other tuple variables s to $\rho(s)$.) □

In section 2, we have shown how to translate expressions of the join-project algebra, without free tuple variables into $\mathcal{E}$-expressions without free tuple variables. That translation is easily shown to preserve meaning. Putting the two translations together we obtain

Theorem 3.2 *$\mathcal{E}$-expressions without free tuple variables can be translated into semantically equivalent expressions in the project-join algebra, and conversely.*

Note that the translations are not, in general inverses. Also note that the translations are not dependent on the relations being flat: the tuple expressions are polymorphic in the types of the components.

We now add conditionals to the language of $\mathcal{E}$-expressions, and show that this corresponds to adding selection to the project-join algebra. We define the larger sublanguage

$$\mathcal{E}' ::= v \mid \emptyset \mid \{\tau\} \mid \mathcal{E}' \cup \mathcal{E}' \mid \Phi_\cup(\lambda t.\mathcal{E}')\ \mathcal{E}' \mid \textit{if } P \textit{ then } \mathcal{E}' \textit{ else } \mathcal{E}'$$

where P ranges over predicate expressions, of which we need stipulate only that they be of type *bool*, that their free variables be tuple variables, and that if P is a predicate, so is its negation, $\neg P$.

Since $\textit{if } P \textit{ then } E_1 \textit{ else } E_2$ is equivalent to $(\textit{if } P \textit{ then } E_1 \textit{ else } \emptyset\ ) \cup (\textit{if } \neg P \textit{ then } E_2 \textit{ else } \emptyset\ )$, it is sufficient to concern ourselves with expressions of the form $\textit{if } P \textit{ then } E \textit{ else } \emptyset$. We will translate the $\mathcal{E}'$-expressions into the expressions of the positive relational algebra (the project-join algebra extended with selection) again enriched with singleton sets of tuple variables.

The translation is as before, with the addition of the translation of expressions of the form $\textit{if } P \textit{ then } E \textit{ else } \emptyset$. Suppose the free (tuple) variables of P are $t_1, \ldots, t_k$. Define

$$\overline{\textit{if } P \textit{ then } E \textit{ else } \emptyset} = \Pi_{I_E}(\sigma_{\lambda t.\overline{P}}(\overline{E} \times \{t_1\} \times \cdots \times \{t_k\})$$

where I_E and $\overline{P}$ are defined as follows. Let $\overline{E}$ be of type $\{\alpha_1 \times \cdots \times \alpha_m\}$ and then let t_E be a tuple variable of type $\alpha_1 \times \cdots \times \alpha_m$. Then I_E is the index vector such that $t_E = \pi_{I_E}(t_E \| t_1 \| \cdots \| t_k)$. Moreover, $\overline{P} = P[\pi_{I_1}(t)/t_1, \ldots, \pi_{I_k}(t)/t_k]$ where I_i is the index vector such that $t_i = \pi_{I_i}(t_E \| t_1 \| \cdots \| t_k)$.

As a result of this translation, $\sigma_{\lambda t.\overline{P}}$ is, as required by the algebra, defined over a combinator. This means that there are no free variables in $T(P)$ to interfere with the substitution that occurs in the translation of $\Phi_\cup(\ldots,\ldots)$. Thus we are in a position to use our existing machinery to establish the equivalence of our language $\mathcal{E}'$ with the relational algebra extended with expressions of the form $\sigma_{\lambda t.P}(A)$. We have

Theorem 3.3 *$\mathcal{E}'$-expressions without free tuple variables can be translated into semantically equivalent expressions in the positive relational algebra, and conversely.*

Note that, provided our predicates contain equality, intersection can be defined as

$$E_1 \cap E_2 = \Phi_\cup(\lambda t_1.\Phi_\cup(\lambda t_2.\textit{if } t_1 = t_2 \textit{ then } \{t_1\} \textit{ else } \emptyset\ )\ E_2)\ E_1$$

This is the first point in our discussion of the equivalence of languages that we need to assume equality on the underlying domain.

