World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:2, No:11, 2008

Expressive Modes and Species of Language

Richard Elling Moe

Abstract — Computer languages are usually lumped together into broad 'paradigms', leaving us in want of a finer classification of *kinds* of language. Theories distinguishing between 'genuine differences' in language has been called for, and we propose that such differences can be observed through a notion of *expressive mode*. We outline this concept, propose how it could be operationalized and indicate a possible context for the development of a corresponding theory. Finally we consider a possible application in connection with evaluation of language revision. We illustrate this with a case, investigating possible revisions of the relational algebra in order to overcome weaknesses of the division operator in connection with universal queries.

Keywords — Expressive mode, Computer language species, Evaluation of revision, Relational algebra, Universal database queries

I. Introduction

THE evolution of computer languages has produced a variety of kinds of languages and programmers' preferences differ. A quick look at computer language discussion forums will reveal the warm feelings some has for their favorite (kind of) languages, as well as spiteful resent for others. We remain neutral to rankings of languages, but observe that the rich variety brings out the question of what kind, or *species*, of languages exist.

> *... we want, that is, a theory that is sensitive to genuine differences between languages, and only to those.* (Graham White [26])

Computer languages are usually lumped together into broad classes or 'paradigms', such as imperative or functional, which renders the classification of language in terms of species to rather coarse judgements. Therefore we understand White's statement as a call for a finer classification of kinds of language.

Pigott [18] presents a taxonomy in which programming languages are classified in terms of being conversational, imperative, operation-oriented, expression-oriented, a lambda-calculus etc as well as their membership in families descending from early languages. I.e. Algol-family, Fortran-family etc. We suggest an other approach. Since computer languages has different repertoires of *mechanisms* we might hope that 'genuine differences' may be operationalized in terms of selections of mechanisms. The extent to which such classification coincides with Piggott's taxonomy remains to be seen.

White does not provide much detail about the structure and content of his desired theory. In the following we propose the concept of *expressive mode* as a possible basis. We shall only outline the concept and rely on examples to flesh out the idea and demonstrate its relevance. Beyond that we will merely point to the discipline of formal semantics as a possible context for the development of such a theory.

Finally, we suggest how expressive mode may be applied in connection with evaluation of revisions of language. We illustrate our points with a case: In the relational algebra, so-called universal queries are traditionally handled by means of the division operator ÷ [3]. However, it has been pointed out that this approach isn't suitable for all kinds of universal quantification [2], [4], [7], [8], [10], [17]. Moreover, it has been suggested that such queries should be dealt with in an entirely different manner:

> *... we should point out that queries of that general nature are often more readily expressed in terms of relational comparisons.*
> (Date [6] on universal queries and division)

This gives rise to the question of whether the relational algebra should be revised to facilitate a different approach to universal queries. We aim to demonstrate that expressive mode may have an impact on the decision.

II. Expressive Mode

The expressive mode of a language concerns the line of thinking involved when forming expressions. Different languages lend themselves to different lines of thinking. Contrast an imperative language with a language for logic programming. The imperative line of thinking does not translate naturally to the workings of logic programming. Clearly these are two very different ways of going about the business of programming.

So, we have every reason to believe that there are genuinely different languages, but what makes up the differences? Theories of expressive power provides little help since languages of equal power would fall under the same theory. Clearly we need to look elsewhere for distinctive features of languages, which in turn may form the basis for an operationalized understanding of the otherwise rather intangible concept of expressive mode.

To our knowledge, no formal foundation for a notion of expressive mode has been established. Well-known lines of division, such as the declarative/procedural dichotomy, are clearly relevant but form too general classes of languages. This would also be the problem if one links the notion of mode too strongly to programming language paradigms such as logical vs functional vs imperative languages. The problem here is one of granularity. Whereas these notions certainly distinguish entire languages from each other, they provide no detailed account as to why. For a more finely grained approach we suggest focusing at the mechanisms-level.

Manuscript received August 27, 2006
Richard E. Moe is with the Department of Information science and Media studies, University of Bergen, PO Box 7800, NO-5020 Bergen, Norway. Email: Richard.Moe@infomedia.uib.no

A. Computer Language Mechanisms

Computer languages provide the means for describing objects and processes. Each language offers a number of mechanisms, and the repertoire surely influences how one goes about solving problems. It seems reasonable to believe that many aspects of expressive mode can be viewed from a mechanism-perspective.

