On Aggregation and Computation on Domain Values

Kim S. Larsen*
Aarhus University

September 1992

Abstract

Query languages often allow a limited amount of arithmetic and string operations on domain values, and sometimes sets of values can be dealt with through aggregation and sometimes even set comparisons. We address the question of how these facilities can be added to a relational language in a natural way. Our discussions lead us to reconsider the definition of the standard operators, and we introduce a new way of thinking about relational algebra computations.

We define a language FC, which has an iteration mechanism as its basis. A tuple language is used to carry out almost all computations. We prove equivalence results relating FC to relational algebra under various circumstances.

1 Introduction

In relational calculus, domain values and tuples can be handled elegantly as (one of) these are chosen as the basic entity. This also means that arithmetic and other operations on domain values can be added in a natural way. But at

*Some of this work was done while visiting the University of Toronto

the level of domain values, relations do not seem to fit in very well, making it more cumbersome to have (and define) aggregation. When this has been done [Klu82], calculus queries often seem to produce slightly different answers than the "natural" algebra counterpart—this is especially true when aggregating over empty sets.

In relational algebra, relations are chosen as the basic entity. This means that arithmetic, for example, is added in an unnatural and restricted form. This has been done through operators such as extend [Gra84], making queries like **extend** r **by** $A := B + C - 3$ possible. The intention, here, is that each tuple in r (the schema of which contains B and C) is extended with a new attribute, A, defined using the original values of each tuple. From a programming language point of view, this is unsatisfactory because the two fundamental concepts of iteration and performing arithmetic are bound tightly together instead of being separate operations. This becomes even more conspicuous as a bad language design when one realizes that select and project (and conceptually also rename) are also mixtures of an iteration mechanism and operations which are inherently tuple/domain value oriented.

On the other hand, aggregation fits very well into the algebra framework; at least at first glance: relations are the basic types, so what is more natural than applying set functions to these? Unfortunately, as aggregate functions return domain values, we get a problem similar to the difficulties with extend. In [Gra81, Klu82], something like **group** r **by** Dept **creating** TotalPay := sum(NetPay) is suggested. A new attribute TotalPay is added (and Dept is deleted) and given a value, which is the sum of the NetPay fields having the same Dept value. As a next step, one wants to perform arithmetic using the aggregate values. Such an operator can be constructed, of course (see ASTRID [Gra84, GB79], for example), but what becomes more and more apparent is that a general iteration mechanism is needed.

This discussion leads to the following: if we want a language with aggregation and operations on domain values, we would want our language to be equipped with an iteration mechanism as the basic operator. For use in this iteration, we need a powerful and flexible tuple language which should be used for the tuple based relational manipulations and for arithmetic etc.

Let us point out that by an iteration mechanism we do not mean a while construct, as it can be found in an ordinary programming language, because

this would increase the computational power. Rather, we are thinking of something like a **for_all_do** operator which would be able to iterate through a set of tuples in an order which is not predetermined. It is of great interest to extend relational algebra in the direction of adding more computational power, but this should be a separate decision; not a side-effect of the decisions concerning the issues under consideration here.

We propose the language FC, which is an acronym for *factorize and combine*. The basis of the language is a general iteration mechanism for iterating through tuples and "small" relations from multiple relational arguments (where there is only a single argument, this is similar to **group_by**). An iteration is initiated through the use of the keyword **factor** (not to be confused with the **factor** operator from [Mai83]). The relational arguments are factorized, i.e., decomposed to (smaller) components. All (real) computations are performed via a tuple language. The relational operator, Cartesian product (and an implicit union), is used to build up the desired output relation after the necessary computation has been performed.

The FC language has at least as good run-time complexities as relational algebra. This is described in [LSS92]. In addition, a new optimization technique for unary queries can be applied [LS]. These complexity aspects are not treated in this paper.

The translations given in this paper are only given in order to prove the expressive equivalence of FC and relational algebra. An actual implementation of FC, which has been carried out for a simple version [Lar92], should not be done via relational algebra.

Just as relational algebra and relational calculus have formed a basis for the implementation of many traditional languages, we believe that FC would be a natural basis for the definition of new languages which include aggregation and computation on domain values.

In the following section, we introduce FC using examples. The rest of the paper is more technical. It is devoted to proving that FC and relational algebra are equivalent with respect to expressive power.

2 Examples

Firsts we will account for the concept of *factorization of relations*. Let two relations, r_1 and r_2, be as listed below.

r_1:

A	B
a_1	b_1
a_2	b_2
a_3	b_3

r_2:

B	C	D
b_1	c_1	d_1
b_2	c_2	d_2
b_3	c_3	d_3
b_4	c_4	d_4

First, we note that $R_1 \cap R_2 = \{B\}$, and that the (tuple) values in the B-columns are $\pi_B(r_1) \cup \pi_B(r_2) = \{[b_1], [b_2], [b_3], [b_4]\}$. We shall write r_1 and r_2 as a combination of the B-tuples. Clearly, r_1 can be written as

$$\{[b_1]\} \times \{[a_1]\} \cup \{[b_2]\} \times \{[a_2]\} \cup \{[b_3]\} \times \{[a_3]\} \cup \{[b_4]\} \times \{\ \}$$

and r_2 can be written as

$$\{[b_1]\} \times \{\ \} \cup \{[b_2]\} \times \{[c_1, d_1], [c_2, d_2]\} \cup \{[b_3]\} \times \{[c_3, d_3]\} \cup \{[b_4]\} \times \{[c_4, d_4]\}$$

We call this the *factorization* of r_1 and r_2 on B.

The name factorization was chosen because of this operation's resemblance to the factorization of integers: if, for example, 315 (corresponding to r_2) is factorized with respect to the first four primes (corresponding to $[b_1], [b_2], [b_3], [[b_4])$, we get $2^0 \cdot 3^2 \cdot 5^1 \cdot 7^1$ where the exponents correspond to the cardinality of the relations in the example (of r_2).

Now, we define a certain class of *environments*, which depend on a factorization. As usual an environment is simply an association of values with identifiers. As an example, consider the environment

$\eta_{[b_2]}$:

$$\begin{array}{lcl} \# & \mapsto & [b_2] \\ @(1) & \mapsto & \{[a_2]\} \\ @(2) & \mapsto & \{[c_1, d_1], [c_2, d_2]\} \end{array}$$

In this example, we have three identifiers, #, @(1), and @(2), each of which has an associated value. The reader has probably noticed that this environment consists of the tuple $[b_2]$ together with the relations consisting of the tuples from r_1 and r_2 which contain $[b_2]$; except that in @(1) and @(2), the $[b_2]$ part of the tuples has been removed. Similarly, we have

$\eta_{[b_4]}$:

#	$\mapsto$	$[b_4]$
@(1)	$\mapsto$	$\{\ \}$
@(2)	$\mapsto$	$\{[c_4, d_4]\}$

We can evaluate expressions in different environments. As an example $@(1) \times @(2)$ evaluated in $\eta_{[b_2]}$ (this is denoted $[\![@(1) \times @(2)]\!]\eta_{[b_2]}$) is $\{[a_2, c_1, d_1], [a_2, c_2, d_2]\}$. similarly, $[\![@(1) \times @(2)]\!]\eta_{[b_4]}) = \{\ \}$. Because of the way the four environments $\eta_{[b_1]}, \eta_{[b_2]}, \eta_{[b_3]}$, and $\eta_{[b_4]}$ are defined, $[\![@(1) \times @(2)]\!]\eta$ any will have the same schema, no matter with of the four environments we use for η. Thus, the union of all four (partial) results is well-defined. The expression

factor r_1, r_2 **do** $@(1) \times @(2)$

denotes exactly this value, i.e., $\{[a_2, c_1, d_1], [a_2, c_2, d_2], [a_3, c_3, d_3]\}$. In fact, by this example, we have just given an *informal* semantics of the **factor** operator. In standard relational algebra, the relation just computed would have been expressed by $\pi_X(r_1 \bowtie r_2)$, where $X = (R_1 \setminus R_2) \cup (R_2 \setminus R_1)$. We will move on to an example using aggregation. We use the notation from [Klu82], $\mathrm{SUM}_A(r)$, where r is a relation which has an (integer) attribute A. This expression returns the sum of the integers in column A from r.

Consider the two relations, Jones and Miller, over the three attribute schema, {District, Buyer, Amount}, containing information about their sales, i.e,, which district, to whom, and the total value of the sale. In the following, we will simply use the initial letters: D, B, and A.