Our selection predicates have so far been limited to boolean expressions built up from predicates on tuples. We now consider what happens if we allow predicates on sets, i.e. we add to our language $\mathcal{E}'$ predicates using the "or-recursor" $\Phi_\vee(\lambda t.P)\ E$. From the equivalence of this expression (using the obvious semantics) and

$$\neg\ \text{Empty}(\Phi_\cup(\lambda t.\textit{if } P \textit{ then } \{t\} \textit{ else } \emptyset\ )\ E)$$

we see that the or-recursor buys us nothing more than an emptiness test, and we may as well consider extending the predicates in our language $\mathcal{E}'$ with expressions of the form $\text{Empty}(E)$, which now makes the language non-monotonic.

Note that, if nothing else, the type system of our language dictates that the only place a predicate such as $\text{Empty}(E)$ can occur is within a conditional. The translation of the simplest conditional containing an emptiness test is given by

$$\overline{\textit{if } \text{Empty}(E_1) \textit{ then } E_2 \textit{ else } \emptyset} = \overline{E_2} \setminus \Pi_{I_2}(\overline{E_2} \times \overline{E_1})$$

where I_2 is the vector of the first $\text{arity}(E_2)$ indices. It is immediate that this translation preserves meaning. To complete the translation we must now work inductively on the structure of the predicate in a conditional, for example:

$$\begin{aligned}
\overline{\textit{if } P_1 \wedge P_2 \textit{ then } E \textit{ else } \emptyset} &= \overline{\textit{if } P_1 \textit{ then } E \textit{ else } \emptyset} \cap \overline{\textit{if } P_2 \textit{ then } E \textit{ else } \emptyset} \\
\overline{\textit{if } \neg P \textit{ then } E \textit{ else } \emptyset} &= \overline{E} \setminus \overline{\textit{if } P \textit{ then } E \textit{ else } \emptyset} \\
&\ldots
\end{aligned}$$

We therefore obtain the main and final result of this section:

Theorem 3.4 *$\mathcal{E}'$-expressions, augmented with or-recursors in their predicates, and without free tuple variables can be translated into semantically equivalent expressions in the relational algebra, and conversely.*

4 Complex objects, transitive closure, and grouping

Without going into the details of the complex object model, we remark that it makes a good match with our language's type system [BJO89, OBB89]. In this section we will first show that our language is at least as expressive as Abiteboul and Beeri's algebra (and therefore calculus) for complex objects [AB89]. We have already shown how to express cartesian product and difference(section 2). The Abiteboul-Beeri algebra has an operation called *replace* which combines the relational algebra's selection and projection operations, moreover allowing algebraic operations to be applied recursively to subobjects. We implement it simply as

$$replace\ p\ f\ S = map\ f\ (filter\ p\ S)$$

The ability to apply algebraic operations recursively to sub objects is provided automatically by our type system: f and p can at their turn contain other program constructs.

Finally, the Abiteboul-Beeri algebra features two truly higher-order operations: *powerset* (self-explanatory) and *collapse* which maps a set of sets into their union. We implement these as follows.

$$collapse = \Phi_{\cup}(\lambda S.S)$$

$$collapse : \{\{\alpha\}\} \to \{\alpha\}$$

$$powerset = \Phi(\{\emptyset\}\ ,\ \lambda x.\{\emptyset\} \cup \{\{x\}\}\ ,\ \lambda(S_1, S_2).\ map\ \cup\ cartprod(S_1, S_2))$$

$$powerset : \{\alpha\} \to \{\{\alpha\}\}$$

To see that the commutative-idempotent monoid requirement is satified for the definition of powerset, note that semantically

$$map\ \cup\ cartprod(S_1, S_2) = \{\ A_1 \cup A_2\ \mid\ A_1 \in S_1\ ,\ A_2 \in S_2\ \}$$