Given the great variety of languages used on computers we should be careful not to fix a too narrow definition of the term *mechanism*. For the present discussion we merely provide a list of examples of concepts found in common computer languages that should qualify as mechanisms: *Variables (local or global; updatable or not), assignment (destructive or not), types (simple or complex), coercion, sequencing, conditionals, operators, functors, pointers, arrays, lists, streams, predicates, connectives, quantifiers, relations, functions, procedures, calls, parameters, binding, scope, nesting, recursion, loops, jumps, modules, higher-order functions, objects, classes, subsumption, inheritance, polymorphism, concurrency, threads, delayed and forced evaluation, quotation, backtracking, cuts, destructive functions, mutable data* .... No claim of the list being exhaustive is implied, nor that all items are relevant for the notion of expressive mode. Our concept of mechanism is independent of the notation used to represent it.

In the following we try to further explicate the concept of expressive mode through an example. As a prelude to our case of application in section IV we focus on the mechanisms of relational comparison and nested queries as found in relational algebra.

III. Example

The relational algebra is the original query-language of the relational model for databases [3]. There is a wealth of literature in which notation and terminology varies. We adopt that of Elmasri and Navathe [12].

For the purpose of illustration let us consider the following schema for a database containing facts about employees, departments and projects of an imaginary enterprise:

- $EMP(\underline{ID}, DNO)$ relates the IDs of employees to the ID-numbers of the departments that employ them.
- $PROJ(\underline{PNUM}, DNUM)$ relates each project, given by a project-number, to the department that controls it.
- $WORK(\underline{EID, PNO}, HOURS)$ relates employees and the projects they work on along with the number of hours per week they put in.

The operations of relational algebra produce relations from relations. Either by means of set-operations or using some of the operators introduced specifically for the algebra, such as projection π, selection σ, join $\bowtie$, attribute renaming ρ and assignment $\leftarrow$ (cf [12] for details).

A. Relational Comparison and Relational Algebra

In connection with database query-languages the term *relational comparison* refers to predicates, for use in selection-conditions, where one or more of the arguments are relations. In SQL we find for instance `in`, `exists`, `>all` and `contains`.

The selection-operator σ_φ retrieves from a relation the tuples which satisfy the selection-condition φ. Selection-conditions are simple, quantifier-free, boolean formulas using elementary attribute comparison-operators such as equality, 'less than' etc for domains of *atomic* values. Furthermore, in selections $\sigma_\varphi(R)$, it is traditionally required that φ refers only to attributes in the relation R.

Relational comparison operators on the other hand would allow more complex selection-conditions. Take for instance the task of retrieving the departments that employ people working more than 20 hours on a project. A straightforward solution would be:

$$\pi_{DNO}(\sigma_{HOURS>20}(EMP \bowtie_{ID=EID} WORK))$$

Now consider how it could be done had the set element relation $\in$ been available for use in selection conditions:

$$\pi_{DNO}(\sigma_{ID\in\pi_{EID}(\sigma_{HOURS>20}(WORKS))}(EMP))$$

Here the outermost σ-conditions employs the $\in$-operator for which the second argument is a relation. But this violates the traditional requirement that arguments in such conditions should be atomic values.

Suppose now that the query was restricted to retrieve the departments with employees working more than 20 hours on any of their own projects. Typically the solution would be to define an auxiliary relation where information about the controlling departments of projects have been integrated in the $WORK$-relation.

$$WORK' \leftarrow WORK \bowtie_{PNO=PNUM} PROJ$$
$$WO \leftarrow \pi_{EID,PNO,DNUM,HOURS}(WORK')$$

Now the above queries could be modified as follows.

$$\pi_{DNO}(\sigma_{HOURS>20\wedge DNUM=DNO}(EMP \bowtie_{ID=EID} WO))$$

and

$$\pi_{DNO}(\sigma_{ID\in\pi_{EID}(\sigma_{HOURS>20\wedge DNUM=DNO}(WO))}(EMP))$$

Note that in the second query the σ-condition $HOURS > 20 \wedge DNUM = DNO$ refers to an attribute (DNO) outside the scope of the σ-operator. This violates the traditional requirements for applications of σ, whereas in the first query the corresponding σ-condition is a valid expression. Allowing relational comparison operators to be used in this manner would in effect introduce the kind of *correlated nested queries* we find in SQL, but which is not part of the original algebra repertoire.