Now, for each of the city districts which are numbered from 5 through 20, we want to find the difference of their sales. This can be expressed by

factor Jones,Miller **on** D **do**
$(5 \leq D) \wedge (D \leq 20)?\{\#[R : \mathrm{SUM}_A(@(1)) - \mathrm{SUM}_A(@(2))]\}$

Notice that we have narrowed the intersection of the schemas down to D using the **on**-construct. A typical environment could look like

$\eta_{[b_4]}$:

$$\begin{array}{lcl} \# & \mapsto & [D\colon 7] \\ @(1) & \mapsto & \{[B\colon \text{Lee},\ A\colon 20], [B\colon \text{Wu},\ A\colon 12], [B\colon \text{Brown},\ A\colon 2]\} \\ @(2) & \mapsto & \{[B\colon \text{Clark},\ A\colon 25]\} \end{array}$$

For this environment, the boolean expression $(5 \leq D) \wedge (D \leq 20)$ evaluates to true. This means that we proceed to evaluate the expressions following the question mark. Had it evaluated to false, the result of the whole evaluation (for this particular environment) would be the empty relation (with schema $\{D, R\}$). In this case, we obtain a relation consisting of one tuple, which is the concatenation of $[D : 7]$ (the value of $\#$) and $[R : 9]$ (the value of $[R : \text{SUM}_A(@(1)) - \text{SUM}_A(@(2))]$). In total, if the involved relations are

Jones:

D	B	A
2	Smith	17
7	Lee	20
7	Wu	12
7	Brown	2
8	Chang	7

Miller:

D	B	A
7	Clark	25
8	Morrison	9
8	Kent	9
13	Hansen	12

we obtain

D	A
7	9
8	−11
13	−12

We can also combine standard relational operators with **factor** expressions. We can always do without them and only use **factor**, but if we need a project or a union, it seems more natural to use standard notation for these. Assume for instance that instead the information was listed in one relation r with

schema {SalesPerson, District, Buyer, Amount}, and Jones and Miller have occasionally been working together, in which case the information is listed for both of them, e.g., [{[Jones,11,Monroe,19], [Miller,11,Monroe,19]}] Now we could find the result of the *individual* work by Jones as (this time regardless of district):

$$\textbf{factor}\ \pi_{DBA}(\sigma_{S=\text{Jones}}(r)), \pi_{DBA}(\sigma_{S=\text{Miller}}(r))\ \textbf{on}\ D\ \textbf{do}\ \{\#[R : \text{SUM}_A(@(1) - @(2))]\}$$

Of course, the minus here is relation difference.

We move on to a *unary* application of **factor**. In the unary case, the intersection of the schemas of the arguments relations is simply the schema of the argument, as there is only one. This means that # will be instantiated with the tuples in the argument relation one by one, and that @(1) will be the empty relation in every environment. In other words, a unary **factor** expression is a *for all* operator, which runs through the tuples of the argument relation and perform operations on them individually.

Assume that r has the schema $\{A_1, A_2, A_3, A_4, B_1, B_2\}$. We want to select the tuples where $B_1 > B_2$, then find the sum of B_1 and B_2 and give it the new name B, and, finally, remove the attribute name B_2. Using **factor** we can write

$$\textbf{factor}\ r\ \textbf{do}\ B_1 > B_2?\ \#[B\colon B_1 + B_2] \backslash B_2$$

In a more standard formulation it would be (the extend operator is from [Gra84]):

$$\pi_{A_1,A_2,A_3,A_4,B_1,B}(extend_{B:=B_1+B_2}(\sigma_{B_1>B_2}(r)))$$

As another example of a unary applications which also demonstrates that expressions can be nested, consider division, usually defined as

$$r_1/r_2 = \{t \mid \{t\} \times r_2 \subseteq r_1\}$$

We can write

$$\textbf{factor}\ r_1, r_2\ \textbf{do}\ \textbf{factor}\ @(1)\ \textbf{do}\ \{\#\} \times r_2 \subseteq r_1?\ \{\#\}$$

where the first **factor** provides us with @(1)'s consisting of tuples from r_1 with schema $R_1 \setminus R_2$. The second **factor** runs through these tuples one by one and includes them in the result if they pass the test.

Notice a nice feature of this example which turns up very often in this languages it is not necessary to know the scheme of r_1 and r_2 to write the expression. In order to "implement" divide using standard relational operators, this *is* necessary; and the expression becomes quite large,

3 Preliminaries

Our starting point is standard relational algebra (as defined in [Mai83, Ull88], among others). A few variations are possible, in which case the theorems still holds but the proofs would be slightly changed.

We fix a domain $\mathcal{D}$ of all constants that can appear in relations and expressions and an infinite set $\mathcal{A}$ of attribute names For the results in this paper, there is no reason to distinguish between different types, so schemas are simply finite subsets of $\mathcal{A}$. Also, null values are not of relevance here, so we require that all relations be total [Mai83].

In order to avoid unreadable definitions and theorems later, we shall fix the letters we use as names for various entities:

c ranges over $\mathcal{D}$

$A, A_1, A_2, \ldots, B, B_1, B_2, \ldots$ range over $\mathcal{A}$

C ranges over both $\mathcal{D}$ and $\mathcal{A}$

X, Y, Z range over finite lists of symbols from $\mathcal{A}$

$r, r_1, r_2, \ldots, q, q_1, q_2, \ldots$ range over relation names and expressions

The notation $R(r)$ is used to indicate that relation r has schema R. As usual, we overload notation and let r refer to the relation $R(r)$ If r is a relational *expression*, then R is the schema of the relation the value of which the expression r denotes. A relation consists of a finite set of tuples over the schema of the relation. A tuple is a total function from the relation schema into $\mathcal{D}$. So, we can write e.g., $t(A)$ for t's value on the attribute name A.

Constant tuples are listed using brackets; the empty tuple, i,e., the tuple over the empty set, is written [ ]. We shall write, e.g., $\{A, B\}(\emptyset)$, to denote the empty relation with schema $\{A, B\}$. Also, if t is a tuple, $\{t\}$ denotes the obvious singleton relation.

We will use the standard symbols for the relational operators: $\cup, -, \bowtie, \sigma, \pi$ and δ for union, difference, natural join, select, project, and rename, respetively. Select conditions consist of a single equality or inequality between two attribute names or an attribute name and a constant. If $t = [A_1 : c_1, \ldots, A_k : c_k]$ and $\{A_1, \ldots, A_k\} \subseteq R$, then we will use $\sigma_t(r)$ as short for $\sigma_{A_k=c_k}(\cdots\sigma_{A_2=c_2}(\sigma_{A_1=c_1}(r))\cdots)$. This could equivalently be written $\{t\} \bowtie r$. We also need the special case of project, $\pi_\emptyset$. Recall that $\pi_\emptyset(r)$ is $\emptyset(\emptyset)$ if r is empty, and $\emptyset(\{[\,]\})$ otherwise, We will also use derived operators: $\times$ for Cartesian product, $\cap$ for intersections and $\underset{A=B}{\bowtie}$ for equijoin.

For tuples, we will use $t_{X \leftarrow Y}$ for the renaming equivalent to $\delta_{X \leftarrow Y}(\{t\})$, and $t \cdot t'$ for tuple concatenation, which is equivalent to $\{t\} \times \{t'\}$. Finally, we will use t_X for tuple projection equivalent to $\pi_X(\{t\})$.

4 The SFC Language

In order to keep the sizes of proofs reasonable, we first present a quite restricted version of FC, which we call SFC for *simple* FC. This language is just strong enough to *simulate* relational algebra. Aggregation and computation on domain values will be covered later.

First, we introduce a simple core language; primarily based on tuple operations. In BNF-notation, the language is defined as follows. The nonterminals `<atom>`, `<tup>`, and `<rel>` stands for atomic expression, tuple expression, and relational expression, respectively.

`<atom>`	`::=`	$c \mid A$
`<tup>`	`::=`	$[\,] \mid [A :$ `<atom>`$, \ldots] \mid$ `<tup>` $\backslash A \mid$ `<tup>` `<tup>` $\mid \#$
`<rel>`	`::=`	$\{\,\} \mid \{$`<tup>`$\} \mid$ `<rel>` $\times$ `<rel>` $\mid @(1) \mid @(2) \mid @(3) \mid \ldots$

The names of the tuple operations are: empty tuple, tuple formation, restriction, concatenation, and factorization tuple. The names of the relation

operations are: empty relation, relation formation, Cartesian product, and factorization relations.

The semantics of these operators are a usual and we shall not bore the reader with formal definitions of these details, which were also illustrated in previous examples. As an example, let t be the expression $[A:\ 3]([B:\ 7, C:\ 9] \setminus C)$. The *semantics* of t is denoted $[\![t]\!]$ and equals $[A:\ 3, B:\ 7]$. The symbol # is a tuple identifier, which is intended to be bound to a tuple value in the surrounding environment at the time when the expression is evaluated. Also, when this evaluation takes plate, the attribute names generated from `<atom>` have to belong to the domain of this tuple.

Furthermore, { } is the empty relation with the empty schema, i.e., $\emptyset(\emptyset)$, and if for example $t = [A:\ 3, B:\ 7]$, then $\{t\}$ is the relation with the one tuple t and with schema $\{A, B\}$, i.e., $\{A, B\}(\{[A:\ 3, B:\ 7]\})$. The relational operators $\times$, is the usual Cartesian product. Finally, the symbols @(1), @(2), @(3), are relation identifiers, which are intended to be bound to relation values in the surrounding environment at the time of evaluations.

Precedence: restriction binds strongest, then concatenation and Cartesian products. Operators associate to the left and parenthesis are used to resolve conflicts.

