Nested Queries in Object Bases

Sophie Cluet | *Guido Moerkotte*

INRIA
BP 105
Domaine de Voluceau
78153 Le Chesnay Cedex
France
Sophie.Cluet@inria.fr

Lehrstuhl für Informatik III
RWTH-Aachen
Ahornstr. 55
52074 Aachen
Germany
moer@gom.informatik.rwth-aachen.de

Abstract

Many declarative query languages for object-oriented (oo) databases allow nested subqueries. This paper contains (1) the first algebra which is capable of handling arbitrary nested queries and (2) the first complete classification of oo nested queries and the according unnesting strategies.

For unnesting, a two-phase approach is used. The first phase—called *dependency-based optimization*—transforms queries at the query language level in order to treat common subexpressions and independent subqueries more efficiently. The transformed queries are translated to nested algebraic expressions. These entail nested loop evaluation which may be very inefficient. Hence, the second phase *unnests* nested algebraic expressions to allow for more efficient evaluation.

The paper also discusses the differences between unnesting in the relational and unnesting in the oo context. Since the first phase is rather simple, the paper concentrates on the second phase.

1 Introduction

Many declarative query languages for object-oriented database management systems have been proposed in the last few years (e.g. [2, 3, 7, 6, 25, 20]). To express complex conditions, access nested structure, or produce nested results, an essential feature found in these languages is the nesting of queries, i.e., the embedding of a query into another query.

The optimization of object-oriented (oo) queries has been intensively studied using algebraic rewriting [4, 5, 10, 35, 37, 38] or rewriting of path expressions [5, 10, 19, 20]. However, in spite of the importance of nested queries [29], there exists only little research on their optimization [11, 29]. Translating nested queries into nested algebraic expressions requiring nested loop evaluation still seems to be the prevailing method.

To a lesser extent, relational query languages also feature nested queries and their optimization has been considered. Clearly, the optimization of nested queries in the oo context should make use of this body of knowledge. Departing from the evaluation of nested SQL queries through nested loops [1], Kim proposed unnesting of SQL queries at the SQL level [26]. The main motivation was a demonstrated substantial gain in efficiency due to unnesting. In order to eliminate some bugs which where subsequently detected, outer-joins were used [13, 17, 24, 16, 30]. Reordering of outer-joins became an important topic [13, 30, 32]. Lately, a unifying framework for different unnesting strategies was proposed in [31].

Nevertheless, the reader should notice that there exist some significant differences between oo and relational nested queries. First, the result of an oo nested query is not always flat. Second, in relational nested queries, the selection clause (e.g., **where** clause) plays a key and dual role: it is where the nesting is located[1] and where dependencies (i.e., references to variables of an outer block) may be expressed. In the oo context, all clauses are of equal importance and nesting may occur or dependency be expressed in any clause: projection (e.g., **select** clause), range (e.g., **from** clause) or selection. As we will see, this has a major impact on the unnesting possibilities. Third, nested queries in the oo context do not always correspond to algebraic operations (consider, e.g., method calls, path expressions), and an appropriate treatment for these nested expressions is needed as well. While the above considerations seem to complicate the issues, there also exists one essential difference that will somehow simplify the problem: set-valued attributes can be represented explicitly. In particular, a consequence is that the introduction of *null* values can be avoided in most cases.

As mentioned above, some relational techniques are easily adapted to the oo context. For instance, the whole idea of using joins and outer-joins for unnesting nested queries will be successfully applied throughout the paper. However, the differences between the two models lead us to consider new optimization techniques. We adopt the two-phase optimization approach of [11]. The first phase addresses a new challenge related to the third difference stated above. During this phase we apply *dependency-based optimization* which transforms queries by factoring out constant or locally constant nested queries as well as common subexpressions. Although not new in the oo context [10, 20, 23], this factorization is essential for finding a good evaluation strategy. Then, the resulting queries are translated in a straightforward manner into nested algebraic expressions.

The algebra we introduce is an extension of the GOM algebra [22, 23] and features some nice properties most oo algebras lack (e.g., associativity of join operations). The most important feature for us is that the algebra is capable of expressing all kinds of nesting occurring in object query languages. This new feature of an algebra allows us to perform unnesting at the algebraic rather than at the source level. This is important since it is our believe that the errors encountered in unnesting in the relational context are due to the fact that unnesting is done at the source level.

The second phase—called *algebraic optimization*—exploits new opportunities brought by the possibility to represent non-atomic attributes. More specifically, the applied algebraic equivalences will make extensive use of two powerful grouping operations, one of them introduced in [11]. Although the problems implied by the first and second differences stated above are solved by a combination of the two phases, we here concentrate on

[1]To a limited extent, relational nested queries can also be found in range clauses: when views are queried. In these cases, the inner query is always constant.

a more complete treatment of nested algebraic expressions within the second phase. For the first phase, the reader is referred to [11].

The paper is organized as follows. The next section introduces the algebra. Further, it gives some basic algebraic equivalences holding within the algebra. These should demonstrate the viability of the algebra for optimization purposes since algebraic equivalences are at the core of any optimization process. Section 3 contains a short review of the first phase by means of an example. The core of the paper is contained within Section 4 where we use an extension of Kim's classification [26], to introduce the according algebraic equivalences necessary to unnest nested algebraic expressions. Here, we repeat some of the results of [11] and introduce new unnesting techniques for those cases which could not be unnested so far. This section is concerned with unnesting of one level nested blocks. In Section 5 we go further in the optimization by considering quantifiers, unnesting d-joins, complex cases of multi-layered nested queries as well as outer restrictions. Section 6 introduces a simple but efficient method to handle outer restrictions. Section 7 concludes the paper.

2 The Algebra

The Data Model Our underlying data model is similar to the O_2[14] or GOM [21] model. It features objects that have an identity, that are manipulated through user-defined methods, whose structures are complex and that belong to classes that may be refined into subclasses. Each class has an extension which is a set containing the object identifiers of all its instances. The model also features complex values with no identity, that are manipulated by standard operators and do not belong to classes. Hence, there are no extensions for them.

General Remarks on the Algebra The algebra is an extension of the GOM algebra [22, 23]. The most predominent features of the algebra we introduce are:

- All operators are polymorphic and can deal with complex input arguments.
- The operators take arbitrary complex expressions as subscripts. This includes algebraic expressions.
- The algebra is redundant since some special cases of the operators can be implemented much more efficiently. In some of these cases, the special cases are introduced as abbreviations.

For the purpose of the paper, the main feature of this algebra is that it can cope with arbitrary nested queries.

The remainder of this section is organized as follows. The next subsection introduces the operators of the algebra. While some of the operators are easy extensions of standard operators, others are quite new. This forces us to reconsider existing algebraic equivalences as well as to discover new ones. The main question here is reorderability of operators since reorderability is fundamental to any optimization process. Hence, the remaining subsections discuss reorderability results. Thereby, we discover general reorderability laws in subsection 2.2 for all operators which are linear. The reorderability features of operators that are not linear (d-join and grouping) are discussed subsection 2.3. Since the join remains the most expensive operator in our algebra, subsection 2.4 discusses simplifications of expressions containing joins.

Summarizing, this section introduces a powerful algebra capable of capturing all commonly used oo query languages including excessive nesting. Further, the basics for building an optimizer based on this algebra are introduced in the form of algebraic equivalences. Only after this section, we concentrate on the very specific issue of unnesting nested queries.

2.1 The Operators

The core of the algebra consists of the following operators that are all defined on set values: union ($\cup$), intersection ($\cap$), difference ($\setminus$), selection (σ), join($\bowtie$), semi-join ($\rhd\!<$), anti-join ($\rhd$), left-outer join ($\sqsupset\!\bowtie$), d-join ($< \cdot >$), mapping (χ), unnest (μ), and grouping (Γ).

The set operators as well as the selection operator and the different join operators (except for the d-join) are known from the relational context. As we will see, the only difference in their definition is that they are tuned to handle nested queries. For this, we allow the subscripts of these operators to contain full algebraic expressions. Further, to adjust them to the oo context, they do not only deal with relations, but with sets of tuples where the tuples can be of arbitrary complexity. This means, that the attribute values are in no way restricted to atomic types but can carry also objects, sets, lists and so on.

The left-outer join additionally needs some tuning in order to exploit the possibility to have sets as attribute values. For this, it carries a superscript giving a default value for some attribute for those tuples in the left argument for which there is no matching tuple in the right argument. The d-join operation is used for performing a join between two sets, the second one being dependent on the first. It is left-associative and will be applied in post order. This operator can be used for unnesting and is in many cases equivalent to a join between two sets with a membership predicate [35]. In some cases (as we will see later on), it corresponds to an unnest operation. We introduced the d-join in order to cope with the values of types that do not have extensions (i.e. there exist no sets on which a join could be applied).

The mapping operator χ ([22]) is well-known from the functional programming language context. A special case of it, where it adds derived information in form of an added attribute with an according value (by object base lookup or by method calls) to each tuple of a set has been proposed in [20, 22]. Lately, this special case was given the name *materialization operator* [5].

The unnest operator is known from NF^2 [34, 33]. It will come in two different flavors allowing us to perform unnesting not only on nested relations but also on attributes whose value is a set of elements which are not tuples. The last operator—the *grouping* operator—generalizes the nest operator quite a bit. That is why we renamed it. In fact, there exist two grouping operators, one unary grouping operator and one binary grouping operator. The unary grouping operator groups one set of tuples according to a grouping condition. Further, it can apply an arbitrary expression to the newly formed group. This allows for efficient implementation by saving on intermediate results. The binary grouping operator adds a group to each element in the first argument set. This group is formed from the second argument. The grouping operator will exploit the fact that in the oo context attributes can have set-valued attributes. As we will see, this is useful for both, unnesting nested queries and producing nested results.

Preliminaries As already mentioned, our algebraic operators can deal not only with standard relations but are polymorphic in the general sense. In order to fix the domain of the operators we need some technical abbreviations and notations. Let us introduce these first.

Since our operators are polymorphic, we need variables for types. We use τ possibly with a subscript to denote types. To express that a certain expression is of type e, we write $e :: \tau$. Starting from concrete names for types and type variables, we can build type expressions the standard way by using type constructors to build tuple tupes ($[\cdot]$), set types $\{\cdot\}$, and list types ($< \cdot >$). Having two type expressions e_1 and e_2, we denote by $e_1 \leq e_2$, that e_1 is a subtype of e_2. It is important to note that this subtype relationship is not based on the sub-/superclass hierarchy found in most oo models. Instead, it denotes that the type(s) e_1 stands for provide at least all the features that those of e_2 provide.

Most of our algebraic operators are tuned to work on sets of tuples. The most important information here is the set of attributes provided by a tuple or a set of tuples. For this we introduce $\mathcal{A}$. The function $\mathcal{A}$ is defined as follows. $\mathcal{A}(e) = \{a_1, \ldots, a_n\}$ if $e :: \{[a_1 : \tau_1, \ldots, a_n : \tau_n]\}$ or $e :: [a_1 : \tau_1, \ldots, a_n : \tau_n]$. Giving a set of attributes A, we are sometimes interested in the attributes provided by an expression e which are not in A. For this complement we use the notation $\overline{A}(e)$ defined as $\mathcal{A}(e) \setminus A$. When e is clear from the context, we use $\overline{A}$ as a shorthand.

Often, we are not only interested in the set of attributes an expression provides, but also in the set of free variables occurring in an expression e. For this, we introduce $\mathcal{F}(e)$ denoting the set of all free variables of e.

Since the subscripts of our algebraic operators can contain arbitrary expressions, they may contain variables or even free variables. Then, there is a need to get bindings for these variables before the subscript expression can be evaluated. These bindings are taken from the argument(s) of the operator. In order to do so, we need a specified binding mechanism. The λ-notation is such a mechanism and can be used e.g., in case of ambiguities. However, for the purpose of the paper, it suffices if we stick to the following convention.

- For an expression e with free variables $\mathcal{F}(e) = \{a_1, \ldots, a_n\}$ and a tuple t with $\mathcal{F}(e) \subseteq \mathcal{A}(t)$ we define $e(t) = e[a_1 \leftarrow t.a_1, \ldots, a_n \leftarrow t.a_n]$.[2] Similarily, we define $e(t_1, t_2)$ for binary operations. Note that the attribute names of t_1 and t_2 should be distinct in order to avoid name conflicts.

- For an expression e with only one free variable x, we define $e(t) = e[x \leftarrow t]$.

[2] $e[v_1 \leftarrow e_1, \ldots, v_n \leftarrow e_n]$ denotes a substitution of the variables v_i by the expressions e_i within an expression e.

The mechanism is very much like the standard binding for the relational algebra. Consider for example a select operation $\sigma_{a=3}(R)$, then we assume that a, the free variable of the subscript expression $a = 3$ is bound to the value of the attribute a of the tuples of the relation R. To express this binding explicitly, we would write for a tuple $t \in R$ $(a = 3)(t)$. Since a is an attribute of R and hence of t, by our convention a is replaced by $t.a$, the value of attribute a of tuple t. Since we want to avoid name conflicts right away, we assume that all variable/attribute names used in a query are distinct. This can be achieved in a renaming step.

Application of a function f to arguments e_i is denoted by either regular (e.g., $f(e_1, \ldots, e_n)$) or dot (e.g., $e_1.f(e_2, \ldots, e_n)$) notation. The dot notation is used for type associated methods, only.

Often, conjunctions of predicates occur. As an abbreviation for a conjunction of predicates of the form $a_1\theta b_1, \ldots, a_n\theta b_n$ $(x.a_1\theta_1 y.b_1, \ldots, x.a_n\theta_n y.b_n)$ we often use $A\theta B$ $(x.[A]\theta y.[B])$ if $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_n\}$.

Last, we introduce the heavily overloaded symbol $\circ$. It denotes function concatenation and (as a special case) tuple concatenation as well as the concatenation of tuple types to yield a tuple type containing the union of the attributes of the two argument tuple types.

Operator Signatures We are now ready to define the signatures of the operators of our algebra. Their semantics is defined in a subsequent step. Remember that we consider all operators as being polymorphic. Hence, their signatures are polymorphic and contain type variables.

$$
\begin{array}{rcl}
\cup & : & \{\tau\}, \{\tau\} \rightarrow \{\tau\} \\
\cap & : & \{\tau\}, \{\tau\} \rightarrow \{\tau\} \\
\setminus & : & \{\tau\}, \{\tau\} \rightarrow \{\tau\} \\
\sigma_p & : & \{\tau\} \rightarrow \{\tau\} \\
 & & \text{if } p : \tau \rightarrow \mathcal{B} \\
\bowtie_p & : & \{\tau_1\}, \{\tau_2\} \rightarrow \{\tau_1 \circ \tau_2\} \\
 & & \text{if } \tau_i \leq [], \ p : \tau_1, \tau_2 \rightarrow \mathcal{B}, \text{ and} \\
 & & \mathcal{A}(\tau_1) \cap \mathcal{A}(\tau_2) = \emptyset \\
\rhd\!\!<_p & : & \{\tau_1\}, \{\tau_2\} \rightarrow \{\tau_1\} \\
 & & \text{if } \tau_i \leq [], \ p : \tau_1, \tau_2 \rightarrow \mathcal{B} \\
\rhd_p & : & \{\tau_1\}, \{\tau_2\} \rightarrow \{\tau_1\} \\
 & & \text{if } \tau_i \leq [], \ p : \tau_1, \tau_2 \rightarrow \mathcal{B} \\
\sqsupset\!\!\bowtie_p^{g=c} & : & \{\tau_1\}, \{\tau_2\} \rightarrow \{\tau_1 \circ \tau_2^+\} \\
 & & \text{if } \tau_1 < [], \ c :: \tau, \ \tau_2 < [g : \tau], \\
 & & + \text{ just adds a null value to the domain of } \tau_2, \\
 & & \text{if it does not already contain one,} \\
 & & p : \tau_1, \tau_2 \rightarrow \mathcal{B} \\
< \cdot > & : & \{\tau_1\} || \{\tau_2\} \rightarrow \{\tau_1 \circ \tau_2\} \\
 & & \text{if } \tau_i \leq [],
\end{array}
$$

$$
\begin{array}{rcl}
 & & \mathcal{A}(\tau_1) \cap \mathcal{A}(\tau_2) = \emptyset \\
\chi_f & : & \{\tau_1\} \rightarrow \{\tau_2\} \\
 & & \text{if } f : \tau_1 \rightarrow \tau_2 \\
\Gamma_{g;\theta A;f} & : & \{\tau\} \rightarrow \{\tau \circ [g : \tau']\} \\
 & & \text{if } \tau \leq [], f : \{\tau\} \rightarrow \tau' \\
\Gamma_{g;\theta A;f} & : & \{\tau_1\}, \{\tau_2\} \rightarrow \{\tau_1 \circ [g : \tau']\} \\
 & & \text{if } \tau_1 \leq [], f : \{\tau_2\} \rightarrow \tau' \\
\mu_g & : & \{\tau\} \rightarrow \{\tau'\} \\
 & & \text{if } \tau = [a_1 : \tau_1, \ldots, a_n : \tau_n, g : \{\tau_0\}], \\
 & & \tau_0 \leq [], \\
 & & \tau' = [a_1 : \tau_1, \ldots, a_n : \tau_n] \circ \tau_0, \\
\mu_{g;c} & : & \{\tau\} \rightarrow \{\tau'\} \\
 & & \text{if } \tau = [a_1 : \tau_1, \ldots, a_n : \tau_n, g : \{\tau_0\}], \\
 & & \tau_0 \not\leq [], \\
 & & \tau' = [a_1 : \tau_1, \ldots, a_n : \tau_n] \circ [c : \tau_0], \\
flatten & : & \{\{\tau\}\} \rightarrow \{\tau\} \\
max_{g;m;a\theta;f} & : & \{\tau\} \rightarrow [m : \tau_a, g : \tau_f] \\
 & & \text{if } \tau \leq [a : \tau_a], f : \{\tau\} \rightarrow \tau_f
\end{array}
$$

The semantics of the set operators is standard and omitted here. We start with the selection and then go through all the operators thereby commenting on them. In the subsequent definitions, the e_i's denote both expressions (in the left hand side) and their evaluation (in the right hand side).