We believe that the introduction of relational comparison and/or correlated nested queries would alter the dynamics of the language and the line of thinking we employ when dealing with queries. Not only can the σ-conditions involve far greater complexity, but an entirely different approach to solving queries would emerge and thus a possible shift in the pragmatic dimension of the language.

It seems reasonable to conclude that the adoption of such mechanisms would affect the expressive mode of the algebra, perhaps. On the other hand, we can hardly claim it would represent a transition from one paradigm to another. Thus, the example supports the view that modes and paradigms,

although related concepts, differ in levels of detail and granularity.

B. Semantics as Foundation for a Theory of Mode

Where could we find the basis for a proper theory of expressive mode, operationalized in terms of mechanisms? If we look for others who need to identify and reason about mechanisms we find those concerned with programming language semantics.

Traditionally, the semantics of programming languages are specified by means of denotation functions based on Scott domains [24], [26]. This provides a sufficiently detailed framework for overcoming the problem of coarse granularity. But there is also the danger of producing a too finely grained account. Hence, an important concern is that the theory should establish suitable *levels of abstraction* reflecting the desired granularity, rather than just reducing everything to first principles.

White's reflections [26] cited above contains similar thoughts. Although he does not mention mechanisms as the basis for the theory, we observe that our proposal is consistent with White's considerations.

As formal semantics maps syntactic constructs onto denotations over Scott domains, every mechanism must be embedded in some syntactic representation. The essence of its nature is then reflected at some (abstract) level of denotation. Finding the right level is essential to obtaining a proper conceptualization of mechanisms. Only then can the theory be a vehicle for identifying and representing genuine differences and give rise to a taxonomy of languages, which ultimately could form the basis for a theory of expressive mode.

The goal for such a theory would be to form a *thesis* of correspondence between mechanism-defined categories and modes. I.e principles for how mechanisms can be used to identify or distinguish between expressive modes. It does seem obvious that mechanisms are relevant to the question of mode, after all we can identify 'hallmark' mechanisms of programming paradigms. On the other hand it is unlikely that every combination of mechanisms you might conceivably put into a language defines a unique line of thinking.

It is beyond our present scope to attempt a formulation of such a thesis. Still, under the assumption that it could be found, we proceed to present a possible application for a mechanism-oriented concept of expressive mode.

IV. Application: Evaluating Language-revision

Different kinds of languages might lead to different styles of problem solving. Expressive mode is thus an important factor in the pragmatic dimension of a language and should therefore have a role in the evaluation of computer language revisions.

When languages evolve, a certain conservativeness comes naturally with respect to the expressive mode. There should be no major shifts in the pragmatics of forming expressions.

Presumably one would introduce changes in a language for the purpose of enhancing it in some way. Often with respect to the qualities we may refer to as *expressive power* and *expressive convenience*. We hold that such developments should not be seen in isolation from expressive mode.

The three concepts involves different kinds of methodology. Whereas the expressive power is a mathematical inquiry, the expressive convenience rests crucially on human preference. Its study therefore relies on methods resembling those applied within Human-Computer Interaction (HCI) [11], a discipline which has absorbed techniques from a variety of fields, including psychology and the social sciences. Importantly, it embraces 'human factors' which are central in earlier evaluations of computer languages [20], [21], [23], [25].

In the following we consider these concepts and their roles as criteria for the evaluation of language revision.

A. Expressive Power

Theoretically, a formal language has a certain *expressive power*. I.e. the potential for what might be achieved using the language, regardless of how easy or hard it may be to write the code.

A notion of expressive power could also apply to individual mechanisms of languages (see section II-A). Revision of a language may well be through the strengthening of mechanisms, but may be for convenience only and need not change the expressive power of the language as a whole.

In its purest form, the relational algebra comprises the selection and projection operators σ and π, the set operations $\cup$, $-$ and $\times$, in addition to the renaming operator ρ. In principle, these operators are all that is needed. They define the expressive power of relational algebra[1]. Any set of operators which is equally expressive, or more, is said to be *relationally complete*. This way of measuring the expressive power readily lends itself to other languages [14], [15]. In fact, relational completeness has become a minimum requirement for relational database query languages.

Other operators has been added to the algebra for practical reasons, such as the join $\bowtie$ and division $\div$. Even if their introduction makes no difference when it comes to the power of the algebra, certain types of queries seems to be more readily expressed when these operators are available. Apparently, their presence in the language is entirely for convenience.