We now define the SFC language. We want to emphasize at this point that FC is a *free algebra* in the sense that the only requirement for the use of a language construct at a certain place is that the expected type at this place correspond with the type of the language construct. However, the simpler SFC language does not have this nice property.

In BNF-notation the SFC language looks like this:

`<cond>`	::=	`<atom>` = `<atom>` | `<atom>` $\neq$ `<atom>` |
		`<atom>` $>$ `<atom>` | `<atom>` $\leq$ `<atom>` | ...
		$@(m) = @(p) \mid @(m) \neq @(p)$
`<fac>`	::=	**factor** `<ra>`, ..., `<ra>` **do** `<rel>` |
		factor `<ra>`, ..., `<ra>` **do** `<cond>`? `<rel>`
`<ra>`	::=	r | `<fac>`

The `<ra>`'s are arguments to **factor**, which will be relation identifiers or **factor** expressions (derived from `<fac>`). There is the restriction that if

$@(m)$ appears in `<rel>`, then there should be at lest m arguments to the surrounding **factor** expression.

Conditionals are evaluated as ordinary boolean expression. When there are comparisons between $@(i)$'s, the two operands are required to have the same schema.

We have already discussed the semantics of expressions generated from `<rel>`. The intention of the *gate* operator, `<cond>`? `<rel>`, is that if the conditional evaluates to true, then the result is the result of evaluating the relational expression. Otherwise, the result is the empty relation (with the same schema as the relational expression). More formally,

$$[\![a?\ e]\!] = \begin{cases} [\![e]\!] & \text{if } [\![a]\!] \\ E(\emptyset), & \text{otherwise} \end{cases}$$

As in relational algebra, schemas can always be determined statically (on compile-time).

We shall use $f_1, f_2, \ldots$ to range over expressions which could be either purely standard relational or could contain **factor** expressions, and use $e, e', e'' \ldots$ to range over relation expressions (from the **do**-part of **factor**). The semantics of `<fac>` is given in the following. It is no more than a formalization of what we have already covered earlier through the use of examples. More intuition on the semantics can be obtained by studying definition 5.1. Some of the choices we have made in defining the semantics of `<fac>` will be motivated right after this definition.

If η is an environment and # an identifier, then $\eta(\#)$ is the "look-up" operation, i.e., it gives the value bound to #.

Definition 4.1 Let $\mathcal{F}$ = **factor** $r_1, \ldots, r_n$ **do** e. The *semantics* of $\mathcal{F}$ is denoted $[\![\mathcal{F}]\!]$ and is defined as

$$[\![\mathcal{F}]\!] = \bigcup_{t \in b_{\mathcal{F}}} [\![e]\!]\eta_t$$

where $X = \bigcap_i R_i,\ b_{\mathcal{F}} = \bigcup_i \pi_X(r_i)$, and for each $t \in b_{\mathcal{F}}$ and $i \in \{1, \ldots, n\}$

$$r_{\mathcal{F}}^{i,t} = \pi_{R_i \setminus X}(\sigma_t(r_i))$$

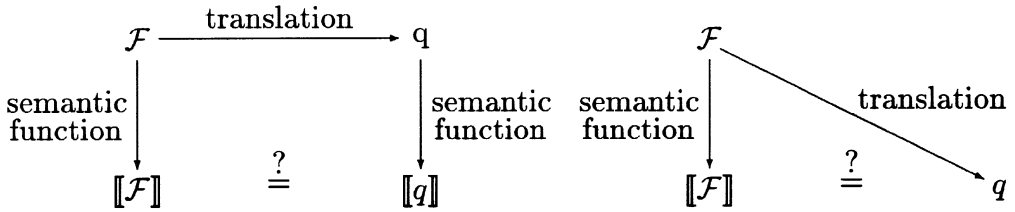
and, finally, for each $t \in b_{\mathcal{F}}$, we define the environment η_t by

$$\begin{array}{lrcl} \eta_t: & \# & \mapsto & t \\ & @(1) & \mapsto & r_{\mathcal{F}}^{1,t} \\ & . & . & . \\ & & & \\ & . & . & . \\ & @(n) & \mapsto & r_{\mathcal{F}}^{n,t} \end{array}$$

If $\mathcal{F} =$ **factor** $f_1, \ldots, f_n$ **do** e, i.e., the arguments themselves contain **factor** expressions, then we (recursively) find the semantics of $f_1, \ldots, f_n$. We then find the semantics of **factor** $r_1, \ldots, r_n$ **do** e, and for each i, we substitute $[\![f_i]\!]$ for r_i in the result. □

Remark 4.2 Note that # is always bound to a tuple value which has domain X. □

In this paper, we deal with translations between FC and relational algebra, and we are concerned with the correctness of these. The traditional way of proving such correctness results is to define the semantics of both languages in terms of a common semantic domain. Consider the following illustration (the diagram to the left), where $\mathcal{F}$ is an FC expression, which, we assume, translates to the relational algebra expression q.



Here, $[\![\cdot]\!]$ denotes an entity's semantics. The translation is correct if $[\![\mathcal{F}]\!] = [\![q]\!]$ for all $\mathcal{F}$ (and their translations q). Traditionally, the common semantic domain one would choose here would be set theory; as that is the only "reasonable" domain which is simpler than both of the involved domains (languages). This choice would mean, however, that we would have to spend a lot of time giving semantics to relational algebra and dealing with almost trivial relational algebra and set theoretic manipulations. As relational algebra operations are in fact what we need to define the semantics of FC we would have to reinvent these (with new names) on sets (which is almost the same as relational algebra anyway).

As the reader might have guessed at this point, we will do this in a different way. Our opinion is that relational algebra is so well understood by now that it is reasonable to use it as a semantic domain. This implies that we will not have to give a semantics for relational algebra (the diagram to the right); the semantics of a relational algebra expression is simply the value it evaluates to.

These decisions also imply that when we translate $\mathcal{F}$ to q, the latter should be considered pure syntax, whereas when we ask, e.g., $[\![\mathcal{F}]\!] = q?$, we mean 'do these two relational algebra expressions *evaluate* to the same *relation*' ?

5 From Relational Algebra to SFC

In this section, we give the translation from relational algebra to SFC. We use a standard version of relational algebra which is complete in the sense of [Cod72]. The translations given here were first stated in [LSS92] in almost the same form.

Definition 5.1 The standard relational operators can be translated into SFC as follows:

union	$r_1 \cup r_2$	translates into **factor** r_1, r_2 **do** $\{\#\}$
difference	$r_1 - r_2$	translates into **factor** r_1, r_2 **do** $@(2) = \{\}?\ \{\#\}$
join	$r_1 \bowtie r_2$	translates into **factor** r_1, r_2 **do** $@(1) \times \{\#\} \times @(2)$
project	$\pi_Z(r)$	translates into **factor** $r, Z(\emptyset)$ **do** $\{\#\}$
select	$\sigma_{A\theta C}(r)$	translates into **factor** r **do** $A\theta C?\ \{\#\}$
rename	$\delta_{A \leftarrow B}(r)$	translates into **factor** r **do** $\# \setminus A[B :\ A]$

C is either an attribute name from R or a constant. □

Even in this restricted version, we only need very few of the possible expressions in FC to define the standard relational operators. The proof of correctness of the translation is easy.

Lemma 5.2 The translations defined in definition 5.1 are semantics preserving.

Proof We use the relation names from definition 5.1 and let $\mathcal{F}$ denote the translation defined there. Notation from definition 4.1 is used.

union — As $R_1 = R_2$, we have that $X = R_1 \cap R_2 = R_1 (= R_2)$, so $b_{\mathcal{F}} = r_1 \cup r_2$. Now

$$[\![\mathcal{F}]\!] = \bigcup_{t \in r_1 \cup r_2} [\![\{\#\}]\!]\eta_t = \bigcup_{t \in t_1 \cup r_2} \{t\} = r_1 \cup r_2.$$

difference — Again, $R_1 = R_2$ and $b_{\mathcal{F}} = r_1 \cup r_2$. For each $t \in b_{\mathcal{F}}$, @(2) is bound to

$$r_{\mathcal{F}}^{2,t} = \pi_{R_2 \setminus X}(\sigma_t(r_2)) = \pi_{\emptyset}(\sigma_t(r_2)) = \begin{cases} \{[\,]\}, & \text{if } t \in r_2 \\ \{\,\}, & \text{if } t \notin r_2 \end{cases}.$$

So, for a given environment η_t, the equality @(2) = { } holds exactly when $t \notin r_2$. Now,

$$\begin{aligned} [\![\mathcal{F}]\!] &= \bigcup_{t \in r_1 \cup r_2} [\![@(2) = \{\,\}?\ \{\#\}]\!]\eta_t \\ &= \bigcup_{\substack{t \in r_1 \cup r_2 \\ t \notin r_2}} [\![\{\#\}]\!]\eta_t \\ &= \bigcup_{t \in r_1 - r_2} \{t\} \\ &= r_1 - r_2 \end{aligned}$$

join — X is $R_1 \cap R_2$ and $b_{\mathcal{F}} = \pi_X(r_1) \cup \pi_X(r_2)$.