Selection Operator Note that in the following definition there is no restriction on the selection predicate. It may contain path expressions, method calls, nested algebraic operators, etc.

$$\sigma_p(e) \quad = \quad \{x | x \in e, p(x)\}$$

Join Operators The algebra features five join operators. Besides the complex join predicate, the first four of them are rather standard: join, semi-join, anti-join and left-outer join are defined similarly to their relational counterparts. One difference is that the left-outer join accepts a default value to be given, instead of null, to one attribute of its right argument.

$$
\begin{array}{rcl}
e_1 \bowtie_p e_2 & = & \{y \circ x | y \in e_1, x \in e_2, p(y, x)\} \\
e_1 \rhd\!\!<_p e_2 & = & \{y | y \in e_1, \exists x \in e_2, (p(y, x))\} \\
e_1 \rhd_p e_2 & = & \{y | y \in e_1, \neg\exists x \in e_2 \; p(y, x)\} \\
e_1 ⟕_p^{g=c} e_2 & = & \{y \circ x | y \in e_1, x \in e_2 p(y, x)\} \cup \\
 & & \{y \circ z | y \in e_1, \neg\exists x \in e_2 \; p(y, x), \mathcal{A}(z) = \mathcal{A}(e_2), g \in \mathcal{A}(e_2), \\
 & & z.g = c, \forall a \in \mathcal{A}(e_2)(z \neq g \Longrightarrow z.a = NULL)\}
\end{array}
$$

Remember that the function $\mathcal{A}$ used in the last definition returns the set of attributes of a relation.

The last join operator is called *d-join* ($< \cdot >$). It is a join between two sets, where the evaluation of the second set may dependent on the first set. It is used to translate **from** clauses into the algebra. Here, the range definition of a variable may depend on the value of a formerly defined variable. Whenever possible, d-joins are rewritten into standard joins.

$$e_1 < e_2 > \quad = \quad \{y \circ x | y \in e_1, x \in e_2(y)\}$$

Map Operator (and Projection, Renaming) These operators are fundamental to the algebra. As the operators mainly work on sets of tuples, sets of non-tuples (mostly sets of objects) must be transformed into sets of tuples. This is one purpose of the map operator. Other purposes are dereferenciation, method and function application. Our translation process also pushes all nesting into map operators.

The first definition corresponds to the standard map as defined in, e.g., [22]. The second definition concerns the special case of a materialize operator [20, 5]. The third definition handles the frequent case of constructing a set of tuples with a single attribute out of a given set of (non-tuple) values.

$$\begin{aligned} \chi_{e_2}(e_1) &= \{e_2(x) | x \in e_1\} \\ \chi_{a:e_2}(e_1) &= \{y \circ [a : e_2(y)] | y \in e_1\} \\ e[a] &= \{[a : x] | x \in e\} \end{aligned}$$

Note that the oo map operator obviates the need of a relational projection. Sometimes the map operator is equivalent to a simple projection (or renaming). In these cases, we will use π (or ρ) instead of χ.

Grouping Operators Two grouping operators will be used for unnesting purposes. The first one — called *unary grouping* — is defined on a set and its subscript indicates (i) the attribute receiving the grouped elements (ii) the grouping criterion, and (iii) a function that will be applied to each group.

$$\begin{aligned} \Gamma_{g;A\theta;f}(e) = \; & \{y.A \circ [g : G] | y \in e, \\ & G = f(\{x | x \in e, x.A\theta y.A\})\} \end{aligned}$$

Note that the traditional nest operator [34] is a special case of unary grouping. It is equivalent to $\Gamma_{g;A=;id}$. Note also that the grouping criterion may be defined on several attributes. Then, A and θ represent sequences of attributes and comparators.

The second grouping operator — called *binary grouping* — is defined on two sets. The elements of the second set are grouped according to a criterion that depends on the elements of the first argument.

$$e_1 \Gamma_{g;A_1\theta A_2;f} e_2 \quad = \quad \{y \circ [g : G] | y \in e_1, G = f(\{x | x \in e_2, y.A_1 \theta x.A_2\})\}$$

In the sequel, the following abbreviations will be used: $\Gamma_{g;A;f}$ for $\Gamma_{g;A=;f}$, $\Gamma_{g;A}$ for $\Gamma_{g;A;id}$.

New implementation techniques have to be developed for these grouping operators. Obviously, those used for the nest operator can be used for simple grouping when θ stands for equality. For the other cases, implementations based on sorting seem promising. We also consider adapting algorithms for non-equi joins, e.g. those developed for the band-width join [15]. A very promising approach is the use of θ-tables developed for efficient aggregate processing [12].

Unnest Operators The unnest operators come in two different flavor. The first one is responsible for unnesting a set of tuples on an attribute being a set of tuples itself. The second one unnests sets of tuples on an attribute not being a set of tuples but a set of something else, e.g., integers.

$$\begin{aligned}\mu_g(e) &= \{y.[\mathcal{A}(y) \setminus \{g\}] \circ x | y \in e, x \in y.g\}\\ \mu_{g;c}(e) &= \{y.[\mathcal{A}(y) \setminus \{g\}] \circ [c : x] | y \in e, x \in y.g\}\end{aligned}$$

Flatten Operator The flatten operator flattens a set of sets by unioning the elements of the sets contained in the outer set.

$$flatten(e) = \{y | x \in e, y \in x\}$$

Max Operator The Max operator has a very specific use that will be explained in the sequel. Note that an analogous Min operator can be defined.

$$Max_{g;m;a\theta;f}(e) = [m : max(\{x.a | x \in e\}), g : f(\{x | x \in e, x.a\theta m\})]$$

This definition is a generalization of the Max operator as defined in [11]. Since the equivalences don't care whether we use Max or Min, we write Agg to denote either of them.

Remarks Note that, apart from the χ and $flatten$ operations, all these operations are defined on sets of tuples. This guarantees some nice properties among which is the associativity of the join operations. Note also that the operators may take complex expressions in their subscript, therefore allowing nested algebraic expressions. This is the most fundamental feature of the algebra when it comes to express nested queries at the algebraic level. Unnesting is then expressed by algebraic equivalences moving algebraic expression out of the superscript.

The Γ, $flatten$ and Max operations are mainly needed for optimization purposes, as we will see in the sequel, but do not add power to the algebra. Note that a Min operation similar to the Max operation can easily be defined.

The algebra is defined on sets whereas most OBMS also manipulate lists and bags. We believe that our approach can easily be extended by considering lists as set of tuples with an added positional attribute and bags as sets of tuples with an added key attribute.

2.2 Linearity and Reorderability

2.2.1 Linearity of Algebraic Operators

As already mentioned, the core argument for optimizability of algebraic expressions is the reorderability of their operators. One could discuss the reorderability of each two operators resulting in n^2 investigations if the number of operators in the algebra is n. In order to avoid this, we introduce a general argument covering most of the cases. This argument is that linearity of operators implies their reorderability easily. Hence, let us first look at the linearity property.

A unary mapping $f : \{\tau\} \rightarrow \{\tau'\}$ is called *linear*, iff

$$
\begin{aligned}
f(\emptyset) &= \emptyset \\
f(A \cup B) &= f(A) \cup f(B)
\end{aligned}
$$

An n-ary mapping

$$f : \tau_1, \ldots, \tau_{i-1}, \{\tau\}, \tau_{i+1}, \ldots, \tau_n \rightarrow \{\tau'\}$$

is called *linear in its i-th argument*, iff for all $e_1, \ldots, e_n, e'_i$

$$
\begin{aligned}
f(e_1, \ldots, e_{i-1}, \emptyset, e_{i+1}, \ldots, e_n) &= \emptyset \\
f(e_1, \ldots, e_{i-1}, e_i \cup e'_i, e_{i+1}, \ldots, e_n) &= f(e_1, \ldots, e_{i-1}, e_i, e_{i+1}, \ldots, e_n) \\
&\quad \cup f(e_1, \ldots, e_{i-1}, e'_i, e_{i+1}, \ldots, e_n)
\end{aligned}
$$

It is called *linear*, if it is linear in all its arguments. Note that if an equivalence with linear mappings on both sides has to be proven, it suffices to proof it for disjunct singletons, i.e. sets with one element only.

The following summarizes the findings on the linearity of the algebraic operators:

χ_f is linear.

$$
\begin{aligned}
\chi_f(\emptyset) &= \emptyset \\
\chi_f(e_1 \cup e_2) &= \{f(x) | x \in e_1 \cup e_2\} \\
&= \{f(x) | x \in e_1\} \cup \{f(x) | x \in e_2\} \\
&= \chi_f(e_1) \cup \chi_f(e_2)
\end{aligned}
$$

σ is linear (for a proof see [39])

$\bowtie_p$ is linear (for a proof see [39]). Similarly, $\triangleright\!<$ is linear. $\triangleright$ is linear in its first argument but obviously not in its second.

$\sqsupset\!\!\bowtie$ is linear in its first argument.

$$\begin{aligned}
\emptyset \sqsupset\!\!\bowtie_p^{g=c} e &= \emptyset \\
(e_1 \cup e_2) \sqsupset\!\!\bowtie_p^{g=c} e &= \{y \circ x | y \in e_1 \cup e_2, x \in e, p(y,x)\} \cup \\
&\quad \{y \circ z | y \in e_1 \cup e_2, \neg \exists x \in e \; p(y,x), \mathcal{A}(z) = \mathcal{A}(e), \\
&\quad \forall a \; a \neq g \succ z.a = NULL, a.g = c\} \\
&= (e_1 \sqsupset\!\!\bowtie_p^{g=c} e) \cup (e_2 \sqsupset\!\!\bowtie_p^{g=c} e)
\end{aligned}$$

To see that $\sqsupset\!\!\bowtie$ is not linear in its second argument consider

$$e_1 \sqsupset\!\!\bowtie_p^{g=c} \emptyset \quad = \quad \emptyset \text{ iff } e_1 = \emptyset$$

$< >$ is linear in its first argument:

$$\begin{aligned}
\emptyset < e > &= \emptyset \\
(e_1 \cup e_2) < e > &= \{y \circ x | y \in e_1 \cup e_2, x \in e(y)\} \\
&= \{y \circ x | y \in e_1, x \in e(y)\} \cup \{y \circ x | y \in e_2, x \in e(y)\} \\
&= (e_1 < e >) \cup (e_2 < e >)
\end{aligned}$$

Note that the notion of linearity cannot be applied to the second (inner) argument of the d-join, since, in general, it cannot be evaluated independently of the first argument. Below, we summarize some basic observations on the d-join.

$\Gamma_{g;A;f}$ is not linear.

Consider the following counterexample:

$$\begin{aligned}
&\Gamma_{g;a}(\{[a:1,b:1],[a:1,b:2]\}) \\
&\quad = \{[a:1, g:\{[a:1,b:1],[a:1,b:2]\}]\} \\
&\quad \neq \{[a:1, g:\{[a:1,b:1]\}]\} \cup \{[a:1, g:\{[a:1,b:2]\}]\} \\
&\quad = \Gamma_{g;a}(\{[a:1,b:1]\}) \cup \Gamma_{g;a}(\{[a:1,b:2]\})
\end{aligned}$$

μ_g is linear.

$$\begin{aligned}
\mu_g(\emptyset) &= \emptyset \\
\mu_g(e_1 \cup e_2) &= \{x.[\overline{g}] \circ y | x \in e_1 \cup e_2, y \in x.g\} \\
&= \{x.[\overline{g}] \circ y | x \in e_1, y \in x.g\} \cup \{x.[\overline{g}] \circ y | x \in e_2, y \in x.g\} \\
&= \mu_g(e_1) \cup \mu_g(e_2)
\end{aligned}$$

$\mu_{g;c}$ is also linear. This is shown analogously to the linearity of μ_g.

$flatten$ is linear.

$$\begin{aligned}
&flatten(e_1 \cup e_2) \\
&\quad = \{x | y \in e_1 \cup e_2, x \in y\} \\
&\quad = \{x | y \in e_1, x \in y\} \cup \{x | y \in e_2, x \in y\} \\
&\quad = flatten(e_1) \cup flatten(e_2)
\end{aligned}$$

Note that the notion of linearity does not apply to the *max* operator, since it does not return a set.

The concatenation of linear mappings is again a linear mapping. Assume f and g to be linear mappings. Then

$$\begin{aligned} f(g(\emptyset)) &= \emptyset \\ f(g(x \cup y)) &= f(g(x) \cup g(y)) \\ &= f(g(x)) \cup f(g(y)) \end{aligned}$$

2.2.2 Reorderability Laws

From the linearity considerations of the previous section, it is easy to derive reorderability laws.

Let $f : \{\tau_1^f\} \rightarrow \{\tau_2^f\}$ and $g : \{\tau_1^g\} \rightarrow \{\tau_2^g\}$ be two linear mappings. If $f(g(\{x\})) = g(f(\{x\}))$ for all singletons $\{x\}$ then

$$f(g(e)) \quad = \quad g(f(e)) \tag{1}$$

For the linear algebraic operators working on sets of tuples, we can replace the semantic condition $f(g(\{x\})) = g(f(\{x\}))$ by a set of syntactic criterions. The main issue here is to formalize that two operations do not interfere in their consumer/producer/modifier relationship on attributes. In the relational algebra we have the same problem. Nevertheless, it is often neglected there. Consider for example the algebraic equivalence

$$\sigma_p(R \bowtie S) \quad = \quad (\sigma_p(R)) \bowtie S$$

Then, this algebraic equivalence is true only if the predicate p accesses only those attributes already present in R. Now, for our operators we can be sure that $f(g(e)) = g(f(e))$ for singleton sets e, if g does not consume/produce/modify an attribute that f is going to access and if f is not going to consume/produce/modify an attribute that is accessed by g. In fact, most of the conditions attached to the algebraic equivalences given later on concern this point.

We now consider binary mappings. Let f_1 be a binary mapping being linear in its first argument, f_2 a binary mapping being linear in its second argument, and g a unary linear mapping. If $f_1(g(\{x\}), e') = g(f_1(\{x\}, e'))$ for all x and e' then

$$f_1(g(e), e') \quad = \quad g(f_1(e, e')) \tag{2}$$

for all e. Again, we can recast the condition $f_1(g(\{x\}), e') = g(f_1(\{x\}), e')$ into a syntactical criterion concerning the consumer/producer/modifier relationship of attributes.

Analogously, if $f_2(e, g(\{x\})) = g(f_2(e, \{x\}))$ for all x and e then

$$f_1(e, g(e')) \quad = \quad g(f_1(e, e')) \tag{3}$$

for all e'.

Since the outerjoin is not linear in its second argument, we state at least some reorderability results concerning the reordering of joins with outerjoins since much performance can be gained by chosing a (near-) optimal evaluation order. The results are not original but instead taken from [32] and repeated here for convenience:

$$
\begin{aligned}
(e_1 \bowtie_{p_{1,2}} e_2) \sqsupset\!\!\bowtie_{p_{2,3}} e_3 &= e_1 \bowtie_{p_{1,2}} (e_2 \sqsupset\!\!\bowtie_{p_{2,3}} e_3) && (4)\\
(e_1 \sqsupset\!\!\bowtie_{p_{1,2}} e_2) \sqsupset\!\!\bowtie_{p_{2,3}} e_3 &= e_1 \sqsupset\!\!\bowtie_{p_{1,2}} (e_2 \sqsupset\!\!\bowtie_{p_{2,3}} e_3) && (5)\\
&\quad \text{if } p_{2,3} \text{ is strong w.r.t. } e_2 &&\\
(e_1 \sqsupset\!\!\bowtie_{p_{1,2}} e_2) \bowtie\!\!\sqsubset_{p_{2,3}} e_3 &= e_1 \sqsupset\!\!\bowtie_{p_{1,2}} (e_2 \bowtie\!\!\sqsubset_{p_{2,3}} e_3) && (6)
\end{aligned}
$$

where an outer join predicate p is strong w.r.t. some expression e_2, if it yields false if all attributes of the relation to be preserved are NULL.