B. Expressive Convenience

Expressive convenience is the quality of a language related to its ease of use and suitability. In contrast, where expressive power concerns what *can* be expressed, the expressive convenience is a matter of *how* it may be expressed.

Expressive convenience is a very different sort of inquiry than expressive power, not least in terms of the methods involved. It could be studied by adapting HCI-techniques [11] dealing with human factors and ease-of-use. Trovåg [22] works along these lines when he operationalizes expressive convenience in terms of effectiveness and learnability.

Occasionally, claims for convenience are supported by reference to syntactic complexity. Suppose you have revised a

[1]We disregard extensions of the algebra designed to add to its original power, such as aggregate functions and recursive closure operators.

mechanism by adding to its power. It might be tempting to prove its convenience through examples showing that certain expressions may be simplified in some superficial way. For instance in terms of the length of the expression, depth of nesting or the number of operators used. This is a pitfall. Even if such examples can demonstrate aspects of expressive convenience they will not suffice to establish an overall gain in this respect. Plausibly, an increase in power of the mechanism may render it more difficult to understand and use, causing a loss of expressive convenience which measures of syntactic complexity may fail to register.

Mechanism may have different syntactic manifestations. Such apparent differences will by nature be superficial. Compare, for instance, ontology-specifications in description logics with those of RDF-based ontology-languages like OWL or DAML+OIL. The differences in terms of verbosity is striking. Nevertheless there is a close correspondence between these languages and how they describe concepts [1]. We hold that measures based on syntactic complexity alone are too superficial. Reisner's investigation [20] include reflections over syntactic form, but we maintain that human factors are essential for evaluation.

Past evaluations of computer languages mainly deal with aspects of convenience when trying to provide answers to questions like *'How difficult is it to learn the language?'*, *'How well do operators perform using the language?'*, *'How much effort is required to use the language?'* and *'How well do the features of the language match the tasks to be accomplished?'* [23], all with a focus on human factors and ease-of-use.

Human factor studies are typically conducted by measuring the performance and/or polling the opinions of a suitable selection of human respondents. Whereas HCI mainly deals with end-users, computer programmers usually have much higher levels of computer expertise. This does not imply that less attention is required with regard to the selection of respondents. The evaluation of computer languages in terms of expressive convenience is by no means neutral to the skills held by the users. In some circumstances, for instance when complex mechanisms are evaluated, expert respondents would be called for.

With skilled respondents the evaluation can be *cooperative* in the sense that their professional opinions are also input for the analysis, and not only the observation of their behaviour.

C. Criterion Interdependence

A change for the better with respect to one criterion could be a change for the worse with respect to some other. Hence, a proposed change may require evaluation along several dimensions before being adopted into the language.

Enhanced expressive convenience may result from increasing the power of a single mechanism, but not necessarily. Added strength may lead to higher complexity which could in turn render the mechanism difficult to understand and use.

Having revised a language with an apparent gain in expressive convenience, there is still a need to check whether the revision has caused changes in the overall expressive power. A loss of power is unlikely to be acceptable, but could occur if one mechanism is replaced with an other. Furthermore, one should not add to the expressive power of a language without some consideration. It is not always the case that 'The more power the better!' because powerful languages are also complex languages so desired properties may be harder to maintain. For example: It is very hard, if not impossible, to implement a truly declarative programming language. Whereas for less powerful languages, such as query-languages, it may be easier to obtain declarative qualities.

It is not so apparent whether a shift of mode can have an effect on the expressive convenience of a language. There are some indications that database users perform better with procedural query languages than with declarative ones [25], but because of the confinement to query languages and the procedural/declarative dichotomy, these findings will not bring us to conclude that some modes are objectively more convenient than others.

When languages are merged, the result may be a mix of modes. (We are reluctant to think that a mix of modes necessarily represents a separate unique mode.) For instance, the object-relational model for databases combines the declarative SQL query language with an imperative object-oriented programming language for defining structured objects and operations associated with them. Plausibly, having to switch between, or even mix, lines of thinking within the same language could pose a threat to expressive convenience.

V. Case: Universal Queries in Relational Algebra

Universal queries involve forms of universal quantification, similar to the kind expressed by means of the $\forall$ quantifier in first order logic. For example:

Find the employees who work on all projects run by department 5. (1)

Universal queries are somewhat complicated compared to the typical database query. They have received a fair amount of attention, both with respect to their specification [2]–[4], [7], [10] and the algorithms for processing them [13], [19].