$$\begin{aligned} [\![\mathcal{F}]\!] &= \bigcup_{t \in b_{\mathcal{F}}} [\![@(1) \times \{\#\} \times @(2)]\!]\eta_t \\ &= \bigcup_{t \in \pi_X(r_1) \cup \pi_X(r_2)} r_{\mathcal{F}}^{1,t} \times r_{\mathcal{F}}^{2,t} \\ &= \bigcup_{t \in \pi_X(r_1) \cup \pi_X(r_2)} \pi_{R_1 \setminus X}(\sigma_t(r_1) \times \{t\} \times \pi_{R_2 \setminus X}(\sigma_t(r_2)) \\ &= \bigcup_{t \in \pi_X(r_1) \cup \pi_X(r_2)} \sigma_t(r_1) \bowtie \sigma_t(r_2) \\ &= r_1 \bowtie r_2 \end{aligned}$$

project — $X = R \cap Z = Z$ and $b_{\mathcal{F}} = \pi_X(r) \cup \pi_X(Z(\emptyset)) = \pi_Z(r)$. So,

$$[\![\mathcal{F}]\!] = \bigcup_{t \in b_{\mathcal{F}}} [\![\{\#\}]\!]\eta_t = \bigcup_{t \in \pi_Z(r)} \{t\} = \pi_Z(r).$$

select — $b_{\mathcal{F}} = \pi_R(r) = r$, so

$$\begin{aligned} [\![\mathcal{F}]\!] &= \bigcup_{t \in r} [\![A\theta C?\{\#\}]\!]\eta_t \\ &= \bigcup_{\substack{t \in r \\ [A\theta C]\eta_t}} [\![\{\#\}]\!]\eta_t \\ &= \bigcup_{\substack{t \in r \\ t(A)\theta t(C)}} \{t\} \\ &= \sigma_{A\theta C}(r) \end{aligned}$$

If C is a constant, $t(C)$ is replaced by C.

rename — $b_{\mathcal{F}} = \pi_R(r) = r$, so

$$[\![\mathcal{F}]\!] = \bigcup_{t \in r} [\![\# \setminus A[B :\ A]]\!]\eta_t = \bigcup_{t \in r} \{t_{A \leftarrow B}\} = \delta_{A \leftarrow B}(r).$$

□

We now observe that because of the denotational nature of relational algebra, a sequence of semantic preserving translations is again semantic preserving, and we obtain:

Theorem 5.3 A relational algebra expression q can be translated into an SFC expression $\mathcal{F}$ such that $[\![F]\!] = q$. □

Example 5.4 Compute the symmetrical difference of r_1 and r_2:

$$(r_1 - r_2) \cup (r_2 - r_1)$$

This would be translated into

factor
  factor r_1, r_2 **do** @(2) = { }? {#},
  factor r_2, r_1 **do** @(1) = { }? {#}
do {#}

Notice that this is not the way one would naturally write this query in FC (or SFC). Rather, one would write

factor r_1, r_2 **do** @(1) ≠ @(2)? {#}

However, here we are merely concerned with the *existence* of a translation.

□

6 From SFC to Relational Algebra

We will define a step by step translation. First, we will eliminate gates. In the process, we allow relational algebra expressions to appear as arguments to **factor**. As the semantics of **factor** is purely denotational, the semantics of this is, of course, exactly the same as the semantics of **factor** expressions with relation identifiers or other **factor** expressions as arguments.

Consider an expression **factor** $r_1, \ldots, r_n$ **do** $b?$ e. We want to translate this into an expression of the form **factor** $r'_1, \ldots, r'_n$ **do** e such that the two are semantically equivalent. We define relational algebra expressions which for each boolean expression give the tuples of the relations that make the boolean expressions true.

Definition 6.1 Given relation expressions $r_1, \ldots, r_n$. Let $X = \bigcap_i R_i$ and let b be a boolean expression (derived from `<cond>`). We define $lim(b)$ to be the *limiting relational algebra expression* for b w.r.t. $r_1, \ldots, r_n$. Let x be a unique variable name, and let θ be $=$, $\neq$, $>$, $<$, $\geq$, or $\leq$, so that $A\theta C$ is one of the usual select conditions.

b	$lim(b)$
$A\theta C$	$\sigma_{A\theta C}(x)$
$@(m) \neq @(p)$	$x \cap \pi_X((r_m - r_p) \cup (r_p - r_m))$
$@(m) = @(p)$	$x - \pi_X((r_m - r_p) \cup (r_p - r_m))$

□

The rôle of x at the moment is to act as place holder. Later, it will be replaced by one of the (relational) arguments to **factor**.

Definition 6.2 $e[r/x]$ denotes the term which is constructed by syntactically replacing each occurrence of x in e with r. □

Example 6.3 Continuing example 5.4. we look at

$$\mathcal{F} = \textbf{factor } r_1, r_2 \textbf{ do } @(1) \neq @(2)?\{\#\}$$

With b equal to $@(1) \neq @(2)$, $lim(b)$ is $x \cap \pi_X((r_1 - r_2) \cup (r_2 - r_1))$.

In lemma 6.4, we will prove a general result which, when applied to this example, means that $\mathcal{F}$ and the following expression are semantically identical.

factor
$r_1 \bowtie (\pi_X(r_1) \cap \pi_X((r_1 - r_2) \cup (r_2 - r_1)))$,
$r_2 \bowtie (\pi_X(r_2) \cap \pi_X((r_1 - r_2) \cup (r_2 - r_1)))$
do $\{\#\}$

Notice that x has been replaced by $\pi_X(r1)$ and $\pi_X(r2)$.

Now, $X = R_1 \cap R_2$, so if we assume that $R_1 = R_2$, then this can be simplified to

$$\textbf{factor } r_1 - r_2, r_2 - r_1 \textbf{ do } \{\#\}$$

Here the symmetrical difference operator is more apparent. Notice, however, that this simplification is only possible if we assume that $R_1 = R_2$. The original $\mathcal{F}$ is well-defined also when $R_1 \neq R_2$. □

We prove that definition 6.1 works as intended, i.e., that a boolean expression b is true for an environment induced by a certain tuple exactly when for some argument, r_i, this tuple belongs to $lim(b)[\pi_X(r_i)/x]$.

Lemma 6.4 Let $\mathcal{F}, X, b_\mathcal{F}, t$, and η_t be as in definition 4.1. Then

$$[\![b]\!]\eta_t \Longleftrightarrow \exists i : \ t \in lim(b)[\pi_X(r_i)/x]$$

Proof We perform a case analysis on b.

$\underline{b \text{ is } A\theta C}$

$$\begin{array}{l} [\![b]\!]\eta_t \\ \Updownarrow \\ [\![A\ \theta\ C]\!]\eta_t \\ \Updownarrow \\ t(A)\ \theta\ t(C) \\ \Updownarrow \\ t \in \sigma_{A\theta C}(b_\mathcal{F}), \text{ as } t \in b_\mathcal{F} \\ \Updownarrow \\ \exists i : \ t \in \sigma_{A\theta C}(\pi_X(r_i))), \text{ by definition of } b_\mathcal{F} \\ \Updownarrow \\ \exists i : \ t \in lim(b)[\pi_X(r_i)/x] \end{array}$$

$\underline{b \text{ is } @(m) \neq @(p)}$ Recall the requirement that $@(m)$ and $@(p)$ have identical schemas. This means that $R_m \setminus X = R_p \setminus X$ which, because of the definition of X, implies that $R_m = R_p$.

Observe first that

$$\pi_{R_m \setminus X}(\sigma_t(r_m)) - \pi_{R_p \setminus X}(\sigma_t(r_p)) \neq \emptyset \Leftrightarrow t \in \pi_X(r_m - r_p)$$

Now,

$$\begin{array}{l} [\![b]\!]\eta_t \\ \Updownarrow \\ [\![@(m)]\!]\eta_t \neq [\![@(p)]\!]\eta_t \\ \Updownarrow \\ r_{\mathcal{F}}^{m,t} \neq r_{\mathcal{F}}^{p,t}, \text{ by definition 4.1} \\ \Updownarrow \\ \pi_{R_m \setminus X}(\sigma_t(r_m)) - \pi_{R_p \setminus X}(\sigma_t(r_p)) \neq \emptyset \vee \\ \pi_{R_p \setminus X}(\sigma_t(r_p)) - \pi_{R_m \setminus X}(\sigma_t(r_m)) \neq \emptyset \\ \Updownarrow \\ t \in \pi_X(r_m - r_p) \vee t \in \pi_X(r_p - r_m), \text{ obs. above} \\ \Updownarrow \\ t \in \pi_X((r_m - r_p) \cup (r_p - r_m)) \\ \Updownarrow \\ \exists i : \ t \in \pi_X(r_i) \wedge t \in \pi_X((r_m - r_p) \cup (r_p - r_m)) \\ \Updownarrow \\ \exists i : \ t \in lim(b)[\pi_X(r_i)/x] \end{array}$$

$\underline{b \text{ is } @(m) = @(p)}$ By negating the implications from the previous case, we immediately get

$$[\![b]\!]\eta_t \Longleftrightarrow t \notin \pi_X((r_m - r_p) \cup (r_p - r_m))$$

As $t \in \bigcup_i \pi_X(r_i)$, we obtain that

$$\begin{array}{l} [\![b]\!]\eta_t \\ \Updownarrow \\ \exists i : \ t \in \pi_X(r_i) \wedge t \notin \pi_X((r_m - r_p) \cup (r_p - r_m)) \\ \Updownarrow \end{array}$$

$$
\begin{array}{l}
\quad \exists i: \ t \in \pi_X(r_i) - \pi_X((r_m - r_p) \cup (r_p - r_m)) \\
\Updownarrow \\
\quad \exists i: \ t \in lin(b)[\pi_X(r_i)/x]
\end{array}
$$

□

Some properties of limiting relational algebra expressions are needed later:

Proposition 6.5 Let b, X, and $r_1, \ldots, r_n$, be as in definition 6.1. Then the following hold:

1) $\forall i$: the schema of $lim(b)[\pi_X(r_i)/x]$ is X.