Abiteboul and Beeri show that this (quite small) collection of operations is equivalent to a powerful higher-order logical calculus of nested tuples and sets. Interestingly, they also show that **transitive closure** (which, as shown by Aho and Ullman [AU79], cannot be implemented in relational algebra), can, in fact, be specified as a query in this calculus: it is the least transitive relation containing the given one among all subsets of the (cartesian) squaring of the set of all elements that occur in the relation. Since in our language we can implement the Abiteboul-Beeri algebra, we can also implement their calculus, hence transitive closure. Unfortunately, it is clear that the resulting algorithm is severely inefficient. However, we will show next that the language we consider can express a much better algorithm for transitive closure, using structural recursion on the empty-insert presentation of sets.First we will need relation composition, which is in fact expressible in relational algebra $R \circ S = \Pi_{(1,4)}(\sigma_{\lambda t.\pi_2(t)=\pi_3(t)}(R \times S))$. Now, consider $i : (\alpha \times \alpha) \times \{\alpha \times \alpha\} \to \{\alpha \times \alpha\}$ defined by

$$i(r, T) = \{r\}\ \cup\ T\ \cup\ \{r\} \circ T\ \cup\ T \circ \{r\}\ \cup\ T \circ \{r\} \circ T$$

and then

$$\begin{array}{lll} fun & TC(\emptyset) & = \emptyset \\ \mid & TC(Insert(s, R)) & = i(s, TC(R)) \end{array}$$

We will have to verify that this is correctly defined, that is, that the semantics of i satisfies the commutativity and idempotence conditions (see section 1) on the right set of values, and moreover, that the meaning of TC is in fact the transitive closure operator. In what follows we will perpetrate a slight abuse of notation by writing semantic proofs of semantic facts in programming syntax. (In fact, the proofs for the next lemma can all be formalized in syntax too, by using one of the logics described in [BS91].) We still need one more notation: the *semantic* transitive closure is denoted by $R \longmapsto R^+$.

Lemma 4.1 *1. $\emptyset$ is transitive. If T is transitive then $i(r, T)$ is also transitive.*

2. Let T be transitive. Then $i(r, i(s, T)) = i(s, i(r, T))$ and $i(r, i(r, T)) = i(r, T)$.

3. If T is transitive then $i(r, T) = (\{r\} \cup T)^+$.

4. $i(r, R^+) = (\{r\} \cup R)^+$.

The key observation in proving part 1 is the following simple fact: for *any* R, $\{s\} \circ R \circ \{s\} \subseteq \{s\}$. Part 2, which implies that TC is correctly defined, (working with arange consisting only of transitive relations), is shown using part 1. [2] Part 3 follows immediately from part 1, and part 4 from part 3.Part 4 of the lemma is the essential step in showing by structural induction on the empty-insert presentation of sets, that for any R, $TC(R) = R^+$. This algorithm resembles Warshall's algorithm, except that we are doing edge insertion rather than node insertion. To actually obtain Warshall's algorithm, suppose we are given a set of nodes $V : \{\alpha\}$ and a set of edges $E : \{\alpha \times \alpha\}$.

$$\begin{array}{lrcl} \mathit{fun} & W(\emptyset) & = & E \\ \mid & W(\mathit{Insert}(v, A)) & = & W(A) \cup W(A) \circ \{\langle v, v\rangle\} \circ W(A) \end{array}$$