2.3 Basic Equivalences for d-Join and Grouping

The d-join and the grouping operators are not linear. Thus, so far, we do not have any reorderability results of these operators. Since they are further quite new, we give some algebraic equivalences which hold despite the fact that they are not linear. This shows that there exist still some kind of optimization which can be performed in the presence of these operators.

Already at the beginning of this section, we mentioned that d-join and unnest are closely related. To be more precise, we state the following:

$$
\begin{aligned}
e_1 < e_2 > &= \mu_g(\chi_{g:e_2}(e_1)) && (7)\\
\pi_{\mathcal{A}(e_2)} e_1 < e_2 > &= \mu_g(\chi_{[g:e_2]}(e_1)) && (8)
\end{aligned}
$$

Between *flatten* and the d-join there also exists a correspondance:

$$
flatten(\chi_{e_2}(e_1)) = \pi_{\mathcal{A}(e_2)}(e_1 < e_2 >) \qquad (9)
$$

The following summarizes basic equivalences on the d-join:

$$
\begin{aligned}
e < e > &= e && (10)\\
e_1 < e_2 > &= e_1 \times e_2 && (11)\\
&\quad \text{if } \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset &&\\
e_1 < e_2 > &= e_1 \bowtie (e_1 < e_2 >) && (12)\\
e_1 < e_2 >< e_3 > &= e_1 < e_3 >< e_2 > && (13)\\
&\quad \text{if } (\mathcal{A}(e_2) \setminus \mathcal{F}(e_2)) \cap \mathcal{F}(e_3) = \emptyset &&\\
\pi_{\mathcal{A}(e_1)}(e_1 < e_2 >) &= \sigma_{e_2 != \emptyset}(e_1) && (14)\\
e_1 < e_2 > &= e_1 \bowtie_{A_1 \theta A_2} (e_2' < e_2 >) && (15)\\
&\quad \text{if } \mathcal{F}(e_2) = \{a_1, \ldots, a_n\} \text{ and} &&\\
&\quad e_2 = \sigma_{A_1 \theta A_2}(e_2'))) \text{ and} &&\\
&\quad A_i \subseteq \mathcal{A}(e_i) &&
\end{aligned}
$$

Unnesting of operations burried in the d-join can be performed by applying the following equivalence:

$$
e_1 < \sigma_p(e_2) > = \sigma_p(e_1 < e_2 >) \qquad (16)
$$

$$
\begin{aligned}
e_1 < \sigma_{A_1 \theta A_2}(e_2) > &= e_1 \Join_{A_1 \theta A_2} e_2 && (17)\\
& \quad \text{if } \mathcal{F}(e_2) \cap \mathcal{P}(e_1) = \emptyset,\, A_i \subseteq \mathcal{A}(e_i) \\
\pi_{A':A}(e) < f(\sigma_{A=A'}(e)) > &= \mu_g(\pi_{A:A'}(\Gamma_{g;A;f}(e))) && (18)\\
e_1 < e_2 \Join e_3 > &= (e_1 < e_2 >) \Join e_3 && (19)\\
e_1 < \chi_f(e_2) > &= \chi_f(e_1 < e_2 >) && (20)\\
& \quad \text{if } f \text{ extending} \\
\pi_A(e_1 < \chi_f(e_2) >) &= \pi_A(\chi_f(e_1 < e_2 >)) && (21)\\
& \quad \text{if } A \subseteq \mathcal{A}(\chi_f(e_2)), \text{ and } f \text{ restricting} \\
e_1 < \mu_g(e_2) > &= \mu_g(e_1 < e_2 >) && (22)\\
e_1 < \mu_{g;c}(e_2) > &= \mu_{g;c}(e_1 < e_2 >) && (23)
\end{aligned}
$$

For a linear f_1 and the non-linear unary Γ we still have

$$
\begin{aligned}
f_1(\Gamma_{g;\theta A;f_2}(e) &= \Gamma_{g;\theta A;f_2}(f_1(e)) && (24)\\
& \quad \text{if } \mathcal{F}(f_1) \cap (A \cup \mathcal{A}(f_2) \cup \{g\}) = \emptyset,\ (A \cup \mathcal{F}(f_2)) \subseteq \mathcal{A}(f_1(e))
\end{aligned}
$$

This equivalence should, e.g., be used form left to right for selections, in order to reduce the cardinality for the Γ operator. For the binary Γ we have

$$
f_1(e_1 \Gamma_{g;A_1\theta A_2;f} e_2) = f_1(e_1)\Gamma_{g;A_1\theta A_2;f} e_2) \qquad (25)
$$

since the binary Γ is linear in its first argument.

Lately, work has been reported on the reordering of grouping and join operations despite the fact that grouping is not linear [9, 42, 41, 43]. Since pushing grouping inside join operations can result in a tremendious speed up, let us sketch at least the most basic equivalence:

$$
\Gamma_{g;A;agg(e)}(e_1 \Join e_2) \equiv e_1 \Join (\Gamma_{g;A;agg(e)}(e_2))
$$

This sketched equivalence only holds under certain conditions. For details on the conditions and the correctness proof see [43].

2.4 Simplifying Expressions Containing Joins

Since the join operation is very expensive, it makes sense to investigate expressions containing joins very intensely in order to discover optimization potential. In this subsection, we do so.

Sometimes, redundant joins can be eliminated:

$$
\begin{aligned}
\pi_{\mathcal{A}(e_2)}(e_1 \Join_{A_1=A_2} e_2) &= e_2 && (26)\\
& \quad \text{if } A_1 = \mathcal{A}(e_1),\ \pi_{A_2}(e_2) \subseteq \pi_{A_2:A_1}(e_1) \\
e_1 \sqsupset\!\!\Join^{g:c}_{A_1=A_{2,3}}(e_2 \Join_{A_2=A_3} e_3) &= e_1 \sqsupset\!\!\Join^{g:c}_{A_1=A_3} e_3 && (27)\\
& \quad \text{if } A_1 \subseteq \mathcal{A}(e_1),\ A_2 \subseteq \mathcal{A}(e_2),\ A_3 \subseteq \mathcal{A}(e_3), \\
& \quad\ A_{2,3} \subseteq \mathcal{A}(e_2) \cup \mathcal{A}(e_3),\ A_2' \subseteq \mathcal{A}(e_1) \cup \mathcal{A}(e_3),
\end{aligned}
$$

$$
\begin{aligned}
e_1 \Gamma_{g;A_1=A_{2,3};f}(e_2 \bowtie_{A_2=A_3} e_3) \;=\; & \pi_{A_2:A_2'}(e_1 \bowtie_{A_1=A_3} e_3) \subseteq e_2 \\
& e_1 \Gamma_{g;A_1=A_2;f} e_3 \qquad (28) \\
& \text{if } A_1 \subseteq \mathcal{A}(e_1),\ A_2 \subseteq \mathcal{A}(e_2),\ A_3 \subseteq \mathcal{A}(e_3), \\
& A_{2,3} \subseteq \mathcal{A}(e_2) \cup \mathcal{A}(e_3),\ A_2' \subseteq \mathcal{A}(e_1) \cup \mathcal{A}(e_3), \\
& \pi_{A_2:A_2'}(e_1 \bowtie_{A_1=A_3} e_3) \subseteq e_2
\end{aligned}
$$

Equivalences 27 and 28 bear similarity to equivalence EJA of [39] (p. 107), where the outer-aggregation is used instead of semi-join and binary grouping, respectively. Note that for checking conditions of the form $\pi_{A_2}(e_2) \subseteq \pi_{A_2:A_1}(e_1)$ subtyping implying subsetrelationships on type extensions plays a major role in object bases.

The following shows how to turn an outerjoin into a join:

$$
\begin{aligned}
e_1 ⟕_p^{g:c} e_2 &= e_1 \bowtie_p e_2 \qquad (29) \\
& \quad \text{if } \neg p(c) \\
\sigma_{p_s}(e_1 ⟕_{p_j}^{g:c} e_2) &= \sigma_{p_s}(e_1 \bowtie_{p_j} e_2) \qquad (30) \\
& \quad \text{if } \neg p_s(c) \\
\sigma_{f_1 \theta f_2}(e_1 ⟕_{p_j}^{g:c} e_2) &= \sigma_{f_1 \theta f_2}(e_1 \bowtie_{p_j} e_2) \qquad (31) \\
& \quad \text{if } \neg(f_1 \theta f_2)(c)
\end{aligned}
$$

We can easily check these conditions if some predicate $(f_1 \theta f_2)(c)$ yields, e.g.

$$
\begin{array}{ccll}
i & \theta & 0 & (i \text{ constant}) \\
f_1 & \in & \emptyset &
\end{array}
$$

or has a similar form. An application of these equivalences can sometimes be followed by a removal of the join.

Note, that the latter equivalence depends on the knowledge we have on the selection predicate. Note also, that the outerjoin is only introduced by unnesting nested χ operations. Hence, we could combine the equivalences introducing the outerjoin and replacing it by a join into one. In case we have even more information on the selection predicate than above, more specifically, if it depends on a max or min aggregate, we can do so in a very efficient way:

$$
\begin{aligned}
\sigma_{a=m}(e_1)[m : agg(\chi_b(e_2))] &= Agg_{g;m;a}(e_1).g \qquad (32) \\
& \quad \text{if } \pi_a(e_1) = \pi_b(e_2) \\
\chi_{g:\sigma_{a=m}(e_2)}(\chi_{m:agg(e_1)}(e)) &= \chi_{\circ Agg_{g;m;a}(e_1)}(e) \qquad (33) \\
& \quad \text{if } \pi_a(e_1) = \pi_b(e_2)
\end{aligned}
$$

This will save some scans as will be demonstrated in Section 4.1.1.

3 Dependency-Based Optimization

Throughout the paper we use O_2SQL [2] for the examples. However, the techniques that we present are not restricted to this language and can easily be applied to other languages like GOMql [20]. Maybe it is more important to note that O_2SQL is a subset of the oo standard query language OQL as defined in [7]. We do not give an introduction into the query language but assume the reader to be familiar with O_2SQL or OQL. Further, the schema we will use for our example queries will not be given explicitly since we are sure that it can be inferred easily from the queries.

In the relational context, a nested query is one containing a block nested inside. In the object-oriented context, things are different. Operations inside a block may be method calls, long path expressions, set operations on attributes, etc. Hence, we have to consider the optimization of nested expressions—which are also queries by definition—which are not blocks.

We briefly review the first phase called *dependency-based optimization*. This phase factors out common subexpressions by introducing appropriate definitions collected within an extra **define** clause. We illustrate the phase by repeating an example of [11].

```
select  tuple(name:emp.name,
              sale:emp_sale.description,
              month: emp_sale.date.month)
from    emp in Employee,
        emp_sale in Sale
where   emp_sale in emp.best_sales() and
        emp_sale.amount > avg(select  sale.amount
                              from    sale in Sale
                              where   sale.date.month = emp_sale.date.month)
```

The method call to *best_sales* is used in the join predicate. We consider this as a nested query. A good optimization will push it out of the join operation such that it is not evaluated more often than necessary. The *emp_sale.date.month* path expression in the nested block is also a nested query that should be pushed out of (i) its nesting block and (ii) the join performed in the higher level block.

The *dependency-based optimization* is performed at the O_2SQL level. This kind of optimization, although vital, is simple enough and requires mainly one traversal of the syntax tree of the query. Since it had been presented elsewhere [10], we will not detail it. We merely present the transformed above O_2SQL query and comment on it.

```
select  tuple(name:en, sale:esd, month: esdm)
from    emp in Employee,
        emp_sale in ebs
where   esa > as
define  en = emp.name,
        esd = emp_sale.description,
        esdm = emp_sale.date.month,
        ebs = emp.best_sales(),
        esa = emp_sale.amount,
        as = avg(select  sa
                 from    sale in Sale
                 where   sdm = esdm
```

$$\begin{array}{ll}\textbf{define} & \text{sa} = \text{sale.amount}\\ & \text{sdm} = \text{sale.date.month}\end{array}$$

The transformed query features a new **define** clause in each block. It is used to introduce variables representing expressions dependent on its owner block only. Note that the outer block **define** clause contains a variable for the *emp_sale.date.month* that is referenced twice (common subexpressions factorization): once in the inner block (constant subexpression factorization) and once in the outer block.

Translation into the Algebra The translation of a SFWD-block into the algebra proceeds as follows. First, all the entries in the **from** clause are connected via d-joins. This results in an expression e'_f. If possible, they will be replaced by cross products resulting in an expression e_f not necessarily different from e'_f. Next, all entries $v_i = e'_i$ of the **define** clause are translated into $\chi_{v_i:e_i}$ expressions, where e_i is the translation of e'_i. The third step consists in translating the **where** clause into a selection σ_p with p being the translation of the **where** clause predicate. The fourth and last step consists of adding a last χ_{e_s} where e_s is the translation of the expression in the **select** clause. Hence, the total result of the translation process is

$$\chi_{e_s}(\sigma_p(\chi_{v_1:e_1}(\ldots(\chi_{v_n:e_n}(e_f)))))$$

This normal form has been called MCNF [22] in case only crossproducts and no d-joins occur in e_f. The next sections show plenty of examples. Hence, we don't give one here.

If some entries in the **define** clause of the outermost block are not dependent on the entries in the **from** clause of that block, than they can be factored out and be evaluated only once. This then results in a translation of the form $[v : e]$ for constant expressions $v_j = e_j$ in the **define** clause. If e is the translation of the rest of the block, then the result is an expression $e([v_1 : e_1, \ldots, v_k : e_k])$. The tuple constructed just serves to bind the v_j within the expression e.

Remark It is important to note that each nested block B will be replaced by a new variable V_B whose definition $V_B = B$ is added to the **define** clause. This convention mostly results — via the translation process — in expressions of the form $\chi_{V_B=B}(e)$ where e is the translation of (part of) the surrounding block. Note that evaluating this expression results in a nested loop. This is rather inefficient. To avoid nested loop evaluation, the main issue is to move B outside the subscript. This is what unnesting boils down to when — as in this paper — performed at the algebraic level. Moving B outside cleverly such that the result can be efficiently evaluated is far from trivial. But if we succeed, we replace the nested loop evaluation by some more efficiently evaluable operation (e.g., a join).

4 Classification and Algebraic Optimization

In this section we present a classification of oo nested queries along with appropriate algebraic optimization techniques. In order to better understand the analogies between relational and oo environments, we extend the relational classification of nested queries as given in [26]. Hereby, we restrict ourselves to a single nested block. This is relaxed in the next section. Kim's classification introduces five types of nested queries one of which is not used here (Type D). The reason for this will be found in the sequel: They can be treated in a more uniform way than in the relational case (see subsection 4.1.2). The four remaining types are

- **Type A** nested queries have a constant inner block returning single elements.
- **Type N** nested queries have a constant inner block returning sets.
- **Type J** nested queries have an inner block that is dependent on the outer block and returns a set.
- **Type JA** nested queries have an inner block that is dependent on the outer block and returns a single element.

Obviously, the need for extending the relational classification arises from the richness of the oo model compared to the relational one and its impact on the query language. The classification we propose has three dimensions: the original one plus two that are required by the following oo characteristics. In the oo context, as opposed to the relational, (i) nested blocks may be located in any clause of a **select-from-where** query and (ii) a dependency (i.e., reference to a variable of the outer block) may be expressed in any clause of a query inner block. The organization of this section follows the three dimensions. We start by presenting nested queries of type A/N/J/JA with nesting and dependency (J/JA only) in the **where** clauses. This will allow us to show the differences and similarities between relational and oo context. We continue by explaining the treatment required by other locations of nesting (i.e., **select** and **from** clauses). We end with the optimization of type J/JA queries with range (**from** clause) or projection (**select** clause) dependencies.

As in the relational context, the optimization of nested queries is done by unnesting, using different kinds of joins and group operators. There are two good reasons for unnesting nested queries. The first is that the underlying evaluation plan of a nested query relies on nested loops that, as shown in [26], can be very inefficient. On the other hand, we know of good algorithms for joins and group operations (using indexes, sorting, hashing). The second reason is that algebraic operators have nice properties that can be used for further rewriting whereas nested algebraic expressions don't have them a priori.

4.1 Different Types of Nesting

4.1.1 Queries of Type A

For queries of this kind, we rely on a technique similar to that of [26]. The first phase of the optimization process factors out constant subqueries. Thus, as for relational queries, the inner block of a type A query is evaluated first and its result is used for the evaluation of the outer block.

In special though frequent cases, it is possible to do better than just pushing out the constant block to evaluate type A queries: if the function applied on the inner block is either **min** or **max** and if the inner and outer blocks have a common domain [11]. The last condition seems rather restrictive but retrieving the element exhibiting the min/max value implies just this. For these queries, the scan needed for the evaluation of the aggregate function on the inner block can also be used to compute the outer block. Let us illustrate this by means of a **max** query.

```
select x1
from   x1 in Employee
where  x1.TotSales =
       max (select x2.TotSales
            from   x2 in Employee)
```

```
define m = max(select x2s
               from   x2 in Employee)
               define x2s = x2.TotSales
select x1
from   x1 in Employee
where  x1s = m
define x1s = x1.TotSales
```

The original query is on the left-hand side. The rewritten query after the first phase is shown on the right-hand side—a convention holding throughout the paper. Note that the first **define** entry is independent of the second block. Hence, it is written before it.