2) $\forall i$: $lim(b)[\pi_X(r_i)/x] \subseteq \pi_X(r_i)$

Proof Easy observation. □

We can now prove that a gate operator can be removed by applying relational algebra operations ($lim(b)$'s) to the arguments instead.

Lemma 6.6 Let

$$\mathcal{F} = \textbf{factor } r_1, \ldots, r_n \textbf{ do } b? \ e \quad \text{and} \quad \mathcal{F}' = \textbf{factor } r'_1, \ldots, r'_n \textbf{ do } e$$

where $r'_i = r_i \bowtie lim(b = [\pi_X(r_i)/x]$. Then $[\![\mathcal{F}]\!] = [\![\mathcal{F}']\!]$

Proof First,

$$
\begin{array}{l}
\quad t \in b_{\mathcal{F}} \wedge [\![b]\!]\eta_t \\
\Updownarrow \\
\quad t \in b_{\mathcal{F}} \wedge \exists i: \ t \in lim(b)[\pi_X(r_i)/x] \text{ by lemma 6.4} \\
\Updownarrow \\
\quad t \in \bigcup_i lim(b)[\pi_X(r_i/x], \text{ by 1) in proposition 6.5} \\
\Updownarrow \\
\quad t \in \bigcup_i \pi_X(r_i \bowtie lim(b)[\pi_X(r_i)/x]), \text{ proposition 6.5 and } X \subseteq R_i \\
\Updownarrow \\
\quad t \in \bigcup_i \pi_X(r'_i), \text{ by definition of } r'_i \\
\Updownarrow \\
\quad t \in b_{\mathcal{F}'}, \text{ by definition of } b_{\mathcal{F}'}
\end{array}
$$

So,

$$\begin{array}{rcl}
[\![\mathcal{F}]\!] & = & \bigcup_{t \in b_{\mathcal{F}}} [\![b?\ e]\!]\eta_t \\
& = & \bigcup_{t \in b_{\mathcal{F}}} [\![e]\!]\eta_t, \text{ by definition of the gate operator} \\
& & \quad [\![b]\!]\eta_t \\
& = & \bigcup_{t \in b_{\mathcal{F}'}} [\![e]\!]\eta_t, \text{ as we have just proved} \\
& = & [\![\mathcal{F}]\!]
\end{array}$$

□

Having dealt with gates, we turn our attention to the relation expression, i.e., the **do**-part. First, we simplify matters by putting tuple expressions in normal form.

Tuple expressions can be put in normal form, i.e., they can be translated to a normal form, which is semantically equivalent to the original expression. The translation exploits the fact that no new values can be introduced. The only values that can be put into a tuple come from the value bound to the tuple identifier # or from constants appearing in the tuple expression.

Definition 6.7 The translation of a tuple expression t into a tuple expression normal form, denoted $\mathrm{TT}_X(t)$, is defined inductively in the structure of t. The translation is performed relative to a set X of attribute names. Assume that $X = \{A_1, \ldots, A_p\}$.

t	$\mathrm{TT}_X(t)$
c	c
A	$\mathrm{TT}_X(\#)(A)$
$[\,]$	$[\,]$
$[A:\ d]$	$[A:\ \mathrm{TT}_X(d)]$
$t' \setminus A$	$\mathrm{TT}_X(t') \setminus A$
$t't''$	$\mathrm{TT}_X(t')\mathrm{TT}_X(t'')$
$\#$	$[A_1:\ A_1, \ldots, A_p:\ A_p]$

□

The constructs on the left-hand side are treated as purely syntactical structures, whereas the tuple operations on the right-hand side should be carried out (on the syntactical structures). The intention is to use this translation as a "tuple expression analyzer".

Example 6.8 Assume that $X = \{A_1, A_2, A_3\}$. Then

$$\mathrm{TT}_X(\#\backslash A_1 \backslash A_2[A_2 :\ 5][A_4 :\ A_3]) = [A_2 :\ 5,\ A_3 :\ A_3,\ A_4 :\ A_3]$$

This tuple expression shows as directly as possible what effect the tuple expression has, provided that $\eta(\#)$ is bound to a tuple value with domain X. □

The following two statements express formally that $\mathrm{TT}_X(t)$ is a normal form of t.

Proposition 6.9 If t is a tuple expression and $dom(\eta(\#)) = X$, then

- $[\![t]\!]\eta = [\![\mathrm{TT}_X(t)]\!]\eta$.
- $\mathrm{TT}_X(t)$ is of the form $[A_1 :\ d_1, \ldots, A_k :\ d_k]$, where each d_i is either a constant appearing syntactically in t or an attribute name from X.

Proof Trivial. □

We now move on to the core of the matter of actually replacing a **factor** expression by standard relational operators. This is also technically the most difficult part of the proof.

First, we need the ability to take an extra "copy" of a set of attribute names.

Definition 6.10 If $X = A_1, \ldots, A_p$ is a set of attribute names and $\mathcal{F}$ is an SFC expression, then Y is a *unique copy* of X w.r.t. $\mathcal{F}$ if the following holds: for each A_j there is an A'_j such that $Y = A'_1, \ldots, A'_p$ and $|X| = |Y|$, $X \cap Y = \emptyset$, and no attribute name from Y appears in $\mathcal{F}$ (syntactically). □

We are now ready to define the translation of SFC expressions into relational algebra. Basically, we have to compute this union of core language expressions evaluated in different environments, $\bigcup_{t \in b_{\mathcal{F}}} [\![e]\!]\eta_t$. We have to calculate all these expressions, $[\![e]\!]\eta_t$, at the same time. This can be done by keeping all (partial) results in one relation. The problem is to keep these separated

until all tuple operations etc. are carried out. We use an extra copy of X to ensure this.

Definition 6.11 Let $\mathcal{F} = \textbf{factor } r_1, \ldots, r_n \textbf{ do } e$, where we allow only tuple subexpressions in e of the form described in proposition 6.9. We assume further that the gate operator does not appear in e. The *translation* of $\mathcal{F}$ into a standard relational expression is defined recursively in the structure of e as follows.

First, let $X = \bigcap_i R_i$ and let Y be a unique copy of X w.r.t. $\mathcal{F}$. Let $b_{\mathcal{F}} = \bigcup_i \pi_X(r_i)$. Now, we define the translation, q, of e:

$\underline{e}$	$\underline{q}$
$\{\ \}$	$\{Y\}(\emptyset)$
$\{t\}$	By assumption, $t = [B_1 :\ d_1, \ldots, B_k :\ d_k]$, for some attribute names $B_1, \ldots, B_k$ such that the d_j's are either constants or attribute names from X (remark 4.2).
	$\underline{t = [\ ]}$ $\quad \delta_{X \leftarrow Y}(b_{\mathcal{F}})$
	$\underline{d_1 \text{ is } c}$ $\quad$ Inductively translate $[B_2 :\ d_2, \ldots, B_k :\ d_k]$ to q'. The translation is then given by $\{[B_1 :\ c]\} \times q'$.
	$\underline{d_1 \text{ is } A_j}$ $\quad$ Inductively translate $[B_2 :\ d_2, \ldots, B_k :\ d_k]$ to q'. The translation is then given by $\delta_{A_j \leftarrow B_1}(\pi_{A_j}(b_{\mathcal{F}})) \underset{B_1 = A'_j}{\bowtie} q'$
$@(m)$	$\delta_{X \leftarrow Y}(r_m)$
$e' \times e''$	If e' and e'' translate to q' and q'', respectively, then $e' \times e''$ translates to $q' \bowtie q''$.

□

We observe the following.

Proposition 6.12 In definition 6.11, the following holds:

1) the schema of the q's are always the schema of e plus Y, i.e., $Q = EY$.

2) $\delta_{Y \leftarrow X}(\pi_Y(q)) \subseteq b_{\mathcal{F}}$

Proof Easy induction. □

Before we move on, we illustrate the last definitions by an example. In order to keep things a reasonable size, we give a rather simple example. This means that only some aspects of the technique will be illustrated.