One can show that W is correctly defined and that for any $A \subseteq V$, $W(A)$ is the set of pairs of nodes which are connected by paths whose intermediate nodes all belong to A. It follows that $W(V)$ gives the desired transitive closure. Warshall's algorithm runs in $O(n^3)$ time while the edge insertion algorithm runs in $O(en^2)$ time(n is the number of nodes and e is the number of edges).In any case, these are efficient algorithms for transitive closure (compare with the complex object algebra query mentioned before). In the spirit of Warshall's algorithm, one can also represent Floyd's shortest paths algorithm.Abiteboul and Beeri also show that their calculus (and algebra) can simulate **grouping** which is an operation akin to the one obtaining a set-valued function out of a relation. Given a complex object $R : \{\alpha_1 \times \alpha_2\}$, and a "domain" for its first projection, *i.e.*, $D : \{\alpha_1\}$ such that $\Pi_1(R) \subseteq D$, grouping returns the complex object of type $\{\alpha_1 \times \{\alpha_2\}\}$ whose meaning is

$$[\![\mathit{grouping}(D, R)]\!] = \{\langle z, T\rangle \mid z \in D \text{ and } T = \{y \mid \langle z, y\rangle \in R\}\}$$

Aggregation by groups is a useful feature, especially in conjunction with other aggregate operators. It is somewhat ironical that whereas query languages are set oriented, the relational data model deals only with flat relations. In many applications, one needs to construct a set of elements satisfying certain properties (e.g., all parts supplied by a supplier), to be subsequently manipulated by some computation (e.g.,find total cost of all parts used in a composite part). Indeed, the plethora of "explosion" diagrams in any industrial catalog, points to the ever present need for such an operation. All practical query languages introduce a grouping operation, and this operation plays a central role in LDL [NT89], which is based on a complex object model. As in the case of transitive closure we give a direct implementation of grouping, which avoids the use of powerset. To do so, first we define

$$\begin{array}{lrcl} \mathit{fun} & g_D(\emptyset) & = & D \times \{\emptyset\} \\ \mid & g_D(\mathit{Insert}(\langle x, y\rangle, S)) & = & \mathit{map}\ f\ g_D(S) \end{array}$$

where f stands for

$$\lambda\langle z, T\rangle.\ \mathit{if}\ z = x\ \mathit{then}\ \langle z, \mathit{Insert}(y, T)\rangle\ \mathit{else}\ \langle z, T\rangle$$

and then we take $\mathit{grouping}(D, R) = g_D(R)$. It is not hard to see that this definition is correct (that is, the commutativity and idempotence conditions are satisfied), since the range of g_D here consists only of graphs of functions $D \longrightarrow \{\alpha_2\}$;. In connection with other work [BNST87], we remark that the grouping operator along with the empty-insert presentation of sets gives interesting expressions for negation, difference and union ofsets.

5 Pump, partition, and hom

FAD [BBKV88], LDL [NT89] and Machiavelli [OBB89] all have a construct related to structural recursion on sets. FAD and Machiavelli come closest with the operators *pump*, respectively *hom*, which are equivalent to the following

$$\begin{array}{lrcll} \mathit{fun} & h(\emptyset) & = & e & \\ \mid & h(\{x\}) & = & f(x) & \\ \mid & h(S_1 \cup S_2) & = & u(h(S_1), h(S_2)) & \text{where } S_1 \cap S_2 = \emptyset \end{array}$$

[2] For all we know, i may be commutative and idempotent on all relations, not only on the transitive ones, but checking this fact seems to be better left to a machine! The restriction to transitive relations is sufficient for our purposes.

and where u is not required to be idempotent (but, of course, u and e must form a commutative monoid). LDL defines a predicate $partition(S_1, S_2, S_3)$ which imposes the disjointness of S_1 and S_2. For example consider the rules for computing the sum of a set of integers:

$$\begin{aligned} sum(\emptyset, 0) &:- \ . \\ sum(\{n\}, n) &:- \ . \\ sum(S, n) &:- \ partition(S_1, S_2, S) \ , \ sum(S_1, n_1) \ , \ sum(S_2, n_2) \ , \ n = n_1 + n_2 \ . \end{aligned}$$