Translation into the algebra yields

$$\begin{aligned} q &\equiv \chi_{x1}(\sigma_{x1s=m}(\chi_{x1s:x1.TotSales}(Employee[x1])))) \\ m &\equiv Max(\chi_{x2s}(\chi_{x2s:x2.TotSales}(Employee[x2))) \\ &\equiv Max(\chi_{x2.TotSales}(Employee[x2])) \end{aligned}$$

where a block with one variable in the **from** clause is translated by (i) a map operation for constructing tuples whose single attribute represents the variable (e.g., $Employee[x1]$), (ii) a map operation for evaluating the expressions depending on the block (e.g., $\chi_{x1s:x1.TotSales}$), (iii) a selection when needed (e.g., $\sigma_{x1s=m}$) and finally (iv) a map operation for building the final result (e.g., χ_{x1}). Of course, all the tuple constructions implied by this translation do not have to be performed (e.g., $Employee[x1]$ can be interpreted as a scan on $Employee$ with variable $x1$).

We may now apply the following equivalence which is a slightly generalized version of the one found in [11]:

$$\begin{aligned} f(\sigma_{a\theta max(e_2)}(e_1)) &\equiv Max_{g;m;a\theta;f}(e_1).g \\ &\quad \text{if } e_2 = \pi_a(e_1) \end{aligned} \tag{34}$$

Remember that the $Max_{g;m;a\theta;f}(e)$ operation returns a tuple containing (i) an attribute m representing the maximum value for the attribute a in the set e and (ii) an attribute g representing the result of f applied to the set of elements of e satisfying $a\theta m$. The above equivalence applied to the query yields:

$$q \equiv Max_{g;m;x1s;\chi_{x1}}(\chi_{x1s:x1.TotSales}(Employee[x1])).g$$

Note that the Max operator can be computed within a single pass (linear time) for $Max_{g;m\theta;f}$, if f is linear. Also note that an equivalent treatment for Min can be applied. Furthermore, although the equivalence we used can easily be adapted to the relational context, we are not aware of any such optimization.

4.1.2 Queries of Type N

An important difference between relational and oo type N queries concerns the connection predicate between the two blocks. In SQL, this predicate is built using either the **IS IN** or an atom based comparator. In the latter case, the relation resulting from the inner block has to contain one tuple with one unique attribute and the comparison is made on the value of this attribute. SQL type N queries are transformed using a semi-join operation. An atom based predicate requires some additional mechanisms (e.g., based on functional dependencies [13]) for checking the uniqueness of the element returned by the inner block.

In O_2SQL, things are different. Since the object model does not require a first normal form, comparing an attribute with the result of a block is not restricted to a membership test but can be any set comparison operation. Furthermore, in O_2SQL, atomic comparison implies the use of the **element** operation on the inner block. This operation extracts the element of a singleton set. Since it returns a single element, we consider these queries as being of type A.

We distinguish three different kinds of predicates occurring within the outer **where** clause:

1. $f(\vec{x})$ **in select** ...
2. **not** ($f(\vec{x})$ **in select** ...)
3. $f(\vec{x}) = (\subseteq, \supseteq, \ldots)$ **select** ...

where $\vec{x}$ represents variables of the outer block, f a function (or subquery) on these variables and $=, \subseteq, \supseteq, \ldots$ are set comparisons.

The techniques for unnesting the first two cases are adapted from the relational context [8, 13, 26] and, hence, we will only briefly cast these techniques into algebraic equivalences.

1. Type N queries with an **in** operator can be transformed into a semi-join by using the following equivalence inspired by relational type N unnesting:

$$\begin{aligned} \sigma_{A_1 \in \chi_{A_2}(e_2)} e_1 \quad &\equiv \quad e_1 \rhd\!\!<_{A_1=A_2} e_2 \\ &\quad\;\; \text{if } A_i \subseteq \mathcal{A}(e_i),\ \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset \end{aligned} \tag{35}$$

   The first condition is obvious, the second merely stipulates that expression e_2 must be independent of expression e_1. The interest for having this equivalence and the following is obvious. As stated previously, semi-joins and anti-joins can be implemented efficiently and they allow further rewriting.

2. Also inspired by the relational type N unnesting is the following equivalence which turns a type N query with a negated **in** operator into an antijoin:

$$\begin{aligned} \sigma_{A_1 \notin \chi_{A_2}(e_2)} e_1 \quad &\equiv \quad e_1 \rhd_{A_1=A_2} e_2 \\ &\quad\;\; \text{if } A_i \subseteq \mathcal{A}(e_i),\ \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset \end{aligned} \tag{36}$$

The third case does not have a counterpart in SQL. However, if we formulate the corresponding queries on a relational schema using the non-standard SQL found in [26], they would be of type D — resolved by a division. Using a standard SQL, they would require a double nesting using **EXISTS** operations. Treating Type D queries by a relational division can only treat very specific queries where the comparison predicate corresponds, in our context, to a non-strict inclusion as in the example below. The query returns the employees who have sold all the expensive products.

```
select x                                      define  ExpItems = select  i
from   x in Employee                                             from    i in Item
where  x.SoldItems ⊇ select i                                    where   p > 20000
                     from   i in Item                            define  p = i.price
                     where  i.price > 20000   select x
                                              from   x in Employee
                                              where  xsi ⊇ ExpItems
                                              define xsi = x.SoldItems
```

One solution to evaluate this query is to use a technique similar to that of [26] and add to our algebra an object division. If the set of expensive items is important, a well implemented division operation could do much compared to a nested loop evaluation. However, we voted against this operation for three reasons. The first reason is, as we stated before, that the division is based on a non strict inclusion of the divider set. There are no more reasons to have this inclusion than any other set comparison ($\subseteq$, $\supset$, ...). Accordingly, to be coherent, we would have to introduce one operation per set comparator (as a matter of fact, this also holds for the relational context). The second reason is that division does not have particularly nice algebraic properties that we would like to exploit. The third reason is that, since object models feature set attributes, it seems more natural to add good algorithms for dealing with selections involving set comparisons than to add new algebraic operators. Further, there already exist proposals to treat restriction predicates involving set comparison operators [36]. Thus, we prefer not to rewrite the following algebraic expression which corresponds to the translation of the above query.

$$\begin{aligned} q &\equiv \chi_x(\sigma_{xsi \supseteq ExpItems}(\chi_{xsi:x.SoldItems}(Employee[x]))) \\ ExpItems &\equiv \chi_i(\sigma_{p>20000}(\chi_{p:i.price}(Item[i]))) \end{aligned}$$

The set $ExpItems$ will be evaluated first, independently of query q. The result of its evaluation will be used by the selection $\sigma_{xsi \supseteq ExpItems}$ in the outer block. The selection itself can be evaluated using an algorithm similar to that of a relational division.

Note that there is no need to consider the negation of set comparisons, since it is possible to define for each set comparison an equivalent negated counterpart. Consider for example $\neg(e_1 \subseteq e_2)$ and the set comparison operator $\not\subseteq$ defined as $(e_1 \not\subseteq e_2) := (e_1 \setminus e_2 \neq \emptyset)$.

4.1.3 Queries of Type J

For Type J queries, we distinguish the same three cases as for Type N queries. Again, queries with **in** (**not in**) as the connection predicate are transformed, as in the relational, using semi-joins (anti-joins). However, as we will see in short, there exists another unnesting possibility for these kind of queries. Nevertheless, we adapt the traditional unnesting technique first. This is done by recasting the technique into algebraic equivalences.

1.

$$\sigma_{A_1 \in \chi_{A_2}(\sigma_p(e_2))} e_1 \quad \equiv \quad e_1 \triangleright\!\!<_{A_1 = A_2 \wedge p}\, e_2 \qquad (37)$$
$$\text{if } A_i \subseteq \mathcal{A}(e_i),\ \mathcal{F}(p) \subseteq \mathcal{A}(e_1 \cup e_2),\ \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$$

This equivalence is similar to the one used for type N queries. It just takes into account a predicate p relying on both e_1 and e_2 (second condition).

2.

$$\sigma_{A_1 \notin \chi_{A_2}(\sigma_p(e_2))} e_1 \quad \equiv \quad e_1 \triangleright_{A_1 = A_2} (e_2 \triangleright\!\!<_p e_1) \qquad (38)$$
$$\text{if } A_i \subseteq \mathcal{A}(e_i),\ \mathcal{F}(p) \subseteq \mathcal{A}(e_1 \cup e_2),\ \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$$

Type J **not in** queries cannot be translated directly using an anti-join operation: a semi-join has to be performed first.

Now, let us consider the third case. The query below returns the employees who have sold all the items with a high-tech degree larger than the sales speciality of the employee.

```
select x
from   x in Employee
where  x.SoldItems ⊇ select i
                     from   i in Item
                     where  i.hTD > x.speciality
```

```
select x
from   x in Employee
where  xsi ⊇ SpecialItems
define xsi = x.SoldItems
       xs = x.speciality
       SpecialItems = select  i
                      from    i in Item
                      where   ihTD > xs
                      define  ihTD = i.hTD
```

The algebraic translation of the query is splitted for reasons of clarity:

$$\begin{aligned}
q &\equiv \chi_x(\sigma_{xsi \supseteq SpecialItems}(q_1)) \\
q_1 &\equiv \chi_{SpecialItems:\chi_i(\sigma_{ihTD > xs}(q_3))}(q_2) \\
q_2 &\equiv \chi_{xsi:x.SoldItems, xs:x.speciality}(Employee[x]) \\
q_3 &\equiv \chi_{ihTD:i.hTD}(Item[i])
\end{aligned}$$

The problem here is that the nested query is not constant. In order to unnest the query and avoid several costly scans over the set of items, we have to associate with each employee its corresponding set of special items. For this, we rely on the following equivalence:

$$\chi_{g:f(\sigma_{A_1 \theta A_2}(e_2))}(e_1) \quad \equiv \quad e_1 \Gamma_{g;A_1 \theta A_2;f}\, e_2 \qquad (39)$$
$$\text{if } A_i \subseteq \mathcal{A}(e_i),\ g \notin A_1 \cup A_2,\ \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$$

Applying this equivalence on q_1 results in

$$q \quad \equiv \quad \chi_x(\sigma_{xsi \supseteq SpecialItems}(q_2 \Gamma_{SpecialItems;ihTD > xs;\chi_i}\, q_3))$$

The binary grouping operation can be implemented by adapting standard grouping algorithms. There is another alternative to this operation that will be given in the sequel.

Two remarks. First, note that the selection with set comparator $\supseteq$ is now evaluated between two attributes. As for type N queries, we rely on good algorithms for such selections. Second, note that the application of the equivalence did not depend on the set comparison of the predicate in the outer **where** block but on the comparison of the correlation predicate within the inner block. We will come back to this point, soon.

Eqv. 39 is the most general equivalence for the considered type of queries. There exist two other equivalences which deal more efficiently, using simple grouping, with two special cases. The equivalence

$$\begin{array}{rcl} \chi_{g:f(\sigma_{A_1=A_2}(e_2))}(e_1) & \equiv & \pi_{\overline{A_2}}(e_1 \sqsupset\!\!\bowtie^{g=f(\emptyset)}_{A_1=A_2} (\Gamma_{g;A_2;f}(e_2))) \\ & & \text{if } A_i \subseteq \mathcal{A}(e_i),\ \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset,\ A_1 \cap A_2 = \emptyset, g \not\in \mathcal{A}(e_1) \cup \mathcal{A}(e_2) \end{array} \quad (40)$$

relies on the fact that the comparison of the correlation predicate is equality. The superscript $g = f(\emptyset)$ is the default value given when there is no element in the result of the group operation which satisfies $A_1 = A_2$ for a given element of e_1. The equivalence

$$\begin{array}{rcl} \chi_{g:f(\sigma_{A_1 \theta A_2}(e_2))}(e_1) & \equiv & \pi_{A_1:A_2}(\Gamma_{g;A_2\theta;f}(e_2)) \\ & & \text{if } A_i \subseteq \mathcal{A}(e_i),\ \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset, \\ & & g \not\in \mathcal{A}(e_1) \cup \mathcal{A}(e_2), \\ & & e_1 = \pi_{A_1:A_2}(e_2) \text{ (this implies that } A_1 = \mathcal{A}(e_1)) \end{array} \quad (41)$$

relies on the fact that there exists a common range over the variables of the correlation predicate (third condition). We believe that these two cases are more common than the general case. We will show one application of Eqv. 40 in this section. In the next one, we will give an example using an equivalence derived from Eqv. 41.

Eqv. 39, 40, and 41 are not only useful for unnesting type J nested queries occurring within the **where** clause in a predicate utilizing set comparison. As already remarked above, applying these equivalence solely depends on the presence of a correlation predicate. Hence, they enable the derivation of alternative unnested expressions for the **in** and **not in** cases. To see this, consider $\sigma_{A \in e_2}(e_1) \equiv \sigma_{A \in B}(\chi_{B:e_2}(e_1))$. Further, as demonstrated in the next Section, they play a major role in unnesting type JA nested queries. That is why they should be considered *the core* of unnesting nested queries in the oo context.

Further, alternative unnested evaluation plans avoiding the binary grouping operator can also be achieved by applying the following equivalence which produces an intermediate flat result and then groups it

$$\begin{array}{rcl} \chi_{g:f(\sigma_{A'_1 \theta A'_2}(e_2))}(e_1) & \equiv & \Gamma_{g;A_1;f \circ \pi_{\overline{A_1}} \circ \sigma_{A_2 \neq \perp_{A_2}}}(e_1 \sqsupset\!\!\bowtie_{A'_1 \theta A'_2} e_2) \\ & & \text{if } A_i = \mathcal{A}(e_i),\ A'_i \subseteq A_i,\ g \not\in A_1 \cup A_2,\ \mathcal{F}(e_2) \cap A_1 = \emptyset \end{array} \quad (42)$$

where $\perp_A$ is a tuple with attributes A and null values only. Which of the Eqns. 39–42 to apply is a matter of costs.

Last, there is a variant of Eqn. 39 in case no selection is present:

$$\begin{array}{rcl} \chi_{g:f(e_2)}(e_1) & \equiv & e_1 \Gamma_{g;true;f} e_2 \\ & & \text{if } g \not\in \mathcal{A}(e_1) \cup \mathcal{A}(e_2), \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset \end{array} \quad (43)$$

4.1.4 Queries of Type JA

In the relational context, the treatment of type JA queries is radically different from that of type J, N or A. It requires joins, grouping and sometimes outer-joins [13, 17] (remember, from the previous subsections, that type N/J SQL queries required anti-joins and semi-joins). In the oo context, there is no difference between type J and type JA queries. The reason is that, in order to deal with set comparison, we already introduced outer-joins and grouping operations. The grouping operators have been defined to allow the application of functions to the sets of grouped elements. This function might as well be an aggregate function. Thus, by applying Eqv. 39–42 aggregated type JA queries are treated in exactly the same manner as type J queries.

Note that, if the applied function of the unary Γ in Equivalence 40 is an aggregate function (as implied by type JA queries), then its right-hand side is equivalent to the generalized aggregation of [13].

4.2 Different Locations of Nesting

4.2.1 Nesting in the *select* Clause

Although nothing forbids it, type A or N nesting rarely occurs in **select** clauses. Indeed, there is not much sense in associating a constant (set or element) to each element of a set. Should that happen, we rely on the first phase of the optimization process to factor out the constant block. Thus, it will only be evaluated once.

For type J/JA queries, nesting in the **select** clause is equivalent to nesting in the **where** clause. Remember that the application of Eqns. 39–42 did not depend on the predicate in the outer **where** block but on the correlation predicate within the inner block. The same kind of correlation predicates is used when type J/JA nesting occurs in the **select** clause. We illustrate this with the following type JA query that associates to each department its number of employees.

```
select tuple(dept: d,
             emps: count( select e
                          from   e in Employee
                          where  e.dept=d))
from   d in Department
```

```
select tuple(dept: d, emps: ce)
from   d in Department
define ce = count( select e
                   from   e in Employee
                   where  ed=d
                   define ed=e.dept)
```

Translating this expression into the algebra yields

$$\begin{aligned} q &\equiv \chi_{[dept:d,emps:ce]}(\chi_{ce:q_1}(Department[d])) \\ q_1 &\equiv count(\chi_e(\sigma_{ed=d}(\chi_{ed:e.dept}(Employee[e])))) \end{aligned}$$

Eqv. 40 can be applied yielding:

$$\begin{aligned} q &\equiv \chi_{[dept:d,emps:ce]}(\pi_{\overline{ed}}(Department[d] \;⟕^{ce=0}_{d=ed} (\Gamma_{ce;ed;count\circ\chi_e}(\chi_{ed:e.dept}Employee[e])))) \\ &\equiv \pi_{\overline{ed}}(Department[d] \;⟕^{ce=0}_{d=ed} (\Gamma_{ce;ed;count\circ\chi_e}(\chi_{ed:e.dept}Employee[e]))) \end{aligned}$$

The zero value in the superscript $ce = 0$ corresponds to the result of the **count** function on an empty set. The transformed query can be evaluated efficiently using, for instance, a sort or an index on *Employee.dept*.