Example 6.13 Consider **factor** r **do** $\{\# \setminus A_2\ [B :\ A_2]\}$, which is the **factor** expression for a simple renaming, $\delta_{A \leftarrow B}(r)$. Assume that $R = A_1A_2$. First of all, definition 6.11 can only be used when tuple expressions are in normal form, so we consider instead

$$\textbf{factor}\ r\ \textbf{do}\ \{[A_1 :\ A_1, B :\ A_2]\}$$

This rewriting is what definition 6.7 would do for us.
Let $A'_1A'_2$ be a unique copy of A_1A_2. Now, the empty tuple [ ] is translated into $\delta_{A_1A_2 \leftarrow A'_1A'_2}(b_{\mathcal{F}}$, so $[B :\ A_2]$ is translated into

$$\delta_{A_2 \leftarrow B}(\pi_{A_2}(b_{\mathcal{F}})) \underset{B=A'_1}{\bowtie} \delta_{A_1A_2 \leftarrow A'_1A'_2}(b_{\mathcal{F}})$$

and then $[A_1 :\ A_1, B :\ A_2]$ is translated into

$$\delta_{A_1 \leftarrow A_1}(\pi_{A_1}(b_{\mathcal{F}})) \underset{A_1=A'_1}{\bowtie} (\delta_{A_2 \leftarrow B}(\pi_{A'_2}(b_{\mathcal{F}}) \underset{B=A'_1}{\bowtie} \delta_{A_1A_2 \leftarrow A'_1A'_2}(b_{\mathcal{F}}))$$

As there is only one argument to **factor**, $b_{\mathcal{F}}$ is simply r , so this expression can be simplified to

$$\pi_{A_1}(r) \underset{A_1=A'_1}{\bowtie} \delta_{A_1 \leftarrow B}(\pi_{A_1}(r) \underset{B=A'_1}{\bowtie} \delta_{A_1A_2 \leftarrow A'_1A'_2}(r)$$

So, if r was as shown below (to the left)

r:

A_1	B_1
2	3
9	7
2	9

translation:

A'_1	A'_2	A_1	B
2	3	2	3
9	7	9	7
2	9	2	9

we would calculate the relation shown to the right. The final result can be found, as we will prove later, by then removing the unique copy of X, i.e. $A'_1A'_2$. □

Notice that if the unique copy of X we not used, then tuples, which should be different, would collapse in part results, and, in addition, we would not have a means of combining results as we do now using the equijoin

Lemma 6.14 Let $\mathcal{F}, X, Y$, and $b_{\mathcal{F}}$ be as in definition 6.11, and let t and η_t be as in definition 4.1. If e translates to q, then for any t',

$$t' \in [\![e]\!]\eta_t \Longleftrightarrow t_{X \leftarrow Y} \cdot t' \in q$$

Proof By induction in the structure of e_i.

$\underline{\{\ \}}$ $[\![\{\ \}]\!]\eta = \emptyset(\emptyset)$ and $q = Y(\emptyset)$. As no tuples can belong to an empty relation (independent of the schema), there is nothing to show.

$\underline{\{t''\}}$ By assumption (as in definition 6.11), we have that t'' is of the form $[B_1 :\ d_1, \ldots, B_k :\ d_k]$. Let q' be the translation of $t''_{tail} = [B_2 : d_2, \ldots, B_k :\ d_k]$. We proceed by induction in the length of t''.

$\underline{t'' = [\,]}$ $[\![\{[\,]\}]\!]\eta_t = \emptyset(\{[\,]\})$, while $q = \delta_{Y \leftarrow X}(b_{\mathcal{F}})$. The only tuple that belongs to $\emptyset(\{[\,]\})$ is $[\,]$. From remark 4.2 and proposition 6.12, t' on the right-hand side is also forced to be $[\,]$. It remains to be argued that $t_{X \leftarrow Y} \in q$. But that follows from $t \in b_{\mathcal{F}}$ and the definition of q.

$\underline{d_1 \text{ is } c}$ The induction hypothesis states that for any t'_{tail},

$$t'_{tail} \in [\![\{t''_{tail}\}]\!]\eta_t \Leftrightarrow t_{X \leftarrow Y} \cdot t'_{tail} \in q'$$

Now,

$$\begin{array}{l} t'' \in [\![\{t''\}]\!]\eta_t \\ \Updownarrow \\ t'_{tail} \in [\![\{t''_{tail}\}]\!]\eta_t, \text{ where } t' = [B_1 :\ c] \cdot t'_{tail} \\ \Updownarrow \\ t'_{X \leftarrow Y} \cdot t'_{tail} \in q \text{ by the induction assumption} \\ \Updownarrow \\ t_{X \leftarrow Y} \cdot [B_1 :\ c] \cdot t'_{tail} \in \{[B_1 :\ c]\} \times q' \\ \Updownarrow \\ t_{X \leftarrow Y} \cdot t' \in q \end{array}$$

$\underline{d_1 \text{ is } A_j}$ As above, except that we have to add the assumption $\eta_t(\#)$. A_j evaluates to c, and then observe that

$$t_{X \leftarrow Y} \cdot t' \in [B_1 : \ c] \times q'$$

$$\Updownarrow$$

$$t_{X \leftarrow Y} \cdot t' \in \delta_{A_j \leftarrow B_1}(\pi_{A_j}(b_{\mathcal{F}})) \underset{B_1 = A'_j}{\bowtie} q'$$

$\underline{@(m)}$ $[\![@(m)]\!]\eta_t = \eta_t(@(m) = r_{\mathcal{F}}^{m,t} = \pi_{R_m \setminus X}(\sigma_t(r_m)$ and $q = \delta_{X \leftarrow Y}(r_m)$. Now,

$$t' \in \pi_{R_m \setminus X}(\sigma_t(r_m))$$

$$\Updownarrow$$

$$\exists t'' \in \sigma_t(r_m) : \ t' = t''_{R_m \setminus X}$$

But then t'' can be written $t' \cdot t''_X$, when t''_X is the remaining part of t'', and so $t'' = t' \cdot t''_X = t' \cdot t$, as $dom(t) = \{X\}$ and $t'' \in \sigma_t(r_m)$. Finally,

$$t' \cdot t \in \sigma_t(r_m) \subseteq r_m$$

$$\Updownarrow$$

$$t' = t''_{X \leftarrow Y} \in \delta_{X \leftarrow Y}(r_m)$$

$\underline{e' \times e''}$ $[\![e' \times e'']\!]\eta_t = [\![e']\!]\eta_t \times [\![e'']\!]\eta_t$ and $q = q' \bowtie q''$, where q' and q'' are the translations of e' and e'', respectively. Remember that $E' \cap E'' = \emptyset$. Let $t' \in [\![e']\!]\eta_t \times [\![e'']\!]\eta_t$. Write t' as $t_1 \cdot t_2$, where $dom(t_1) = E'$ and $dom(t_2) = E''$. Then

$$t' \in [\![e']\!]\eta_t \times [\![e'']\!]\eta_t$$

$$\Updownarrow$$

$$t_1 \in [\![e']\!]\eta_t \wedge t_2 \in [\![e'']\!]\eta_t$$

$$\Updownarrow$$

$$t_{X \leftarrow Y} \cdot t_1 \in q' \wedge t_{X \leftarrow Y} \cdot t_2 \in q'', \text{ by induction}$$

$$\Updownarrow$$

$$t_{X \leftarrow Y} \cdot t_1 \cdot t_2 \in q' \bowtie q''$$

$$\Updownarrow$$

$$t_{X \leftarrow Y} \cdot q' \bowtie q''$$

□

Instead of expressing this connection at the tuple level, we can express it in relational algebra.

Corollary 6.15 Let $\mathcal{F}, X, Y$, and $b_{\mathcal{F}}$ be as in definition 6.11, and let t and

η_t be as in definition 4.1. If e translates to q, then

$$[\![e]\!]\eta_t = \pi_{Q\setminus Y}(\sigma_{t_{X\leftarrow Y}}(q))$$

Proof Directly from lemma 6.14. □

Finally, we can handle **factor** expressions without gate-operators and with tuple expressions in normal form.

Lemma 6.16 Let $\mathcal{F}, X, Y$, and $b_{\mathcal{F}}$ be as in definition 6.11, and let q be the translation of $\mathcal{F}$. Then

$$[\![\mathcal{F}]\!] = \pi_{Q\setminus Y}(q)$$

Proof

$$\begin{array}{rcl} [\![\mathcal{F}]\!] & = & \bigcup_{t\in b_{\mathcal{F}}} [\![e]\!]\eta_t \\ & = & \bigcup_{t\in b_{\mathcal{F}}} \pi_{Q\setminus Y}(\sigma_{t_{X\leftarrow Y}}(q)), \text{ by corollary 6.15} \\ & = & \pi_{Q\setminus Y}(\bigcup_{t\in b_{\mathcal{F}}} \sigma_{t_{X\leftarrow Y}}(q)) \\ & = & \pi_{Q\setminus Y}(q), \text{ by 2) in proposition 6.12} \end{array}$$

□

Combining the results in this section we obtain the following.