Database query languages come in different shapes, with different approaches to handling universal queries. In relational algebra the *division* operator is typically called into action when specifying a universal query.

A. Division

Division ($\div$) operates on two relations where the attribute-set of the first properly includes that of the second. Given relations R and S with attribute-sets X and Y, respectively[2], such that $Y \subset X$; $R \div S$ denotes a relation with attribute-set $X - Y$. Specifically, $t \in R \div S$ iff $\{t\} \times S \subseteq R$.

The division operator was designed for universal queries, which otherwise are complicated to express. Using the division operator $\div$ query (1) may be formulated as follows:

$$WO \leftarrow \pi_{EID,PNO}(WORK)$$
$$PRO5 \leftarrow \rho_{(PNO)}(\pi_{PNUM}(\sigma_{DNUM=5}(PROJ)))$$
$$RESULT \leftarrow WO \div PRO5$$

[2]Technically, the attributes are ordered within the relation-schema. We do not go to this level of detail since the reader can easily induce the necessary order on the fly.

There has been numerous reports of $\div$ being obscure and difficult to learn [2], [10], [16]. Furthermore, some voices has been raised against its usefulness and generality [2], [4], [7], [8], [10].

We now look into a kind of universal query for which the standard division operator is insufficient [17]. Consider the following example:

Find the employees who work on all projects run by their own department. (2)

The crucial difference when it comes to the use of division is that for query (1) the 'divisor' ($PRO5$) is unique and pre-defined, whereas this query corresponds to a *series* of divisions, each with a different divisor. Moreover, this series can not be pre-defined since it relies on the database state, which changes over time.

Before we elaborate on how the division line of thinking may still be an approach to query (2) we consider some alternatives.

It is well known that universal queries can be expressed using relational comparison operators. Date [6], [8], [9] argues that such operators would have advantages over the traditional operators of relational algebra, and maintain that they offer a more convenient solution for universal queries. Adapting his approach, employing the relational comparison operator $\subseteq$, query (2) might look like this:

$$\sigma_{\pi_{PNUM}(\sigma_{DNUM=DNO}(PROJ))\subseteq\pi_{PNO}(\sigma_{EID=ID}(WORK))}(EMP)$$

There is another approach to query (2) without the use of relational comparison [5], [17]:

$$EMP \bowtie_{ID=EID} (\pi_{EID,PNO}(WORK) \div \pi_{PNUM}(\sigma_{DNUM=DNO}(PROJ)))$$

However, correlated nesting is still present in that the σ-operation refers to the attribute DNO which belongs to the EMP-relation.

These may well be alternative ways of expressing query (2) and some would prefer them to the division-approach, but for reasons discussed in sections III-A and IV, relational comparison and/or correlated nested queries should not be adopted just like that. Moe [17] proposes to modify the division operator for the purpose of solving such queries, rather than introducing alien mechanisms. The *context-sensitive* division operator [17] adds to the power and applicability of $\div$ while retaining the original expressive power, and presumably the mode, of the algebra as a whole.

Let R and S be relation-schemas with attribute sets X and Y respectively. Suppose $T \subseteq X$, $X \cap Y = C$ and $C \cap T = \emptyset$. Let φ be a boolean condition over $X \cup Y$. For the intended use, T and C will be non-empty. Now, *context-sensitive division* $R \frac{T}{\varphi} S$ is defined to be equivalent with

$$\pi_T(R) - \pi_T(\pi_{T\cup C}(\sigma_\varphi(\pi_{X-C}(R) \times S)) - \pi_{T\cup C}(R))$$

This amounts to a generalization of $\div$: If $X \cup Y = T \cup C$ and φ evaluates to true in all cases, then context-sensitive division boils down to being the same as ordinary division.