There exists one type J case where another more powerful technique can be applied: a *flatten* operation is performed on the outer block, and there is no tuple constructor within the outer block's **select** clause. As shown in [11], these queries can be optimized by pushing the *flatten* operation inside until it is applied on stored attributes; thus eliminating the nesting. For completeness, we repeat the example. The example query is

```
flatten(select
          select  tuple(name:c.name,age:c.age)
          from    c in e.children
          where   c.age < 18)
       from e in employee)
```

```
flatten(select  g
        from    e in employee
        define  ec = e.children
                g = select  tuple(name:n,age:a)
                    from    c in ec
                    where   a < 18
                    define  n = c.name
                            a = c.age )
```

The standard translation gives

$$
\begin{aligned}
q &\equiv flatten(\chi_g(\chi_{g:e_2}(\chi_{ec:e.children}(Emp[e]))))\\
e_2 &\equiv \chi_{[name:n,age:a]}(\sigma_{a<18}(\chi_{a:c.age,n:c.name}(ec[c])))
\end{aligned}
$$

In order to push the flatten operation inside, we have to eliminate the redundant tuple extension for the attribute g:

$$
\begin{aligned}
q &\equiv flatten(\chi_{e_2}(\chi_{ec:e.children}(Emp[e]))))\\
e_2 &\equiv \chi_{[name:n,age:a]}(\sigma_{a<18}(\chi_{a:c.age,n:c.name}(ec[c])))
\end{aligned}
$$

Now, we know that for linear $f : \{\tau\} \rightarrow \{\tau'\}$ that

$$
flatten(\chi_f(e)) \quad = \quad f(flatten(e)) \tag{44}
$$

Hence,

$$
\begin{aligned}
q &\equiv \chi_{[name:n,age:a]}(flatten(\chi_{e_2'})(\chi_{ec:e.children} Emp[e]))\\
e_2' &\equiv \sigma_{a<18}(\chi_{a:c.age,n:c.name}(ec[c]))\\
q &\equiv \chi_{[name:n,age:a]}(\sigma_{a<18}(flatten(\chi_{e_2''}(\chi_{ec:e.children} Emp[e])))))\\
e_2'' &\equiv \chi_{age:c.age,n:c.name}(ec[c])\\
q &\equiv \chi_{[name:n,age:a]}(\sigma_{a<18}(\chi_{a:c.age,n:c.name}(flatten(\chi_{ec[c]}(\chi_{ec:e.children} Emp[e])))))\\
&\equiv \chi_{[name:n,age:a]}(\sigma_{a<18}(\chi_{a:c.age,n:c.name}(flatten(\chi_{e.children[c]}(Emp[e])))))\\
&\equiv \chi_{[name:n,age:a]}(\sigma_{a<18}(\chi_{a:c.age,n:c.name}(flatten(\chi_{children[c]}(Emp)))))\\
&\equiv \sigma_{age<18}(\chi_{[name:n,age:a]}(\chi_{[a:age,n:name]}(flatten(\chi_{children}(Emp)))))\\
&\equiv \sigma_{age<18}(\chi_{[name:name,age:age]}(flatten(\chi_{children}(Emp))))
\end{aligned}
$$

where redundant tuple constructions were eliminated in the last steps. Note that the flatten operation is now applied on stored data.

4.2.2 Nesting in the *from* Clause

Although rare, types A and JA nesting may occur in from clauses if they are combined with another operation. For instance, one may extract a set attribute in a tuple returned by a type A query. Type A queries are always treated in the same way independent of the position in the block. Type JA queries can be treated in a way equivalent to that of type J given below.

Type N nesting in the **from** clause may occur, as in the relational context, when querying a view. SQL queries involving views are considered in [18] where the view and the query have their own query graph model (QGM) representation. The Starbust technique is to merge the two QGM's when possible (problem when **distinct** is involved) in order to have more optimization opportunities. Our query representation is different but the result is equivalent. Translation of the **from** clause results in a d-join. For type N queries this is equivalent to a crossproduct. Predicates which might occur in the outer **where** clause can be used to turn the cross product into a join which often turns out to be a semi-join. We do not elaborate on this since it has been described in [11].

Type J nesting in the **from** clause is also treated easily. To understand this, consider the following equivalence:

$$e_1 < \sigma_p(e_2) > \equiv \sigma_p(e_1 < e_2 >) \tag{45}$$

In the current context (i.e., a dependency in the **where** clause only), once the selection is pushed out, the inner argument of the d-join is constant. Thus, we can transform the d-join coupled with selection into a regular join.

4.3 Different Kinds of Dependency

As stated above, we distinguish three kinds of dependency: projection dependency (a reference to an outer variables occurs in the **select** clause), range dependency (...in the **from** clause) and predicate dependency (...in the **where** clause). Above, we studied queries with predicate dependency. In the sequel, we concentrate on optimization techniques required for range and projection dependencies.

4.3.1 Range Dependency

Consider the following query exhibiting a range dependency. It returns the set of employees having the same name than one of their children.

```
select  x
from    x in Employee
where   x.name in select c.name
                  from   c in x.children
```

```
select  x
from    x in Employee
where   xn in CN
define  xn = x.name
        xc = x.children
        CN = select  cn
             from    c in xc
             define  cn = c.name
```

The algebraic translation is:

$$
\begin{aligned}
q &\equiv \chi_x(\sigma_{xn \in CN}(\chi_{CN:nq}(\chi_{xn:x.name,xc:x.children}(Employee[x]))))\\
nq &\equiv \chi_{c.name}(xc[c])
\end{aligned}
$$

In terms of unnesting, there is nothing one can do. Nevertheless, the **where** clause of the above query is equivalent to the application of the path expression *x.children.name* which passes through a set. Hence, in this case, already known optimization techniques for optimizing path expressions can be applied.

However, there exist cases where we are able to advantagely reduce range dependencies to predicate dependencies and, hence, can unnest these queries by the above introduced techniques. The reduction relies on the existence of type extents and uses *type based rewriting* [10, 19, 27, 28, 29]. Since it has already been described, we merely present its usage as a reduction technique useful for enabling further unnesting of range dependent subqueries. The example query is

```
select tuple (e: x.name, c: select s.customer.name
                            from   s in = x.sales
                            where  s.customer.city
                                     = "Karlsruhe")
from   x in   Employee
```

```
select tuple (e: xn, c: SCN)
from   x in Employee,
define xn = x.name
       xs = x.sales
       SCN = select scn
             from   s in xs
             where  scc = "Karlsruhe"
             define sc = s.customer
                    scn = sc.name
                    scc = sc.city
```

Translation to the algebra yields

$$
\begin{aligned}
q &\equiv \chi_{[e:xn,c:SCN]}(\chi_{SCN:nq}(\chi_{xn:x.name,xs:x.sales}(Employee[x])))\\
nq &\equiv \chi_{scn}(\sigma_{scc=\text{``}Karlsruhe''}(\chi_{scn:sc.name,scc:sc.city}(\chi_{sc:s.city}(xs[s]))))
\end{aligned}
$$

Relying on the fact that the elements of the attribute *sales* of an employee belong to the extent of the class **Sale**, the inner block of the query can be rewritten as

$$
nq \equiv \chi_{sc.name}(\sigma_{scc=\text{``}Karlsruhe''}(\chi_{sc:s.customer,scc:sc.city}(\sigma_{s \in xs}(Sale[s]))))
$$

Type based rewriting can be performed again using the extent of class Customer. This allows us, for instance, to use indexes on *Customer.city* and *Sale.customer* to evaluate the query. However, since our goal is unnesting and not general optimization, we do not detail on this. Concerning unnesting, it is important to note that the dependency no longer specifies the range ($xs[s]$) but now represents a predicate ($\sigma_{s \in xs}$). Herewith, the algebraic expression is of the same form as one resulting from a predicate dependency. Hence, our unnesting techniques apply.

4.3.2 Projection dependency

Queries of this kind should be rare. If they occur, they do so in two different flavors. One nice one and one nasty one. The first occurs if, within the expression forming the **select** clause, an expression occurs whose variables all depend on the outer block. Then, this expression has to be computed only once for each variable combination resulting from the evaluation of the outer block. Besides this expression, the evaluation of the inner block is independent of outer variables. Hence, it can be factored out resulting in a halfway efficient evaluation plan. The nasty case, where the expression contains variables from the outer and the inner block, requires in general the nested loop evaluation or cross product.

Remark Nothing restricts variables of the outer block to occur only at one place within the inner block. If there exist several dependencies, all the corresponding unnesting techniques can be applied alternatively. Hence, if for example a range and a predicate dependency occur, the latter should be used for unnesting if the range dependency cannot be resolved by type based rewriting.

5 More Unnesting Techniques

5.1 Several Nested Blocks

Queries containing several nested blocks of the same level are unnested by successive rewritings as in the relational context. Note that if the nesting occurs in a disjunctive **where** clause, the query has to be transformed first into a union.

Queries containing several levels of nesting are more complex. In the simple case, we can start the rewriting by the lower level and go up. However, when queries contain non-neighbor predicates, it is not always possible to apply the equivalences we introduced in the previous section. We will see below what has to be done then.

Finally, note that some queries may be left nested. This happens when, as stated in the previous section, range dependency cannot be transformed into predicate dependency. This also occurs when boolean methods are used as a correlation predicate between inner and outer block.

5.2 Quantifiers

In the relational context, quantifiers are always used in the **where** clause of a **select-from-where** (SFW) block. In [13], existential quantifiers are considered as special aggregate functions (i.e., checking the existence of one element in a set). This opens the road for efficient evaluation but cannot be adapted to universal quantifiers. In [17], queries involving quantifiers are rewritten using the **count**, **min** or **max** aggregate functions. Due to the fact that, as opposed to quantifiers, the aggregate operations require a full scan on their set argument, the latter technique is less efficient.

In O_2SQL and OQL[7], quantifiers are independent functions. They would correspond in SQL to the combination of a quantifier and a SFW block. They can be used anywhere and even form the outer block of a query. This is why we introduced two special operators to deal with these. $\forall_p(e)$ (resp. $\exists_p(e)$) returns $true$ if all (resp. at least one) elements in e satisfy p, else it returns $false$.

Nested queries with quantifiers come in three flavors. The nested block may define the quantifier domain, be part of the quantifier condition or the quantifier may be nested in another block.

5.2.1 Nesting Within a Quantifier Range Definition

Due to the definition of O_2SQL quantifiers, a nested block defining the domain is always constant. This kind of query is optimized easily as demonstrated by the following example:

```
exists x                                  define  EMP = select  emp
in     select emp                                       from    emp in Employee
       from   emp in Employee                           where   s > 20000
       where  emp.TotSales > 20000                      define  s = emp.TotSales
where (x.address.city="Paris")            exists x
                                          in     EMP
                                          where (c = "Paris")
                                          define c = x.address.city
```

The algebraic translation yields

$$\begin{aligned} q &\equiv \exists_{c=\text{“}Paris''}(\chi_{c:x.address.city}(EMP[x])) \\ EMP &\equiv \chi_{emp}(\sigma_{s>20000}(\chi_{s:emp.TotSales}(Employee[emp]))) \end{aligned}$$

It is transformed in the following way:

$$\begin{aligned} q &\equiv \exists_{c=\text{“}Paris''}(\chi_{c:x.address.city}(\chi_x(\sigma_{s>20000}(\chi_{s:x.TotSales}(Employee[x]))))) \\ &\equiv \exists_{c=\text{“}Paris''}(\chi_{c:x.address.city}(\sigma_{s>20000}(\chi_{s:x.TotSales}(Employee[x])))) \\ &\equiv \exists_{(c=\text{“}Paris''\wedge s>20000)}(\chi_{c:x.address.city}(\chi_{s:x.TotSales}(Employee[x]))) \end{aligned}$$

The first transformation results from the variable binding convention and renaming. The second one eliminates an unnecessary χ operation and pushes another one before a selection. Finally the last one is based on the following equivalence which allows us to push a selection inside an $\exists$ operation.

$$\exists_{p_1}(\sigma_{p_2}(e)) \equiv \exists_{p_1 \wedge p_2} e \tag{46}$$

With this rewriting, the scan on the set of employees may be stopped when encountering one living in Paris and having the right amount of sales. An index on *Employee.address.city* can also be used to evaluate the $\exists$ operation.

The example we considered was based on an **exists** block. Things are rather the same with **forall** blocks. The only difference is in the last equivalence. We would have to use the following with a $\forall$ block.

$$\forall_{p_1}(\sigma_{p_2}(e)) \equiv \forall_{p_1 \vee \neg p_2} e \tag{47}$$

5.2.2 Nesting Within a Quantifier Predicate

When the nested block is part of the condition of a quantifier, a treatment similar to that of SFW blocks with nesting in the **where** clause can be applied. This should be obvious for type A/N queries. Concerning type J/JA queries, remember that applying Equivalences 39, 40, or 41 did not rely on the predicate in the outer **where** block (in this case a quantifier) but on the correlation predicate within the inner block.

Another technique can also be used for special cases. It consists in transforming the quantifier into a set comparison, using one such equivalence.

$$\forall_{\lambda x(x \in e_2)}(e_1) \;\equiv\; e_1 \subseteq e_2 \tag{48}$$

Note that the opposite approach, that is converting set operations into quantifiers has been proposed in [36].

5.2.3 Nested Quantifiers

Nested quantifiers require new equivalences. These equivalences are derived from Eqvs. 39–42. The difference is that the selection in the inner block used to define the grouping in the previous cases is here replaced by a quantifier.

We illustrate this with the query below. It corresponds to the simple case that has not been illustrated thus far: the domain is shared by the inner and the outer blocks. The query returns those employees having the same name as some other employee.

```
select  x1
from    x1 in Employee
where   exists x2
        in     Employee
        where x2.name = x1.name and x1<>x2)
```

```
select  x1
from    x1 in Employee
where   EE
define  x1n = x1.name
        EE = exists x2
             in     Employee
             where x1n=x2n and x1<>x2
             define x2n = x2.name
```

The algebraic translation yields

$$\begin{aligned} q &\equiv \chi_{x1}(\sigma_{EE}(\chi_{EE:nq}(\chi_{x1n:x1.name}(Employee[x1])))) \\ nq &\equiv \exists_{x1n=x2n \wedge x1 \neq x2}(\chi_{x2n:x2.name}(Employee[x2])) \end{aligned}$$

The equivalence we use is given below. It works on existential quantifiers but the same can be defined for universal quantifiers.

$$\chi_{g:\exists_{A_1\theta A_2}(e_2))}(e_1)) \;\equiv\; \pi_{A_1:A_2}(\Gamma_{g;A_1\theta A_2;\exists_{true}(e_2)}(e_1)) \tag{49}$$

$$\text{if } A_i \subseteq \mathcal{A}(e_i),\ g \notin \mathcal{A}(e_1) \cup \mathcal{A}(e_2),\ \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset, \tag{50}$$

$$e_1 = \pi_{A_1:A_2}(e_2) \text{ (implies } A_1 = \mathcal{A}(e_1)),$$

The resulting query is then:

$$q \;\equiv\; \chi_{x1}(\sigma_{EE}(\pi_{x1n:x2n,x1:x2}(\Gamma_{EE;(=x2n,\neq x2);\exists_{true}}(\chi_{x2n:x2.name}(Employee[x2])))))$$

This query can be efficiently evaluated by, for instance, using an index on the employees name or by sorting the employees on their name and identifier.

5.3 Unnesting of d-Joins

For the χ-operator we already had a bunch of unnesting algebaic equivalences. These allowed us to reformulate the implicit loop by some kind of join for which several evaluation techniques (besides nested-loop join) exists. The following equivalence allows us to unnest the d-join:

$$e_1 < e_2 > \quad \equiv \quad \mu_g(\chi_{g:e_2}(e_1)) \tag{51}$$

The application of this equivalence generates a term containing a $\chi_{g:e}$-operator. Hence, we can subsequently apply the unnesting techniques of the previous sections. Although, by applying the unnest operator μ_g, the existing unnesting equivalences for χ_{g_e} can be somewhat simplified. More specifically, since tuples which have under the attribute g the empty set are eliminated, a regular join suffices, i.e., we can get rid of the outer join:

$$\begin{aligned} \mu_g(\chi_{g:f(\sigma_{A_1=A_2}(e_2))}(e_1))) \quad \equiv \quad & \mu_g(\pi_{\overline{A_2}}(e_1 \bowtie_{A_1=A_2} (\Gamma_{g;A_2;f}(e_2))))), \\ & \text{falls } A_i \subseteq \mathcal{A}(e_i), \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset, \\ & A_1 \cap A_2 = \emptyset, g \notin \mathcal{A}(e_1) \cup \mathcal{A}(e_2) \end{aligned} \tag{52}$$

The above equivalence is a modification of Eqvs.40.