This style of programming was found to be very useful in defining aggregates, such as sum, count, average *etc.*, . *Pump*, *hom*, and LDL constructs based on *partition* as above, have a natural denotational semantics. The problem is that their operational semantics is quite contrived. In the case of *pump* and *hom*, the evaluator must evaluate sets eagerly and then do time consuming dynamic tests for equality of values. Of course,this rules out working with sets of functions for example. Even for sets of, say, integers, mapping a function over a disjoint union may yield a non-disjoint one, which fed into *hom* would yield a run-time error. One would like to obtain statically an assurance that the program goes through, but it seems that only a few very simple programs can be shown correct in this sense. The operational semantics of *partition* is such that all possible partitions are generated (whereas any partition will do).

We can, however, express the same functions without using such problematic constructs, and by staying within the framework we have described. One way of doing this would be to replace the definition of h above with [3]

$$\begin{aligned} fun \quad & h(\emptyset) &=& \ e \\ | \quad & h(Insert(x, S)) &=& \ \textit{if } x \in S \textit{ then } h(S) \textit{ else } u(f(x), h(S)) \end{aligned}$$

A cleaner method is to convert sets to *bags* and then to do structural recursion on bags. (see [BS91] for such recursion constructs). Indeed, this is suggested by the fact that such programs plus dropping the disjointness/partition conditions work just fine for computing aggregate operators on bags. It is then sufficient to program the fundamental function that coerces a set into a bag. To do this by structural recursion we need the appropriate commutative-idempotent monoid structure on bags. This is given by the "max" operation, and implementing it requires that the elements of the underlying type have equality. We would like to note, however, that perhaps one of the most interesting features about using structural recursion on sets as advocated in this paper (with idempotent operations) is the ability to program flexibly with objects which lack equality (such as relations of algorithms) while this is quite restricted in the other languages mentioned.

6 Further research

We expect that the linguistic techniques proposed here will also be applicable in dealing with *incomplete objects* [INK91], a data model that captures the ideas of incomplete specifications. In particular, we are interested in studying the semantic properties of *or-objects* as sets of possible worlds.

The transitive closure algorithms (section 4) seem to underscore the ability of structural recursion to represent efficient interesting algorithms. We intend to investigate other such representations. In addition to transitive closure, and generalizing that idea, Abiteboul and Beeri show that their complex object calculus can simulate (stratified) recursive queries. We conjecture that their calculus should be able to simulate structural recursion, hence that their calculus and our language are equivalent in terms of absolute expressiveness. One of the main points of this paper, however, is that structural recursion may allow the implementation of better algorithms (for the same functionality). In particular, we intend to investigate ways of transforming recursive queries into efficient programs with structural recursion. Our paper demonstrates that structural recursion yields *finer grain* programming than relational or complex object algebras. This should allow for importing all classical query optimization techniques, and, in principle, for more optimizations. Searching for such optimizations is perhaps the most important topic for further research here. Optimizations could

[3] This is apparently a more general form of definition than the one we used so far, since in addition to $h(S)$, we also have separate occurrences of S on the right hand side of the second clause. The difference is similar to the one between iteration and primitive recursion in defining arithmetic functions. Kleene's technique for representing the predecessor function in the lambda calculus, which uses pairing, can also be applied here and we can express this more flexible form of definition in terms of the one we gave originally.

be based on semantic equalities such as

$$\textit{filter } p\ (\Phi_\cup(\lambda r.S)\ R) \quad = \quad \Phi_\cup(\lambda r.\ \textit{filter } p\ S)\ R \tag{3}$$

$$\Phi_\cup(\lambda r.S)\ (\textit{filter } p\ R) \quad = \quad \Phi_\cup(\lambda r.\ \textit{if } p(r) \textit{ then } S \textit{ else } \emptyset)\ R \tag{4}$$

For example, using these identities, we can perform the following classic optimization. Using (3) twice

$$\sigma_p(R \times S) = \textit{filter } p\ (\Phi_\cup(\lambda r.\ \Phi_\cup(\lambda s.\{r||s\})\ S)\ R) = \Phi_\cup(\lambda r.\ \Phi_\cup(\lambda s.\ \textit{filter } p\ \{r||s\})\ S)\ R$$