Using context-sensitive division, query (2) can be expressed conveniently as follows:

$$WO \leftarrow \pi_{ID,DNO,PNO}(EMP \bowtie_{ID=EID} WORK)$$
$$WO' \leftarrow \rho_{(ID,DNO,PNUM)}(WO)$$
$$RESULT \leftarrow WO' \frac{ID}{DNO=DNUM} PROJ$$

The usefulness of context-sensitive division extends beyond the specific kind of queries discussed above. An example can be found in [22]. In addition, we find a further class of universal queries which appears to be fundamentally different from query (2) and for which $\div$ is of little use: Let R be a relation with a single number-valued attribute N. The query for the least number in R would typically be solved by means of aggregate functions, but, in fact, this *is* a universal query. Whereas $\div$ is insufficient for finding minimal values, context-sensitive division handles the situation with relative ease [17]:

$$(\rho_{(M)}(R) \bowtie_{M=N} R) \frac{M}{M\geq N'} (R \bowtie_{N=N'} \rho_{(N')}(R)) \quad (3)$$

B. Evaluating the Revised Mechanism

Context-sensitive division has a clear advantage in being more powerful than $\div$, extending the division-approach to a wider range of universal queries [17]. The expressive power of the algebra as a whole is left unchanged by the revision. The expressive mode is also likely to be retained since the new operator is clearly a conservative generalization of $\div$.

Aside from that, syntactic measures provide some suggestive evidence of context-sensitive division being more convenient than $\div$. For instance, query (1) becomes less verbose:

$$PROJ \leftarrow \rho_{(PNO,DNUM)}(\pi_{PNUMBER,DNUM}(PROJECT))$$
$$RESULT \leftarrow WORK \frac{EID}{DNUM=5} PROJ$$

Specifically, the solution for query (1) in section V-A uses 8 operators while this has 5, assignments and renaming included. However, it is questionable whether this reflects a genuine improvement of expressive convenience since the user must handle the extra parameter.

This brings us to a crucial question. Does the inconvenience of added complexity cancel out the convenience gained from the extra power? Now we are in human factors territory. The utility of the operator should be contrasted with the ease and transparency of its use. Query (3) above, for the least number in R, might not be very promising in this respect. Surely the solution is concise, but the trick of joining R to itself is perhaps not something that would spring to mind or suggest itself naturally. Furthermore, at first sight the reader may well jump to the conclusion that the use of $\geq$ in finding the *least* number must be a mistake. But, it isn't. Anyhow, these are only indications and a proper investigation is called for.

The human factors of context-sensitive division have been investigated by Trovåg [22], adapting HCI-techniques. His respondents was recruited among very capable students having gained a basic knowledge of relational algebra and the division operator through an introductory database course. Given the selection of (near-)expert users, Trovåg conducts a cooperative evaluation using the *Think Aloud Protocol* [11] to collect responses. In the choice between participatory and non-participatory role of the observer, Trovåg opts for middle

ground allowing restricted intervention while stressing the need for an intervention protocol to secure equal treatment of all respondents and to avoid giving away too many clues during the interaction.

Trovåg's investigation tends to support the conclusion that the expressive convenience of relational algebra would gain from incorporating context-sensitivity into the division operator.

C. Relational Comparison vs Context-sensitive Division

We know of no investigations into the expressive convenience of relational comparison but, for the sake of discussion, let us accept Date's confident claim for its qualities in this respect. Then the question of how to handle universal queries becomes a choice between the two mechanisms. Both are deemed convenient but, in Date's opinion, relational comparison is more so than division. So without regard for expressive mode, relational comparison might be the solution. However, the introduction of relational comparison may affect the expressive mode of relational algebra. Perhaps too much, since it might involve changes beyond what can reasonably be called conservative towards the feel and dynamics of the language. Hence, bringing the dimension of expressive mode into the picture could make a difference.

VI. Conclusion

Our first goal has been to outline, and demonstrate the relevance of, a notion of expressive mode. Developing a corresponding formal theory is beyond the present scope, but we have pointed out the concept of mechanism as a key component and suggested an approach building on the theory for programming language semantics. Should the semanticists follow up on White's reflections [26] and endeavor to produce a taxonomy of languages based on mechanism, it could form the basis for a formal theory and a thesis of correspondence between mechanisms and modes. With such an approach it is the success in attaining a suitable degree of abstraction that could eventually give rise to an operationalized concept of expressive mode.

We have considered expressive mode as an evaluation criterion for (revisions of) computer languages. Past evaluations of computer languages has mainly focused on human factors, and new mechanisms often receive only a superficial treatment.

The evolution of a computer language should be a process of developments for the better in some respect but conservative towards the feel and pragmatics of the language. Hence, revisions should not be done by whim. We propose that expressive mode, as well as convenience and power are important, and interdependent, factors to be taken into consideration in order to avoid languages evolving haphazardly.