5.4 Superscripts

The only reason, why terms of the form $\chi_{g:f(\sigma_p(e_2))}(e_1)$ and $\chi_{g:f(e_2)}(e_1)$ cannot be unnested maybe the failure of $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$ since $g \notin \mathcal{A}(e_1) \cup \mathcal{A}(e_2)$ can be made valid easily through the right choice of g. Terms which do not fulfill $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$ steam from queries, whose nesting depth is larger than two and which contain non-neighbor predicates. As an example consider

```
select r.a
from   r in R
where  r.b θ2
       count(select s
             from   s in S
             where  r.c = s.e and
                    s.f θ3
                    count(select t
                          from   t in T
                          where  s.g = t.h and
                                 r.d = t.i))
```

This query contains a non-neighbor predicate $r.d = t.i$ in the **where**-clause of the inner block. The introduction of the **define** clause results in

```
select  a
from    r in R
where   b θ2 c2
define  a = r.a
        b = r.b
        c = r.c
        d = r.d
        c2 = count(select  s
                   from    s in S
                   where   c = e and
                           f θ3 c3
                   define  e = s.e
                           f = s.f
                           g = s.g
                           c3 = count(select  t
                                      from    t in T
                                      where   g = h and
                                              d = i
                                      define  h = t.h
                                              i = t.i))
```

Translation into the algebra yields:

$$
\begin{aligned}
q &\equiv \pi_a(\sigma_{b\theta_1 c_2}(\chi_{c_2:e_2}(\chi_{a:r.a,b:r.b,c:r.c,d:r.d}(R[r]))))\\
e_2 &\equiv count(\pi_s(\sigma_{c=e}(\sigma_{f\theta_3 c_3}(\chi_{c_3:e_3}(\chi_{e:s.e,f:s.f,g:s.g}(S[s]))))))\\
e_3 &\equiv count(\pi_t(\sigma_{g=h}(\sigma_{d=i}(\chi_{h:t.h,i:t.i}(T[t])))))
\end{aligned}
$$

Unnesting of q and e_2 at $\chi_{c_2:e_2}$ and $\sigma_{c=e}$ is not possible with the equivalences introduced thus far. Every possible unnesting fails due to the failure of

$$\mathcal{F}(\sigma_{f\theta_3 c_3}(\chi_{c_3:e_3}(\chi_{e:s.e,f:s.f,g:s.g}(S[s])))) \cap \mathcal{A}(\chi_{a:r.a,b:r.b,c:r.c,d:r.d}(R[r])) = \emptyset$$

Since d is referenced by e_3, we have

$$
\begin{aligned}
\mathcal{F}(\sigma_{f\theta_3 c_3}(\chi_{c_3:e_3}(\chi_{e:s.e,f:s.f,g:s.g}(S[s])))) &= \{d\}\\
\mathcal{A}(\chi_{a:r.a,b:r.b,c:r.c,d:r.d}(R[r])) &= \{a, b, c, d, r\}
\end{aligned}
$$

The selection with the non-neighbor predicates $r.d = t.i$ hinders the unnesting process.

Also, complete unnesting of the term fails when starting with unnesting at $\chi_{c_3:e_3}$. To illustrate this, we apply Eqv 39 which results in

$$
\begin{aligned}
q &\equiv \pi_a(\sigma_{b\theta_1 c_2}(\chi_{c_2:e_2}(\chi_{a:r.a,b:r.b,c:r.c,d:r.d}(R[r])))) \\
e_2 &\equiv count(\pi_s(\sigma_{c=e}(\sigma_{f\theta_3 c_3}(\chi_{e:s.e,f:s.f,g:s.g}(S[s])\Gamma_{c_3;g=h;count\circ\pi_t}\sigma_{d=i}(\chi_{h:t.h,i:t.i}(T[t]))))))
\end{aligned}
$$

Then, unnesting stops since the condition

$$\mathcal{A}(\chi_{a:r.a,b:r.b,c:r.c,d:r.d}(R[r])) \cap \mathcal{F}(e_2) = \emptyset,$$

fails, or since the subterm $\sigma_{d=i}$ is no longer available for unnesting due to the fact that it is burried within a non-unary operator (here: binary grouping). It is easy to verify, that all unnesting stops at this point.

To remedy this situation, all algebraic operators are enhanced by a superscript containing a path expression similar to one used in the NF^2 data model to access nested relations. The superscripts allow a separation of operators and there arguments. For expressing that an operator f is applied to an argument e, one would normally write $f(e)$. With superscripts the inner operators can be moved to the outside:

$$f^{g.\alpha}(e) \quad = \quad \rho_{g:g'}(\pi_{\overline{g}}(\chi_{g':f^{\alpha}(g)}(e))) \tag{53}$$

For binary operators, superscripts only apply to the first argument:

$$e_1 op^{g.\alpha} e_2 \quad = \quad \rho_{g:g'}(\pi_{\overline{g}}(\chi_{g':gop^{\alpha}e_2}(e_1))) \tag{54}$$

Superscripts enable us to apply the regular unnesting equivalences. The general idea is to use superscripts in order to introduce more unnesting possibilities. This procedure consists of three different steps:

1. push disturbing operators to the outside
2. apply unnesting equivalences
3. push the operators back to the inside

where the last step is optional.

By using this trick, the example query can be unnested. The original term for the query is

$$
\begin{aligned}
q &\equiv \pi_a(\sigma_{b\theta_1 c_2}(\chi_{c_2:e_2}(\chi_{a:r.a,b:r.b,c:r.c,d:r.d}(R[r])))) \\
e_2 &\equiv count(\pi_s(\sigma_{c=e}(\sigma_{f\theta_3 c_3}(\chi_{c_3:e_3}(\chi_{e:s.e,f:s.f,g:s.g}(S[s])))))) \\
e_3 &\equiv count(\pi_t(\sigma_{g=h}(\sigma_{d=i}(\chi_{h:t.h,i:t.i}(T[t])))))
\end{aligned}
$$

Step 1 yields

$$
\begin{array}{rcl}
q & \equiv & \pi_a(\sigma_{b\theta_1 c_2}(\sigma^{c_2 \cdot c_3}_{d=i}(\chi_{c_2:e_2}(\chi_{a:r.a,b:r.b,c:r.c,d:r.d}(R[r])))) \\
e_2 & \equiv & count(\pi_s(\sigma_{c=e}(\sigma_{f\theta_3 c_3}(\chi_{c_3:e_3}(\chi_{e:s.e,f:s.f,g:s.g}(S[s])))))) \\
e_3 & \equiv & count(\pi_t(\sigma_{g=h}(\chi_{h:t.h,i:t.i}(T[t]))))
\end{array}
$$

We moved $\sigma_{d=i}$ to the outermost block. Hence, the unnesting at $\chi_{c_2:e_2}$ and $\sigma_{c=e}$ are no longer hindered by this selection.

Step 2 yields

$$
\begin{array}{rcl}
q & \equiv & \pi_a(\sigma_{b\theta_1 c_2}(\sigma^{c_2 \cdot c_3}_{d=i}(e_1 \Gamma_{c_2;c=e;count\circ\pi_s} e_2 \\
e_1 & \equiv & \chi_{a:r.a,b:r.b,c:r.c,d:r.d}(R[r]) \\
e_2 & \equiv & \sigma_{f\theta_3 c_3}(\chi_{c_3:e_3}(\chi_{e:s.e,f:s.f,g:s.g}(S[s]))) \\
e_3 & \equiv & count(\pi_t(\sigma_{g=h}(\chi_{h:t.h,i:t.i}(T[t]))))
\end{array}
$$

Also, $\chi_{c_3:e_3}$ and $\sigma_{g=h}$ can now be unnested.

$$
\begin{array}{rcl}
q & \equiv & \pi_a(\sigma_{b\theta_1 c_2}(\sigma^{c_2 \cdot c_3}_{d=i}(e_1 \Gamma_{c_2;c=e;count\circ\pi_s} e_2 \\
e_1 & \equiv & \chi_{a:r.a,b:r.b,c:r.c,d:r.d}(R[r]) \\
e_2 & \equiv & \sigma_{f\theta_3 c_3}(\chi_{e:s.e,f:s.f,g:s.g}(S[s]) \Gamma_{c_3;g=h;count\circ\pi_t} \chi_{h:t.h,i:t.i}(T[t]))
\end{array}
$$

The example term could have been unested if no second anchor $\sigma_{g=h}$ would have been present within the inner block. Without this selection, a term of the form $\chi_{f(e_2)}(e_1)$ would have been derived. Now, Eqv. 43 can be applied.

The next example shows the application of this equivalence. Therefore, we eliminate $s.g = t.h$ from the query. On the algebraic side this results in the vanishing of $\sigma_{g=h}$, $\chi_{h:t.h}$ and $\chi_{g:s.g}$.

$$
\begin{array}{rcl}
q & \equiv & \pi_a(\sigma_{b\theta_1 c_2}(\chi_{c_2:e_2}(\chi_{a:r.a,b:r.b,c:r.c,d:r.d}(R[r])))) \\
e_2 & \equiv & count(\pi_s(\sigma_{c=e}(\sigma_{f\theta_3 c_3}(\chi_{c_3:e_3}(\chi_{e:s.e,f:s.f}(S[s])))))) \\
e_3 & \equiv & count(\pi_t(\sigma_{d=i}(\chi_{i:t.i}(T[t]))))
\end{array}
$$

Again, direct unnesting is impossible. We apply the superscripts by moving all selections as far to the outside as possible.

$$
\begin{array}{rcl}
q & \equiv & \pi_a(\sigma_{b\theta_1 c_2}(count^{c_2}(\pi^{c_2}_s(\sigma^{c_2}_{c=e}(\sigma^{c_2}_{f\theta_3 c_3}(count^{c_2 \cdot c_3}(\pi^{c_2 \cdot c_3}_t(\sigma^{c_2 \cdot c_3}_{d=i}(\chi_{c_2:e_2}(e_1)))))))))) \\
e_1 & \equiv & \chi_{c_2:e_2}(\chi_{a:r.a,b:r.b,c:r.c,d:r.d}(R[r])) \\
e_2 & \equiv & \chi_{c_3:e_3}(\chi_{e:s.e,f:s.f}(S[s]))))) \\
e_3 & \equiv & \chi_{i:t.i}(T[t]))))
\end{array}
$$

Then, we apply Eqv. 43 at $\chi_{c_3:e_3}$ resulting in

$$\begin{array}{rcl} q & \equiv & \pi_a(\sigma_{b\theta_1 c_2}(count^{c_2}(\pi_s^{c_2}(\sigma_{c=e}^{c_2}(\sigma_{f\theta_3 c_3}^{c_2}(count^{c_2 \cdot c_3}(\pi_t^{c_2 \cdot c_3}(\sigma_{d=i}^{c_2 \cdot c_3}(e_1))))))))) \\ e_1 & \equiv & \chi_{c_2:e_2}(\chi_{a:r.a,b:r.b,c:r.c,d:r.d}(R[r])) \\ e_2 & \equiv & \chi_{e:s.e,f:s.f}(S[s])\Gamma_{c_3;;}\chi_{i:t.i}(T[t]) \end{array}$$

Now, we can unnest $\chi_{c_2:e_2}$ by applying Eqv. 43:

$$\begin{array}{rcl} q & \equiv & \pi_a(\sigma_{b\theta_1 c_2}(count^{c_2}(\pi_s^{c_2}(\sigma_{c=e}^{c_2}(\sigma_{f\theta_3 c_3}^{c_2}(count^{c_2 \cdot c_3}(\pi_t^{c_2 \cdot c_3}(\sigma_{d=i}^{c_2 \cdot c_3}(e_1))))))))) \\ e_1 & \equiv & \chi_{a:r.a,b:r.b,c:r.c,d:r.d}(R[r])\Gamma_{c_2;;}e_2 \\ e_2 & \equiv & \chi_{e:s.e,f:s.f}(S[s])\Gamma_{c_3;;}\chi_{i:t.i}(T[t]) \end{array}$$

Now, the term is unnested completely. The decision whether the operators are moved back to their original position is dependent on the resulting costs and, hence, a matter of the query optimizer.

More equivalences on superscripts can be found in [40]

6 Outer Restrictions

Outer restrictions are boolean expressions which occur within an inner block (typically in the **where**-clause) refering only to variables defined in outer blocks. If the outer restriction is satisfied, the block where the outer restriction occured can be evaluated ignoring the outer restriction. If the outer restriction evaluates to false, the result of the block with the outer restriction will be the empty set. Similarly to non-neighbor predicates, simple unnesting is impossible. Consider the following example:

```
select i                                  select i
from   i in Item                          from   i in Item
where  i in                               where  i in st
       (select d.store                    define iv = i.value
        from   d in Dept                         ip = i.procducedBy
        where  i.producedBy = d and              st = (select ds
               i.value > 100)                          from   d in Dept
                                                       where  ip = d and
                                                              iv > 100
                                                       define ds = d.store)
```

Within this example query, the predicate $iv > 100$ is an outer restriction. The translation of the query to the algebra yields:

$$\pi_i(\sigma_{i \in st}(\chi_{st:e_1}(\chi_{ip:i.producedBy}(\chi_{iv:i.value}(Item[i])))))$$

$$e_1 \equiv \pi_{ds}(\sigma_{ip=d}(\sigma_{iv>100}(\chi_{ds:d.store}(Dept[d]))))$$

None of the simple unnesting equivalence can be applied to the term $\chi_{st:e_1}$. The reason is that

$$\mathcal{F}(\sigma_{iv>100}(\chi_{ds:d.store}(Dept[d]))) \cap \mathcal{A}(\chi_{ip:i.producedBy}(\chi_{iv:i.value}(Item[i]))) = \emptyset$$

Opposedly, the more complex binary grouping can be applied. Eqv. 43 yields

$$\begin{array}{l} \pi_i(\sigma_{i\in st}(\chi_{ip:i.producedBy}(\chi_{iv:i.value}(Item[i]))) \\ \quad \Gamma_{st;;\pi_{ds}\circ\sigma_{ip=d}\circ\sigma_{iv>100}} \\ \chi_{ds:d.store}(Dept[d]) \end{array}$$

Also, the application of superscripts is possible to move the outer restriction to the outside enabling the application of the simple unnesting equivalences.

Yet another possibility to handle outer restriction is the usage of a special operator. The block containing an outer restriction is enclosed by an if where the if-operator is defined as follows: For some $f : \tau \to bool$.

$$if : \{\tau_1\} \to \{\tau_1\}$$

$$if_{f(e_2)}(e_1) \quad = \quad \begin{cases} e_1 & , \quad \text{if } f(e_2) = true \\ \emptyset & , \quad \text{sonst} \end{cases} \tag{55}$$

The *if* operator is only applied, if $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$.

[13] proposes to add the outer restriction to the predicate of some outer join. This method seems elegant at a first sight. But leads to the side effect that the block is also evaluated, if the outer restriction fails. If the outer restriction is satisfied, it is nevertheless evaluated for all the tuples involved in the outer join. Hence, we decided not to adopt this method.

Introduction of the if-operator into the above example yields:

$$\begin{array}{l} \pi_i(\sigma_{i\in st}(\chi_{st:e_1}(\chi_{ip:i.producedBy}(\chi_{iv:i.value}(Item[i]))))) \\ e_1 \equiv if_{iv>100}(\pi_{ds}(\sigma_{ip=d}(\chi_{ds:d.store}(Dept[d])))) \end{array}$$

Let us state that the if-operator is linear and reorderable. Hence, it does not hinder optimization or unnesting.

7 Conclusion

Opposed to the relational unnesting where unnesting is performed at the SQL-level, we have introduced an unnesting technique which allows unnesting at the algebraic level. For this purpose, an algebra capable of capturing all kinds of unnesting occurring in a query language as complex as OQL has been developed. The main idea to capture nested queries was to introduce subscripts which can hold arbitrary complex algebraic expressions. Given the fact that nested queries now result in nested algebraic expressions, unnesting boils down to moving expressions outside subscripts of algebraic operators. Several equivalences to do so have been introduced. The main advantages of this approach are

1. Unnesting strategies are expressed at the algebraic level, that is, they are independent of the query language chosen.

2. Correctness proofs of unnesting techniques become more feasible. This is an important issue witnessing all the tiny mistakes found in relational unnesting.

There remain two topics for further research. The first topic concerns the implementation. Therefor, good implementations for the extended operators have to be invented and existing cost models have to be extended to include these. A good starting point might be sorting based grouping operations. Special techniques like θ-tables [12] might be helpful.

The second topic involves the extension of the current approach in order to incorporate bags and lists. As already indicated, a simple strategy is coding of these data structures with sets but it still unclear whether this is the optimal approach.

Acknowledgements: The authors thank Serge Abiteboul and Victor Vianu for many valuable comments on a first draft of the paper.

References

[1] M. M. Astrahan and D. D. Chamberlin. Implementation of a structured English query language. *Communications of the ACM*, 18(10):580–588, 1975.

[2] F. Bancilhon, S. Cluet, and C. Delobel. A query language for the O_2 object-oriented database system. In *Proc. Int. Workshop on Database Programming Languages*, Salishan Lodge, Oregan, 1989.

[3] D. Beech. A foundation for evolution from relational to object databases. In *Proc. of the Int. Conf. on Extending Database Technology*, 1988.

[4] C. Beeri and Y. Kornatzky. Algebraic optimization of object-oriented query languages. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 72–88, 1990.