Theorem 6.17 An SFC expression $\mathcal{F}$ can be translated into a relational algebra expression r such that $[\![\mathcal{F}]\!] = r$.

Proof If $\mathcal{F}$ contains **factor** expressions as arguments, we recursively translate these to relational algebra. This can be done as the semantics of **factor** is purely denotational.

We treat the more difficult case where $\mathcal{F}$ contains a gate operator, so $\mathcal{F}$ is of the form **factor** $r_1, \ldots, r_n$ **do** $b?$ e. Construct $lim(b)$ using definition 6.1. Using lemma 6.6, we can find relational algebra expressions $r'_1, \ldots, r'_n$ such that if **factor** $r'_1, \ldots, r'_n$ **do** e then $[\![\mathcal{F}]\!] = [\![\mathcal{F}']\!]$.

Let $\mathcal{F}''$ be like $\mathcal{F}'$ except that e is changed to e' by putting all tuple expressions in normal form using definition 6.7, i.e., let $X = \bigcap_i R'_i$ and substitute each t with $\mathrm{TT}_X(t)$. Now, by proposition 6.9 and the fact that the tuple language is purely denotational, $[\![\mathcal{F}'']\!] = [\![\mathcal{F}']\!]$, where $\mathcal{F}'' =$ **factor** $r'_1, \ldots, r'_n$ **do** e.

From $\mathcal{F}''$, using definition 6.10, construct the relational algebra expression q. Let Y be a unique copy of X w.r.t. $\mathcal{F}''$. Then by lemma 6.16, $[\![\mathcal{F}'']\!] = \pi_{Q\setminus Y}(q)$.

Let $r = \pi_{Q \setminus Y}(q)$. Now $[\![\mathcal{F}]\!] = [\![\mathcal{F}']\!] = [\![\mathcal{F}'']\!] = \pi_{Q \setminus Y}(q) = r$. □

7 The Full Language

This section requires that the reader have a full understanding of the most technical part of the paper, i.e,, section 6. We give, without proofs, the constructs and additions to the definitions in that section which are necessary in order to show that even the full language can be translated to relational algebra. When we come to aggregation and computation on domain values, relational algebra has to be extended appropriately with a **group_by** operator or similar constructs.

In connection with each construct, we will either list an example to show what type of queries we have in mind, or we shall simply "define" the construct by an example.

7.1 Extending Boolean Expressions

Example 7.1 factor r_1, r_2 **do** $(@(1) \subseteq @(2)) \wedge (A = B)?$ e □

In order to use the boolean operators and comparisons of relations, e.g., contained-in, we need to extend definition 6.1. Assume that b_1 and b_2 give limiting relational algebra expressions $lim(b_1)$ and $lim(b_2)$, respectively.

b	$lim(b)$
$b_1 \wedge b_2$	$lim(b_1) \cap lim(b_2)$
$b_1 \vee b_2$	$lim(b_1) \cup lim(b_2)$
$\neg b_1$	$x - lim(b_1)$
$@(m) \subseteq @(p)$	$x - \pi_X(r_m - r_p)$
$@(m) \supseteq @(p)$	$x - \pi_X(r_p - r_m)$
$@(m) \subset @(p)$	$(x - \pi_X(r_m - r_p)) \cap \pi_X(r_p - r_m)$
$@(m) \supset @(p)$	$(x - \pi_X(r_p - r_m)) \cap \pi_X(r_m - r_p)$

If a constant relation r appears in a comparison in the plate of $@(p)$, say,

then replace r_p in $lim(b)$ with $x \times r$.

7.2 Relational Operators in Relation Expressions

Example 7.2 factor r_1, r_2 **do** $\delta_{B \leftarrow A}(@(1)) \bowtie @(2)$ □

The following operators translate directly: $\cup, -, \bowtie, \sigma, \delta$. Let q, q', and q'' be the translation of e, e', and e'', respectively, and let Y be a unique copy of X, Then $\pi_Z(e)$ translates to $\pi_{YZ}(q)$, $e' \times e''$ translates to $q' \bowtie q''$, and a constant relation r translates to $r \times \delta_{X \leftarrow Y}(b_{\mathcal{F}})$. Derived operators are first translated to standard relational algebra (we have given $\times$ as it is used directly in FC)

7.3 Relational Operators in Boolean Expressions

Example 7.3 factor r_1, r_2 **do** $(@(1) \times \{[D: \ 7]\}) \subseteq (@(2) \bowtie r)?\ e$ □

Expressions are translated exactly as in the actual expression, i.e., there is an extra unique copy of X on everything (Y). We use standard templates for the different comparisons.

Example 7.4 Let e_1 and e_2 be relation expression. Let q_1 and q_2 be the translation of e_1 and e_2, respectively. The template for $\subseteq$ would be $x - \delta_{Y \leftarrow X}(\pi_Y(x_1 - x_2))$, so the limiting relational algebra expression for $e_1 \subseteq e_2$ is $x - \delta_{Y \leftarrow X}(\pi_Y(q_1 - q_2))$. □

7.4 Free Use of Gates

We allow free use of gates in the **do**-part of **factor**; instead of only right after the key word **do**.

Example 7.5 Let $R_1 = AB$ and $R_2 = AC$. Then

factor r_1, r_2 **do** $@(1) - (A = 5?\ \delta_{C \leftarrow B}(@(2)))$

is equivalent to

$$
\begin{array}{ll}
 & \textbf{factor } r_1 \bowtie lim(b)[\pi_A(r_1)/x],\ r_2 \bowtie lim(b)[\pi_A(r_2)/x] \textbf{ do} \\
 & \quad @(1) - \delta_{C \leftarrow B}(@(2)) \\
\cup & \textbf{factor } r_1 \bowtie lim(\neg b)[\pi_A(r_1)/x],\ r_2 \bowtie lim(\neg b)[\pi_A(r_2)/x] \textbf{ do} \\
 & \quad @(1) - \{A, B\}(\emptyset)
\end{array}
$$

where b is $A = 5$. □

7.5 Nested Factors

That is, **factor**'s used in the **do**-part of another **factor**; also in gates. We show how we can remove one **factor** (a last) from a sequence of **factor**'s By applying this repeatly, we can translate the whole expression to relational algebra. Consider

$$\ldots \textbf{do} \ldots \textbf{factor } e_1, \ldots, e_n \textbf{ do } e \ldots$$

where e does not contain **factor**'s except maybe in gates. If $e_1, \ldots, e_n$ contain **factor**'s, recursively remove these first as we did in the simple case. Then recursively remove **factor**'s in gates in e. Now remove gates from e, and then this **factor** can be translated into relational algebra.

Remark 7.6 Notice the scoping here. In

$$\textbf{factor } r_1, r_2 \textbf{ do factor } @(1), r_3 \textbf{ do } @(1)$$

for examples the two @(1)'s are different.

7.6 Narrowing Down Factorization Attributes

We find it more natural to allow

$$\textbf{factor } r_1 \ldots r_n \textbf{ on } A_1 \ldots A_k \textbf{ do } e$$

instead of writing

$$\textbf{factor } r_1 \ldots r_n, \{A_1 \ldots A_k\}(\emptyset\} \textbf{ do } e$$

However, this is merely a syntactical convenience.

7.7 Nested @(i)'s and #'s

We will allow $\backslash @(i), \backslash\backslash\ @(i)$, etc. to denote $@(i)$'s from higher levels when using nested **factor**'s. These are translated as constants except that one $\backslash$ is removed, eg., $\backslash\backslash\ @(i)$ is translated into $\backslash @(i) \times \delta_{X \leftarrow Y}(b_{\mathcal{F}})$.

Having nested **factor**'s also means that several #'s are active at a time. We would like to have access to the values bound to attribute names in each of them. In order to do so, we change the syntax from simply an A referring to the nearest #, to $\#.A$. We can now also allow $\backslash\#, \backslash\backslash\ \#$, etc. to denote these #'s from higher levels; and we access attributes by e.g., $\backslash\backslash\ \#.B$.

We want to translate expressions containing these constructs in a similar way as to how we handled nested $@(i)$'s. There is the problem that if we have a comparison such as $\#.B = \backslash\#.A$, this would result in selections like $\sigma_{B=\#.A}(r)$, which we have not treated previously (because # appears in a selection condition). We notice, however that this can instead be written $\pi_R(\sigma_{B=B'}(\{[B' :\ \#.A]\} \times r))$, the translation of which will be dealt with automatically.

In the definition of TT_X, we take into account the schema (domain) of #. This definition will have to be extended, so that the right schema can be given also for, say, $\backslash\backslash\ \#$. When extended properly, we can still obtain the normal form we want; the difference being that instead of simply having constants and constructs like $\#.A$ as values, we will also have constructs like $\backslash\backslash\ \#.A$.