As for the relational algebra, none of the alien mechanisms that we have surveyed has been seriously proposed for adoption, though profiled members of the database community comes pretty close to doing so. C. J. Date, quoted in section I, clearly states his preference for relational comparisons over division. Moreover, the correlated nested query presented in section V-A originates from Date's suggestion [5] of how to solve query (2).

Should a version of the algebra emerge that includes such mechanisms, we might conclude that there has been a shift in the expressive mode, and subsequently debate whether it has been for the better. Or even whether the revised language should be referred to as relational algebra at all.

References

[1] Baader, F., Horrocks, I., Sattler, U., 2005. Description Logics as Ontology Languages for the Semantic Web. In: D. Hutter, W. Stephan (Eds). Mechanizing Mathematical Reasoning, LNAI 2605. Springer Verlag.
[2] Carlis, J. V., 1986. HAS, a Relational Algebra Operator or Divide is not Enough to Conquer, Proceedings of the second international conference on data engineering, IEEE Computer Society
[3] Codd, E. F., 1972. Relational completeness of data base sublanguages. In: Justin, R. J. (ed) Data base systems, Courant computer science symposia series 6, Englewood Cliffs, N.J., Prentice-Hall
[4] Dadashzadeh, M., 1989. An improved division operator for relational algebra, Information systems 14(5),
[5] Date, C. J., 2003. On relational division, Discussion on Database Debunkings website, www.dbdebunk.com
[6] Date, C. J., 2004. An introduction to database systems, Addison Wesley
[7] Date, C. J., Darwen, H., 1992a. Into the great divide. In: Date, C. J., Darwen, H., Relational database writings 1989-1991, Addison Wesley
[8] Date, C. J., Darwen, H., 1992b. Relation-valued attributes or Will the real first normal form please stand up? In: Date, C. J., Darwen, H., Relational database writings 1989-1991, Addison Wesley
[9] Date, C. J., Darwen, H., 1992c. Toward a reconstituted definition of the relational model Version 1 (RM/V1), In: Date, C. J., Darwen, H., Relational database writings 1989-1991, Addison Wesley
[10] Date, C. J., Darwen, H., 1994. Divide and Conquer? In: Date, C. J., Darwen, H., Relational database writings 1991-1994, Addison Wesley
[11] Dix A., Finlay J., Abowd, G. D., Beale, R., 1994. Human-Computer Interaction, Prentice-Hall
[12] Elmasri, R., Navathe, S. B., 2003. Fundamentals of database systems, Addison Wesley
[13] Graefe, G., Cole, R. L., 1995. Fast algorithms for universal quantification in large databases, ACM Transactions on database systems, Vol. 20, No 2.
[14] Klug, A., 1982. Equivalence of relational algebra and relational calculus query langages having aggregate functions, Journal of the ACM, Vol 29, No. 3
[15] Libkin, L., 2003. Expressive power of SQL, Theoretical Computer Science 296
[16] McCann, L. I., 2003. On making relational division comprehensible, ASEE/IEEE Frontiers in education conference, 2003
[17] Moe, R. E., 2004, Context-sensitive relational division, WSEAS Transactions on information science and applications, Issue 1, Vol 1.
[18] Pigott, D., 2006, HOPL: an interactive Roster of Programming Languages, http://hopl.murdoch.edu.au/
[19] Rantzau, R., Shapiro, L. D., Mitschang, B., Wang, Q., 2003. Algorithms and applications for universal quantification in relational databases. Information Systems 28
[20] Reisner, P., 1981. Human factors studies of database query languages: a survey and assessment. Computer Surveys, Vol 13, No. 1
[21] Rosson, M. B., 1996. Human factors in programming and Software Development. ACM Computing Surveys, Vol 28, No. 1
[22] Trovåg, A., 2004. Beyond the divide, Master thesis, Department of information science and media studies, University of Bergen
[23] Turner, J. A., Jarke, M., Stohr, E. A, 1985. Coupling field studies with laboratory experiments for the evaluation of computer languages. Proceedings of the 1985 ACM Computer Conference
[24] Watt, D. A., 1991. Programming language syntax and semantics, Prentice Hall
[25] Welty, C., Stemple, D. W., 1981. Human factors comparison of a procedural and a nonprocedural query language, ACM Transactions on database systems, vol. 6. No. 4,
[26] White, G., 2004. The Philosophy of Computer Languages, In: Floridi, L. (ed), 2004. Philosophy of Computing and Information, Blackwell