[5] J. Blakeley, W. McKenna, and G. Graefe. Experiences building the Open OODB query optimizer. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 287–295, 1993.

[6] M. Carey, D. DeWitt, and S. Vandenberg. A data model and query language for EXODUS. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 413–423, 1988.

[7] R. Cattell, editor. *The Object Database Standard: ODMG 93.* Morgan Kaufmann, 1993.

[8] S. Ceri and G. Gottlob. Translating SQL into relational algebra: Optimization, semantics and equivalence of SQL queries. *IEEE Trans. on Software Eng.*, pages 324–345, 1985.

[9] S. Chaudhuri and K. Shim. Query optimization in the presence of foreign functions. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 529–542, Dublin, Ireland, 1993.

[10] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 383–392, 1992.

[11] S. Cluet and G. Moerkotte. Nested queries in object bases. In *Proc. Int. Workshop on Database Programming Languages*, 1993.

[12] S. Cluet and G. Moerkotte. Efficient evaluation of aggregates on bulk types. Technical Report 95-5, RWTH-Aachen, 1995.

[13] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *VLDB*, pages 197–208, 1987.

[14] O. Deux. The story of O_2. *IEEE Trans. on Data and Knowledge Eng.*, 2(1):91–108, March 1990.

[15] D. DeWitt, J. Naughton, and D. Schneider. An evaluation of non-equijoin algorithms. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, page 443, Barcelona, Spain, 1991.

[16] G. Lohman et al. Optimization of nested queries in a distributed relational database. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 1984.

[17] R. Ganski and H. Wong. Optimization of nested SQL queries revisited. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 23–33, 1987.

[18] W. Hasan and H. Pirahesh. Query rewrite optimization in starburst. Research Report RJ6367, IBM, 1988.

[19] P. Jenq, D. Woelk, W. Kim, and W. Lee. Query processing in distributed ORION. In *Proc. Int. Conf. on Extended Database Technology (EDBT)*, pages 169–187, Venice, 1990.

[20] A. Kemper and G. Moerkotte. Advanced query processing in object bases using access support relations. In *Proc. Int. Conf. on Very Large Data Bases*, pages 294–305, 1990.

[21] A. Kemper and G. Moerkotte. *Object-Oriented Information Managment in Engineering Applications.* Prentice Hall, 1993.

[22] A. Kemper and G. Moerkotte. Query optimization in object bases: Exploiting relational techniques. In *Proc. Dagstuhl Workshop on Query Optimization (J.-C. Freytag, D. Maier und G. Vossen (eds.))*. Morgan-Kaufman, 1993.

[23] A. Kemper, G. Moerkotte, and K. Peithner. A blackboard architecture for query optimization in object bases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 543–554, 1993.

[24] W. Kiessling. SQL-like and Quel-like correlation queries with aggregates revisited. ERL/UCB Memo 84/75, University of Berkeley, 1984.

[25] W. Kim. A model of queries for object-oriented database. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 1989.

[26] W. Kim. On optimizing an SQL-like nested query. *ACM Trans. on Database Systems*, 7(3):443–469, Sep 82.

[27] G. Mitchell. *Extensible Query Processing in an Object-Oriented Database*. PhD thesis, Brown University, Providence, RI 02912, 1993.

[28] G. Mitchell, S. Zdonik, and U. Dayal. Object-oriented query optimization: What's the problem? Technical Report CS-91-41, Brown University, 1991.

[29] G. Mitchell, S. Zdonik, and U. Dayal. *Object-Oriented Database Systems (A. Dogac, M. T. Özsu, A. Biliris, and T. Sellis (eds.))*, chapter Optimization of Object-Oriented Queries: Problems and Applications. NATO ASI. Springer, 1993. to appear.

[30] M. Muralikrishna. Optimization and dataflow algorithms for nested tree queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 1989.

[31] M. Muralikrishna. Improved unnesting algorithms for join aggregate SQL queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 91–102, 1992.

[32] A. Rosenthal and C. Galindo-Legaria. Query graphs, implementing trees, and freely-reorderable outerjoins. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 291–299, 1990.

[33] M. Roth, H. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Trans. on Database Systems*, 13(4):389–417, 1988.

[34] H.-J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.

[35] G.M. Shaw and S.B. Zdonik. A query algebra for object-oriented databases. In *Proc. IEEE Conference on Data Engineering*, pages 154–162, 1990.

[36] H. Steenhaben, P. Apers, H. Blanken, and R. de By. From nested-loop to join queries in oodb. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 618–629, 1994.

[37] D. Straube and T. Özsu. Queries and query processing in object-oriented database systems. *ACM Trans. on Information Systems*, 8(4):387–430, 1990.

[38] S. L. Vandenberg and D. DeWitt. Algebraic support for complex objects with arrays, identity, and inheritance. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 158–167, 1991.

[39] G. von Bültzingsloewen. *SQL-Anfragen: Optimierung für parallele Bearbeitung*. FZI-Berichte Informatik. Springer, 1991.

[40] S. Voss. Nested Queries in Object Bases. Master thesis, University of Karlsruhe, 1994. in German.

[41] W. Yan and P.-A. Larson. Performing group-by before join. Technical Report CS 93-46, Dept. of Computer Science, University of Waterloo, Canada, 1993.

[42] W. Yan and P.-A. Larson. Performing group-by before join. In *Proc. IEEE Conference on Data Engineering*, pages 89–100, Houston, TX, Feb. 1994.

[43] W. Yan and P.-A. Larson. Interchanging the order of grouping and join. Technical Report CS 95-09, Dept. of Computer Science, University of Waterloo, Canada, 1995.

A Proofs of Selected Equivalences

This appendix contains proofs of selected unnesting equations.

Eqv. 34: $e_2 \equiv \pi_{a_1}(e_1) \succ$

$$f(\sigma_{a_1=m}(e_1)[m : agg(e_2)]) \equiv (agg_{g;m;a_1;f}(e_1)).g$$

Proof

$$\begin{aligned}
lhs &\equiv f(\sigma_{a_1=agg(e_2)}(e_1)) \\
&\equiv f(\sigma_{a_1=agg(\{x'.a_1|x'\in e_1\})}(e_1)) \\
&\equiv f(\{x|x \in e_1, x.a_1 = agg(\{x'.a_1|x' \in e_1\})\}) \\
&\equiv (agg_{g;m;a_1;f}(e_1)).g \\
&\equiv rhs
\end{aligned}$$

$\square$

Eqv. 41:

$$\begin{aligned}
\chi_{g:f(\sigma_{A_1\theta A_2}(e_2))}(e_1) \equiv\ & \pi_{A1:A2}(\Gamma_{g;A_2\theta;f}(e_2)) \\
& \text{if} e_1 = \pi_{A_1:A_2}(e_2), \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset, \\
& A_i \subseteq \mathcal{A}(e_i), g \notin \mathcal{A}(e_1) \cup \mathcal{A}(e_2)
\end{aligned}$$

Proof

$$\begin{aligned}
lhs &\equiv \{y \circ [g : G]|y \in e_1, G = f(\{x|x \in e_2, y.[A_1]\theta x.[A_2]\})\} \\
&\equiv \{y \circ [g : G]|y \in \pi_{A_1:A_2}(e_2), G = f(\{x|x \in e_2, y.[A_1]\theta x.[A_1]\})\} \\
&\equiv \{y.[A_1 : A_2] \circ [g : G]|y \in e_2, G = f(\{x|x \in e_2, y.[A_2]\theta x.[A_2]\})\} \\
&\equiv \pi_{A_1:A_2}(\{y \circ [g : G]|y \in e_2, G = f(\{x|x \in e_2, y.[A_2]\theta x.[A_2]\})\}) \\
&\equiv rhs
\end{aligned}$$

$\square$

Eqv. 40:

$$\begin{aligned}
\chi_{g:f(\sigma_{A_1=A_2}(e_2))}(e_1) \equiv\ & \pi_{\overline{A_2}}(e_1 \sqsupset\!\!\bowtie^{g=f(\emptyset)}_{A_1=A_2} (\Gamma_{g;A_2;f}(e_2))) \\
& \text{if} A_i \subseteq \mathcal{A}(e_i), \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset, \\
& A_1 \cap A_2 = \emptyset, g \notin \mathcal{A}(e_1) \cup \mathcal{A}(e_2)
\end{aligned}$$

Proof

$$\begin{aligned}
rhs \equiv\ & \{z.[\overline{A_2}] | z \in \{x \circ y'' | x \in e_1, x.[A_1] = y''.[A_2], y'' \in \{y'.[A_2] \circ [g : G] | y' \in e_2, \\
& G = f(\{y | y \in e_2, y.[A_2] = y'.[A_2]\})\}\}\} \\
& \cup \\
& \{z.[\overline{A_2}] | z \in \{x \circ y' | x \in e_1, \not\exists y' \in e_2\ x.[A_1] = y'.[A_2], y'.g = f(\emptyset), y.[\overline{g}] = NULL\}\} \\
\equiv\ & \{(x \circ y'').[\overline{A_2}] | x \in e_1, x.[A_1] = y''.[A_2], y'' \in \{y'.[A_2] \circ [g : G] | y' \in e_2, \\
& G = f(\{y | y \in e_2, y.[A_2] = y'.[A_2]\})\}\} \\
& \cup \\
& \{z.[\overline{A_2}] | z \in \{x \circ y' | x \in e_1, \not\exists y' \in e_2\ x.[A_1] = y'.[A_2], y'.g = f(\emptyset), y'.[\overline{g}] = NULL\}\} \\
\equiv\ & \{(x \circ (y'.[A_2] \circ [g : G])).[\overline{A_2}] | x \in e_1, y' \in e_2, x.[A_1] = y'.[A_2], \\
& G = f(\{y | y \in e_2, y.[A_2] = y'.[A_2]\})\} \\
& \cup \\
& \{z.[\overline{A_2}] | z \in \{x \circ y' | x \in e_1, \not\exists y' \in e_2\ x.[A_1] = y'.[A_2], y'.g = f(\emptyset), y'.[\overline{g}] = NULL\}\} \\
\equiv\ & \{x \circ [g : G] | x \in e_1, y' \in e_2, x.[A_1] = y'.[A_2], G = f(\{y | y \in e_2, y.[A_2] = y'.[A_2]\})\} \\
& \cup \\
& \{z.[\overline{A_2}] | z \in \{x \circ y' | x \in e_1, \not\exists y' \in e_2\ x.[A_1] = y'.[A_2], y'.g = f(\emptyset), y'.[\overline{g}] = NULL\}\} \\
\equiv\ & \{x \circ [g : G] | x \in e_1, G = f(\{y | y \in e_2, y.[A_2] = x.[A_1]\})\} \\
\equiv\ & lhs
\end{aligned}$$

□

Eqv. 51:

$$e_1 < e_2 > \quad \equiv \quad \mu_g(\chi_{g:e_2}(e_1))$$

Proof

$$\begin{aligned}
\mu_g(\chi_{g:e_2}(e_1)) &\equiv \mu_g(\{y \circ [g : e_2(y)] | y \in e_1\}) \\
&\equiv \mu_g(\{y \circ [g : G] | y \in e_1, G = e_2(y)\}) \\
&\equiv \{y \circ x | y \in e_1, x \in G, G = e_2(y)\} \\
&\equiv \{y \circ x | y \in e_1, x \in e_2(y)\} \\
&\equiv e_1 < e_2 >
\end{aligned}$$

□

Eqv. 39:

$$\begin{aligned}
\chi_{g:f(\sigma_{A_1 \theta A_2}(e_2))}(e_1) \quad \equiv \quad & e_1 \Gamma_{g;A_1 \theta A_2;f} e_2 \\
& \text{if} A_1 \subseteq \mathcal{A}(e_i), g \notin \mathcal{A}(e_1) \cup \mathcal{A}(e_2), \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset
\end{aligned}$$

Proof

$$
\begin{aligned}
rhs &\equiv \{y \circ [g : G] | y \in e_1, G = f(\{x | x \in e_2, y.[A_1]\theta x.[A_2]\})\} \\
&\equiv \{y \circ [g : G] | y \in e_1, G = f(\sigma_{A_1 \theta A_2}(\{x | x \in e_2\}))\} \\
&\quad \text{, wobei } A_i \subseteq \mathcal{A}(e_i) \\
&\equiv \{y \circ [g : G] | y \in e_1, G = f(\sigma_{A_1 \theta A_2}(e_2))\} \\
&\equiv \{y \circ [g : f(\sigma_{A_1 \theta A_2}(e_2))] | y \in e_1\} \\
&\equiv lhs \text{if} A_i \subseteq \mathcal{A}(e_i)
\end{aligned}
$$

$\square$

Eqv. 42:

$$
\begin{aligned}
\chi_{g:f(\sigma_{A_1 \theta A_2}(e_2))}(e_1) &= \Gamma_{g;=\mathcal{A}(e_1);f \circ \pi_{\overline{\mathcal{A}(e_1)}} \circ \sigma_{\mathcal{A}(e_2) \neq \perp_{\mathcal{A}(e_2)}}}(e_1 \sqsupset\!\!\bowtie_{A_1 \theta A_2} e_2) \\
&\quad \text{if} A_i \subseteq \mathcal{A}(e_i), \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset, \\
&\quad A_1 \cap A_2 = \emptyset, g \notin \mathcal{A}(e_1) \cup \mathcal{A}(e_2)
\end{aligned}
$$

Proof

$$
\begin{aligned}
lhs &\equiv \{x \circ [g : G] | x \in e_1, G = f(\{y | y \in e_2, x.[A_1']\theta y.[A_2']\})\} \\
&\equiv \{x \circ [g : G] | x \in e_1, \\
&\quad G = f(\pi_{\overline{A_1}}(\{x \circ y | x' \in e_1, x' = x, y \in e_2, x.[A_1']\theta y.[A_2']\}))\} \\
&\equiv \{x \circ [g : G] | x \in e_1, \\
&\quad G = f(\pi_{\overline{A_1}}(\{x \circ y | x' \in e_1, y \in e_2, x.[A_1']\theta y.[A_2'], x.[A_1] = x'.[A_1]\}))\} \\
&\equiv \{x \circ [g : G] | x \in e_1, \\
&\quad G = f(\pi_{\overline{A_1}}(\{x \circ y | x' \in e_1, y \in e_2, x.[A_1']\theta y.[A_2'], x.[A_1] = (x' \circ y).[A_1]\}))\} \\
&\equiv \{x \circ [g : G] | x \in e_1, \\
&\quad G = f(\pi_{\overline{A_1}}(\{u | u \in \{x \circ y | x \in e_1, y \in e_2, x.[A_1']\theta y.[A_2']\}, x.[A_1] = u.[A_1]\}))\} \\
&\equiv \{x \circ [g : G] | x \in e_1, G = f(\pi_{\overline{A_1}}(\{u | u \in V, x.[A_1] = u.[A_1]\}))\} \\
&\quad \text{where} \\
&\quad V = \{x \circ y | x \in e_1, y \in e_2, x.[A_1']\theta y.[A_2']\} \\
&\equiv \{x \circ [g : G] | x \in e_1, \exists y \in e_2\ x.[A_1']\theta y.[A_2'],
\end{aligned}
$$

$G = f(\pi_{\overline{A_1}}(\{u|u \in V, x.[A_1] = u.[A_1]\}))\}$
$\cup$
$\{x \circ [g : G]|x \in e_1, \nexists y \in e_2\ x.[A_1']\theta y.[A_2'],$
$G = f(\pi_{\overline{A_1}}(\{u|u \in V, x.[A_1] = u.[A_1]\}))\}$
where
$V = \{x \circ y|x \in e_1, y \in e_2, x.[A_1']\theta y.[A_2']\}$

$\equiv$ $\{(x \circ y).[\overline{A_2}] \circ [g : G]|x \in e_1, y \in e_2, x.[A_1']\theta y.[A_2'],$
$G = f(\pi_{\overline{A_1}}(\{u|u \in V, (x \circ y).[A_1] = u.[A_1]\}))\}$
$\cup$
$\{(x \circ z).[\overline{A_2}] \circ [g : G]|x \in e_1, \nexists y \in e_2\ x.[A_1']\theta y.[A_2'], z = \perp_{A_2},$
$G = f(\pi_{\overline{A_1}}(\{u|u \in V, (x \circ z).[A_1] = u.[A_1]\}))\}$
where
$V = \{x \circ y|x \in e_1, y \in e_2, x.[A_1']\theta y.[A_2']\}$

$\equiv$ $\{v.[\overline{A_2}] \circ [g : G]|v \in \{x \circ y|x \in e_1, y \in e_2, x.[A_1']\theta y.[A_2']\},$
$G = f(\pi_{\overline{A_1}}(\{u|u \in V, v.[A_1] = u.[A_1]\}))\}$
$\cup$
$\{v.[\overline{A_2}] \circ [g : G]|v \in \{x \circ z|x \in e_1, \nexists y \in e_2\ x.[A_1']\theta y.[A_2'], z = \perp_{A_2}\},$
$G = f(\pi_{\overline{A_1}}(\{u|u \in V, v.[A_1] = u.[A_1]\}))\}$
where
$V = \{x \circ y|x \in e_1, y \in e_2, x.[A_1']\theta y.[A_2']\}$