7.8 Computations on Domain Values

Before continuing, we equip relational algebra with an additional operation, **extend** [Gra84], and with some operations on domain values to be used with **extend**. This operator takes one relational argument and can perform computations on domain values for each tuple individually. As an example, we can write

$$\textbf{extend}\ r\ \textbf{with}\ A := (B + 5)/C$$

where $B, C \in R$ and $A \notin R$. In general, a query

extend r **with** $A :=$ `<atom exp>`

can be expressed as follows in FC.

factor r **do** $\#[A :$ `<atom exp>`$]$

In order to translate FC expressions, where these computations on domain values can appear anywhere the types match, into relational algebra with **extend**, we change the definition of TT_X again to incorporate operations on atomic types. We can now obtain a normal form which is a tuple like before, but the entries are now atom expressions over factorization tuples selections.

Example 7.7 Let $t = [B_1 : \ \#.A_1 + \backslash\#.A_2 + \backslash\backslash\,\#.A_3, B_2 : \ d_2, \ldots, B_k : \ d_k]$, and assume that $[B_2 : \ d_2, \ldots, B_k : \ d_k]$ translates into q'. The tuple t is then translated into

$$\pi_{(Q'' \cup \{B_1\}) \setminus \{\hat{A}_2, \hat{A}_3\}}(\textbf{extend } q'' \textbf{ with } B_1 := A_1' + \hat{A}_2 + \hat{A}_3)$$

where A_1' is the unique copy of A_1 (which we know is in Q') and q'' is

$$q' \times \{[\hat{A}_2 : \ \#.A_2]\} \times \{[\hat{A}_3 : \ \backslash\#.A_3]\}$$

That is, a single # is removed and the trailing attribute name changed to its unique copy. Furthermore, one level of \'s is stripped off. By inventing new names, the **with**-part of **extend** can be finished immediately; the argument to **extend** will, if necessary, be dealt with later by rules already defined. □

7.9 Aggregate Functions

We equip relational algebra with aggregate functions and the operator **group_by** [Gra81]. As for **extend, group_by** is unary. There is a list of attributes on which to *group*. These should all belong to the schema of the single relational argument. For example, if $R = \{A_1, A_2, A_3\}$, we can write

group r **by** A_1 **creating** $B :=$SUM(A_3)

Now, let Φ be an aggregate function. We can translate

$$\textbf{group } r \textbf{ by } A_1, \ldots A_k \textbf{ creating } A' := \Phi(A'')$$

into

$$\textbf{factor } r \textbf{ on } A_1, \ldots A_k \textbf{ do } \#[A' : \ \Phi_{A''}(@(1))]$$

Our notation in connection with aggregate functions is slightly different as the argument relation is not implicit as in **group_by**. Our choice of notation is more in line with standard notation for the other relational operators.

In order to translate FC expressions with aggregation to relational algebra with **group_by**, we (recursively) remove appearances of **factor** inside aggregate functions first. We obtain (again changing the definition of TT_X) a normal form whiGh can include entries that are aggregate functions applied to relational algebra expressions.

Example 7.8 Consider $t = [B_1 : \ \Phi_{A''}(r), B_2 : \ d_2, \ldots, B_k : \ d_k]$, and assume that $[B_2 : \ d_2, \ldots, B_k : \ d_k]$ translates into q' and r translates into q''. Then t translates into

$$q' \bowtie \textbf{group } q'' \textbf{ by } Q' \textbf{ creating } B_1 := \Phi(A'')$$

□

7.10 Aggregation And Domain Computation

It turns out that when both of these facilities are present simultaneously, it is even more difficult to translate FC expressions to relational algebra. TT_X is changed again (and so is the normal form). The entries in the tuple in normal form can now be atom expressions over the same variables as under **extend**, but now also aggregate functions. The techniques from above are combined.

Example 7.9 Consider $t = [B_1 : \ \Phi_{A''}(r) + \backslash\backslash \#.A_j, B_2 : \ d_2, \ldots, B_k : \ d_k]$, and assume that $[B_2 : \ d_2, \ldots, B_k : \ d_k]$ translates into q' and r translates into q''. The tuple t is then translated into

$\pi_{(Q''\cup\{B_1\})\setminus\{\hat{A},\hat{A}'\}}$ (**extend**
$q''' \bowtie$ **group** q'' **on** Q'' **creating** $\hat{A} := \Phi(A'')$
with $B_1 := \hat{A} + \hat{A}'$)

where $q''' = q' \times \{[\hat{A}' : \ \backslash\#.A_j]\}$. □

7.11 Summarizing

In summarizing, we want to state results for a number of subsets of FC. To be mecise, we first define FC$^+$ by the following grammar.

`<atom>`	::=	$c \mid \#.A \mid \backslash\#.A \mid \backslash\backslash\#.A \mid \ldots$
`<tup>`	::=	$[\,] \mid [A :$ `<atom>`$, \ldots, A :$ `<atom>`$] \mid$ `<tup>` $\backslash A \mid$
		`<tup><tup>` $\mid \# \mid \backslash\# \mid \backslash\backslash\# \mid \ldots$
`<rel>`	::=	$\{\,\} \mid \{$`<tup>`$\} \mid$ `<rel>` $\times$ `<rel>` $\mid$
		$@(1) \mid @(2) \mid \ldots \mid \backslash@(1) \mid \backslash@(2) \mid \ldots$
		`<cond>` ? `<rel>` $\mid$ **factor** `<rel>`$, \ldots,$ `<rel>` **do** `<rel>` $\mid$
		factor `<rel>`$, \ldots,$ `<rel>` **on** $A, \ldots, A$ **do** `<rel>`
		`<rel>` $\cup$ `<rel>` $\mid$ `<rel>` $-$ `<rel>` $\mid$ `<rel>` $\bowtie$ `<rel>` $\mid$
		$\pi_{A\ldots A}$(`<rel>`) $\mid \sigma_b$ `<rel>`$, \ldots$
`<cond>`	::=	`<atom>` $=$ `<atom>` $\mid$ `<atom>` $\neq$ `<atom>` $\mid$
		`<atom>` $<$ `<atom>` $\mid$ `<atom>` $\leq$ `<atom>` $\mid \cdots$
		`<rel>` $=$ `<rel>` $\mid$ `<rel>` $\neq$ `<rel>` $\mid$ `<rel>` $\subseteq$ `<rel>` $\mid \ldots$

ln previous sections, we have proved that SFC and relational algebra are equivalent in expressive power. In this section, we have given the necessary definitions to the proof of FC$^+$ being equivalent to relational algebra.

Let SFC^{+c} be as FC$^+$, but with the addition of

`<atom>` ::= `<atom>` $+$ `<atom>` $\mid$ `<atom>` $-$ `<atom>` $\mid \ldots$

SFC^{+c} is equivalent to relational algebra with extend (with the same operations on atom types available).

Let FC^{+a} be as FC$^+$, but with the addition of

<atom> ::= SUM_A (<rel>) | COUNT_A <rel> | ...

FC^{+a} is equivalent to relational algebra with **group_by** (and the same aggregate functions available).

Finally, FC is FC^{+} with both of the above additions, and FC is equivalent to relational algebra with both **extend** and **group_by**.

8 Conclusion

We have introduced the language FC in which aggregation and computation on domain values fit in naturally. We believe that FC could form a more elegant basis for language development than standard relational algebra when these features should be included.

An interesting direction of research would be to explore the possibilities of a Query-By-Example [Zlo77] style query language, where the novice user could program via example data instead of **factor** directly; and especially instead of using the #'s and @(i)'s directly.

If a language is going to form a basis for design of new languages, it is important to know the computational power of the language. We have proved that FC without aggregation and computation on domain values is equivalent to relational algebra. With aggregation and/or computation on domain values, it is equivalent to relational algebra with group_by (and the same aggregation functions) and/or extend (and the same operations on domain values).

A simple version of FC, in which functions have also been included, has been implemented [Lar92].

References

[Cod72] E. F. Codd. Relational Completeness of Data Base Sublanguages. In Randall Rustin, editor, *Data base Systems*, pages 65–98. Prentice-Hall, 1972.

[GB79] P. M. D. Gray and R. Bell. Use of Simulators to Help the Inexpert in Automatic Program Generation. In P. A. Samet, editor, *Euro IFIP*, pages 613–620. North-Holland, 1979.

[Gra81] Peter M. D. Gray. The GROUP_BY Operation in Relational Algebra. In S. M. Deen and P. Hammersley, editors, *Databases*, pages 84–98. Pentech Press Limited, 1981.

[Gra84] Peter M. D. Gray. *Logic, Algebra and Databases.* Ellis Horwood Limited, 1984.

[Klu82] Anthony Klug. Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. *J. ACM*, 29(3):699–717, 1982.

[Lar92] Kim S. Larsen. Xrasmus User's Manual. MD 60, Computer Science Department, Aarhus University, 1992.

[LS] Kim S. Larsen and Michael I. Schwartzbach. Optimal Detection of Query Injectivity. Computer Science Department, Aarhus University, 1991.

[LSS92] Kim S. Larsen, Michael I. Schwartzbach, and Erik M. Schmidt. A New Formalism for Relational Algebra. *IPL*, 41(3):163–168, 1992.

[Mai83] David Maier. *The Theory of Relational Databases.* Computer Science Press, 1983.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[Zlo77] M. M. Zloof. Query-by-Example: a data base language. *IBM Systems Journal*, 16(4):324–343, 1977.