Now, suppose that p only test the components of $r||s$ which are in s, for example, $r : \alpha_1 \times \cdots \times \alpha_m$, $s : \beta_1 \times \cdots \times \beta_n$ and p is $\lambda t.\pi_{m+k}(t) = 0$. Then, taking p' to be $\lambda s.\pi_k(s) = 0$, the expression is further equivalent to $\Phi_\cup(\lambda r.\ \Phi_\cup(\lambda s.\ \textit{if } p'(s) \textit{ then } \{r||s\} \textit{ else } \emptyset)\ S)\ R$. Using (4) we get

$$\Phi_\cup(\lambda r.\ \Phi_\cup(\lambda s.\{r||s\})\ (\textit{filter } p'\ S))\ R = R \times \sigma_{p'}(S)$$

which is cheaper to compute than $\sigma_p(R \times S)$.

Another example would be to replace $\Phi_\cup(\lambda r.\ \textit{map } f\ S)\ R$ with the equivalent $\textit{map } f\ (\Phi_\cup(\lambda r.S)\ R)$. The following identity may also yield optimizations

$$\Phi_\cup(\lambda s.T)\ (\textit{map } f\ R) = \Phi_\cup(\lambda r.\ T[f(r)/s])\ R$$

Clearly, more work is needed, especially in investigating more general constructs than $\Phi_\cup$. In a different vein, we intend to investigate optimizations that would result from a *lazy*evaluation of set expressions.

References

[AB89] S. Abiteboul and C. Beeri. *On the power of languages for the manipulation of complexobjects.* Technical Report, INRIA, 1988.

[ACO83] Albano. A., L. Cardelli, and R. Orsini. *Galileo: A strongly typed, Interactive Conceptual Language.* Technical Report, Bell Laboratories, Bell Telephone Laboratories, Internal Technical document Services, Murray Hill 1b-509, NJ, USA, 1983.

[AU79] A. Aho and J. Ullman. Universality of data retrieval languages. In *Proceedings of POPL*, 1979.

[BBKV88] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, a powerful and simple database language. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 97–105, 1988.

[BJO89] P. Buneman, A. Jung, and A. Ohori. Using powerdomains to generalize relational databases. *Theoretical Computer Science*, To Appear, 1989. Available as a technical report from Department of Computer and Information Science, University of Pennsylvania.

[BK86] F. Bancilhon and S. Khoshafian. A calculus for complex objects. In *Proc. ACM Symposium on Principles of Database Systems*, 1986.

[BNST87] C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur. Set constructors in a logic database language. In *Proceedings of PODS*, 1987. Full paper to appear in Journal of Logic Programming.

[BS91] V. Breazu-Tannen and R. Subrahmanyam. Logical and computational aspects of programming with lists/bags/sets. In *Proceedings of ICALP*, 1991. To appear.

[HS88] R. Hull and J. Su. On the expressive power of database queries with intermediate types. In *Proceedings of PODS*, 1988.

[INK91] T. Imielinski, S. Naqvi, and Vadaparty K. Incomplete objects—a data model for design and planning applications. In *Proceedings of SIGMOD*, 1991. To appear.

[NT89] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases.* Computer Science Press, 1989.

[OB88] A. Ohori and P. Buneman. Type inference in a database programming language. In *Proc. ACM Conference on LISP and Functional Programming*, pages 174–183, Snowbird, Utah, July 1988.

[OBB89] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli – a polymorphic language with static type inference. In *Proceedings of the ACM SIGMOD conference*, pages 46–57, May – June 1989.

[Sch77] J.W. Schmidt. Some high level language constructs for data of type relation. *ACM Transactions on Database Systems*, 5(2), 1977.