$\equiv$ $\{v.[\overline{A_2}] \circ [g : G]|v \in V, G = f(\pi_{\overline{A_1}}(\{u|u \in V, v.[A_1] = u.[A_1]\}))\}$
$\cup$
$\{v.[\overline{A_2}] \circ [g : G]|v \in V^\perp, G = f(\pi_{\overline{A_1}}(\{u|u \in V, v.[A_1] = u.[A_1]\}))\}$
where
$V = \{x \circ y|x \in e_1, y \in e_2, x.[A_1']\theta y.[A_2']\}$
and
$V^\perp = \{x \circ z|x \in e_1, \nexists y \in e_2\ x.[A_1']\theta y.[A_2'], z = \perp_{A_2}\}$

$\equiv$ $\{v.[\overline{A_2}] \circ [g : G]|v \in V \cup V^\perp, G = f(\pi_{\overline{A_1}}(\{u|u \in V, v.[A_1] = u.[A_1]\}))\}$
where
$V = \{x \circ y|x \in e_1, y \in e_2, x.[A_1']\theta y.[A_2']\}$
and
$V^\perp = \{x \circ z|x \in e_1, \nexists y \in e_2\ x.[A_1']\theta y.[A_2'], z = \perp_{A_2}\}$

$$\begin{aligned}
\equiv\quad & \{v.[\overline{A_2}] \circ [g : G] | v \in V \cup V^{\perp}, \\
& G = f(\pi_{\overline{A_1}}(\sigma_{A_2 \neq \perp_{A_2}}(\{u | u \in V \cup V^{\perp}, v.[A_1] = u.[A_1]\})))\} \\
& \text{where} \\
& V = \{x \circ y | x \in e_1, y \in e_2, x.[A_1']\theta y.[A_2']\} \\
& \text{and} \\
& V^{\perp} = \{x \circ z | x \in e_1, \not\exists y \in e_2\ x.[A_1']\theta y.[A_2'], z = \perp_{A_2}\} \\
\equiv\quad & rhs
\end{aligned}$$

□

Eqv. 52

$$\begin{aligned}
\mu_g(\chi_{g:f(\sigma_{A_1=A_2}(e_2))}(e_1)) \quad \equiv \quad & \mu_g(\pi_{\overline{A_2}}(e_1 \bowtie_{A_1=A_2} (\Gamma_{g;A_2;f}(e_2))))) \\
& \text{,if } A_i \subseteq \mathcal{A}(e_i), \\
& \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset, \\
& g \notin \mathcal{A}(e_1) \cup \mathcal{A}(e_2)
\end{aligned}$$

Proof

$$\begin{aligned}
& \mu_g(\chi_{g:f(\sigma_{A_1=A_2}(e_2))}(e_1)) \\
\equiv\quad & \mu_g(\{z \circ [g : f(\sigma_{A_1=A_2}(e_2))] | z \in e_1\}) \\
\equiv\quad & \mu_g(\{z \circ [g : f(\{x | x \in e_2, z.[A_1] = x.[A_2]\})] | z \in e_1\}) \\
\equiv\quad & \{z \circ w | z \in e_1, w \in f(\{x | x \in e_2, z.[A_1] = x.[A_2]\})\}
\end{aligned}$$

$$\begin{aligned}
& \mu_g(\pi_{\overline{A_2}}(e_1 \bowtie_{A_1=A_2} (\Gamma_{g;A_2;f}(e_2))))) \\
\equiv\quad & \mu_g(\pi_{\overline{A_2}}(e_1 \bowtie_{A_1=A_2} \{y.[A_2] \circ [g : G] | y \in e_2, G = f(\{x | x \in e_2, x.[A_2] = y.[A_2]\})\})) \\
\equiv\quad & \mu_g(\pi_{\overline{A_2}}(\{z \circ y.[A_2] \circ [g : G] | z \in e_1, y \in e_2, z.[A_1] = y.[A_2], \\
& G = f(\{x | x \in e_2, x.[A_2] = y.[A_2]\})\})) \\
\equiv\quad & \mu_g(\{z \circ [g : G] | z \in e_1, \exists y \in e_2, z.[A_1] = y.[A_2], G = f(\{x | x \in e_2, x.[A_2] = y.[A_2]\})\}) \\
\equiv\quad & \{z \circ w | z \in e_1, \exists y \in e_2, z.[A_1] = y.[A_2], w \in f(\{x | x \in e_2, x.[A_2] = y.[A_2]\})\} \\
\equiv\quad & \{z \circ w | z \in e_1, w \in f(\{x | x \in e_2, x.[A_2] = z.[A_1]\})\}
\end{aligned}$$

□

Binary Grouping is linear in the first argument:

$$\begin{aligned}
(e_1 \cup e_2)\Gamma_{g;A_1\theta A_2;f}e_3 &= (e_1\Gamma_{g;A_1\theta A_2;f}e_3) \cup (e_2\Gamma_{g;A_1\theta A_2;f}e_3) \\
\emptyset\Gamma_{g;A_1\theta A_2;f}e_3 &= \emptyset
\end{aligned}$$

Proof

$$\begin{aligned}
& (e_1 \cup e_2)\Gamma_{g;A_1\theta A_2;f} e_3 \\
\equiv\ & \{y \circ [g : G] | y \in e_1 \cup e_2, G = f(\{x|x \in e_3, y.[A_1]\theta x.[A_2]\})\} \\
\equiv\ & \{y \circ [g : G] | y \in e_1, G = f(\{x|x \in e_3, y.[A_1]\theta x.[A_2]\})\} \cup \\
& \{y \circ [g : G] | y \in e_2, G = f(\{x|x \in e_3, y.[A_1]\theta x.[A_2]\})\} \\
\equiv\ & (e_1\Gamma_{g;A_1\theta A_2;f} e_3) \cup (e_2\Gamma_{g;A_1\theta A_2;f} e_3)
\end{aligned}$$

$$\begin{aligned}
& \emptyset\Gamma_{g;A_1\theta A_2;f} e_3 \\
\equiv\ & \{y \circ [g : G] | y \in \emptyset, G = f(\{x|x \in e_3, y.[A_1]\theta x.[A_2]\})\} \\
\equiv\ & \emptyset
\end{aligned}$$

□

Next, the equations for superscripts follow:

$$e_1\Gamma_{g_2;A_1\theta A_2}(e_2\Gamma_{g_3;A_2\theta A_3} e_3) \equiv (e_1\Gamma_{g_2;A_1\theta A_2} e_2)\Gamma^{g_2}_{g_3;A_2\theta A_3} e_3 \tag{56}$$

Proof

$$\begin{aligned}
rhs \equiv\ & \rho_{g_2:g_2'}(\pi_{\overline{g_2}}(\chi_{g_2':g_2\Gamma_{g_3;A_2\theta A_3} e_3}(e_1\Gamma_{g_2;A_1\theta A_2} e_2))) \\
\equiv\ & \rho_{g_2:g_2'}(\pi_{\overline{g_2}}(\chi_{g_2':g_2\Gamma_{g_3;A_2\theta A_3} e_3}(\{y \circ [g_2 : G_2] | y \in e_1, G_2 = \{x|x \in e_2, y.[A_1]\theta x.[A_2]\}\}))) \\
\equiv\ & \rho_{g_2:g_2'}(\pi_{\overline{g_2}}(\{z \circ [g_2' : G_2\Gamma_{g_3;A_2\theta A_3} e_3]| \\
& z \in \{y \circ [g_2 : G_2] | y \in e_1, G_2 = \{x|x \in e_2, y.[A_1]\theta x.[A_2]\}\}\})) \\
\equiv\ & \rho_{g_2:g_2'}(\pi_{\overline{g_2}}(\{y \circ [g_2 : G_2] \circ [g_2' : G_2\Gamma_{g_3;A_2\theta A_3} e_3] | y \in e_1, \\
& G_2 = \{x|x \in e_2, y.[A_1]\theta x.[A_2]\}\})) \\
\equiv\ & \rho_{g_2:g_2'}(\{y \circ [g_2' : G_2\Gamma_{g_3;A_2\theta A_3} e_3] | y \in e_1, G_2 = \{x|x \in e_2, y.[A_1]\theta x.[A_2]\}\}) \\
\equiv\ & \{y \circ [g_2 : G_2\Gamma_{g_3;A_2\theta A_3} e_3] | y \in e_1, G_2 = \{x|x \in e_2, y.[A_1]\theta x.[A_2]\}\} \\
\equiv\ & \{y \circ [g_2 : \{w \circ [g_3 : G_3] | w \in G_2, G_3 = \{v|v \in e_3, w.[A_2]\theta v.[A_3]\}\}] | y \in e_1, \\
& G_2 = \{x|x \in e_2, y.[A_1]\theta x.[A_2]\}\} \\
\equiv\ & \{y \circ [g_2 : \{w \circ [g_3 : G_3]\}] | y \in e_1, w \in G_2, G_2 = \{x|x \in e_2, y.[A_1]\theta x.[A_2]\}, \\
& G_3 = \{v|v \in e_3, w.[A_2]\theta v.[A_3]\}\}
\end{aligned}$$

$$\begin{aligned}
lhs \equiv\ & e_1\Gamma_{g_2;A_1\theta A_2}\{z \circ [g_3 : G_3] | z \in e_2, \\
& G_3 = \{v|v \in e_3, z.[A_2]\theta v.[A_3]\}\} \\
\equiv\ & \{y \circ [g_2 : G_2] | y \in e_1, G_2 = \{x|x \in \{z \circ [g_3 : G_3] | z \in e_2, \\
& G_3 = \{v|v \in e_3, z.[A_2]\theta v.[A_3]\}\}, y.[A_1]\theta x.[A_2]\}\}
\end{aligned}$$

$$
\begin{aligned}
&\equiv \{y \circ [g_2 : G_2] | y \in e_1, G_2 = \{z \circ [g_3 : G_3] | z \in e_2, \\
&\quad G_3 = \{v | v \in e_3, z.[A_2]\theta v.[A_3]\}, y.[A_1]\theta z \circ [g_3 : G_3].[A_2]\}\} \\
&\equiv \{y \circ [g_2 : \{z \circ [g_3 : G_3]\}] | y \in e_1, z \in e_2, \\
&\quad G_3 = \{v | v \in e_3, z.[A_2]\theta v.[A_3]\}, y.[A_1]\theta z \circ [g_3 : G_3].[A_2]\} \\
&\equiv rhs
\end{aligned}
$$

$\square$

$$
e_1 \Gamma_{g_1;A_1\theta A_2}(e_2 \Gamma^{\alpha_2}_{g_2;A_2\theta A_3} e_3) \equiv (e_1 \Gamma_{g_1;A_1\theta A_2} e_2) \Gamma^{g_1.\alpha_2}_{g_2;A_2\theta A_3} e_3 \tag{57}
$$

Proof

$$
lhs \equiv \{y \circ [g_1 : G_1] | y \in e_1, G_1 = \{x | x \in \{e_2 \Gamma^{\alpha_2}_{g_2;A_2\theta A_3} e_3\}, A_1\theta A_2\}\}
$$

$$
\begin{aligned}
rhs &\equiv \{y \circ [g_1 : G_1] | y \in e_1, G_1 = \{x | x \in e_2, y.[A_1]\theta x.[A_2]\}\} \Gamma^{g_1.\alpha_2}_{g_2,A_2\theta A_3} e_3 \\
&\equiv \rho_{g_1:g_1'}(\pi_{\overline{g_1}}(\chi_{g_1':g_1 \Gamma^{\alpha_2}_{g_2,A_2\theta A_3} e_3}(\{y \circ [g_1 : G_1] | y \in e_1, G_1 = \{x | x \in e_2, y.[A_1]\theta x.[A_2]\}\}))) \\
&\equiv \rho_{g_1:g_1'}(\pi_{\overline{g_1}}(\{y \circ [g_1 : G_1] \circ [g_1' : G_1'] | y \in e_1, G_1 = \{x | x \in e_2, y.[A_1]\theta x.[A_2]\}, \\
&\quad G_1' = \{G_1 \Gamma^{\alpha_2}_{g_2,A_2\theta A_3} e_3\}\})) \\
&\equiv \rho_{g_1:g_1'}(\{y \circ [g_1' : G_1'] | y \in e_1, G_1 = \{x | x \in e_2, y.[A_1]\theta x.[A_2]\}, G_1' = \{G_1 \Gamma^{\alpha_2}_{g_2,A_2\theta A_3} e_3\}\}) \\
&\equiv \{y \circ [g_1 : G_1'] | y \in e_1, G_1 = \{x | x \in e_2, y.[A_1]\theta x.[A_2]\}, G_1' = \{G_1 \Gamma^{\alpha_2}_{g_2,A_2\theta A_3} e_3\}\} \\
&\equiv lhs \text{if } A_i \subseteq \mathcal{A}(e_i)
\end{aligned}
$$

$\square$

$$
\chi_{g:f(e_2)}(e_1) \equiv f^g(\chi_{g:e_2}(e_1)) \tag{58}
$$

Proof

$$
lhs \equiv \{y \circ [g : f(e_2(y))] | y \in e_1\}
$$

$$
\begin{aligned}
rhs &\equiv \rho_{g:g'}(\pi_{\overline{g}}(\chi_{g':f(g)}(\chi_{g:e_2}(e_1)))) \\
&\equiv \rho_{g:g'}(\pi_{\overline{g}}(\chi_{g':f(g)}(\{y \circ [g : e_2(y)] | y \in e_1\}))) \\
&\equiv \rho_{g:g'}(\pi_{\overline{g}}(\{y \circ [g : e_2(y)] \circ [g' : f(e_2(y))] | y \in e_1\})) \\
&\equiv \rho_{g:g'}(\{y \circ [g' : f(e_2(y))] | y \in e_1\}) \\
&\equiv \{y \circ [g : f(e_2(y))] | y \in e_1\} \\
&\equiv lhs
\end{aligned}
$$

□

$$\Gamma_{g;\theta A;f_1 \circ f_2}(e) \equiv f_1^g(\Gamma_{g;\theta A;f_2}(e)) \tag{59}$$

Proof

$$lhs \equiv \{y.[A] \circ [g : G] | y \in e, G = f_1(f_2(\{x | x \in e, x.[A] \theta y.[A]\}))\}$$

$$\begin{aligned}
rhs &\equiv f_1^g(\{y.[A] \circ [g : G'] | y \in e, G' = f_2(\{x | x \in e, x.[A] \theta y.[A]\})\}) \\
&\equiv \rho_{g:g'}(\pi_{\overline{g}}(\chi_{g':f_1(g)}(\{y.[A] \circ [g : G'] | y \in e, G' = f_2(\{x | x \in e, x.[A] \theta y.[A]\})\}))) \\
&\equiv \rho_{g:g'}(\pi_{\overline{g}}(\{y.[A] \circ [g : G'] \circ [g' : f_1(G')] | y \in e, G' = f_2(\{x | x \in e, x.[A] \theta y.[A]\})\})) \\
&\equiv \rho_{g:g'}(\{y.[A] \circ [g' : f_1(G')] | y \in e, G' = f_2(\{x | x \in e, x.[A] \theta y.[A]\})\}) \\
&\equiv \rho_{g:g'}(\{y.[A] \circ [g' : G] | y \in e, G = f_1(f_2(\{x | x \in e, x.[A] \theta y.[A]\}))\}) \\
&\equiv \{y.[A] \circ [g : G] | y \in e, G = f_1(f_2(\{x | x \in e, x.[A] \theta y.[A]\}))\} \\
&\equiv lhs
\end{aligned}$$

□

$$e_1 \Gamma_{g;A_1 \theta A_2;f_1 \circ f_2} e_2 \equiv f_1^g(e_1 \Gamma_{g;A_1 \theta A_2;f_2} e_2) \tag{60}$$

Proof

$$lhs \equiv \{y \circ [g : G] | y \in e_1, G = f_1(f_2(\{x | x \in e_2, y.[A_1] \theta x.[A_2]\}))\}$$

$$\begin{aligned}
rhs &\equiv f_1^g(\{y \circ [g : G'] | y \in e_1, G' = f_2(\{x | x \in e_2, y.[A_1] \theta x.[A_1]\})\}) \\
&\equiv \rho_{g:g'}(\pi_{\overline{g}}(\chi_{g':f_1(g)}(\{y \circ [g : G'] | y \in e_1, G' = f_2(\{x | x \in e_2, y.[A_1] \theta x.[A_1]\})\}))) \\
&\equiv \rho_{g:g'}(\pi_{\overline{g}}(\{y \circ [g : G'] \circ [g' : f_1(G')] | y \in e_1, G' = f_2(\{x | x \in e_2, y.[A_1] \theta x.[A_1]\})\})) \\
&\equiv \rho_{g:g'}(\{y \circ [g' : f_1(G')] | y \in e_1, G' = f_2(\{x | x \in e_2, y.[A_1] \theta x.[A_1]\})\}) \\
&\equiv \{y \circ [g : f_1(G')] | y \in e_1, G' = f_2(\{x | x \in e_2, y.[A_1] \theta x.[A_1]\})\} \\
&\equiv \{y \circ [g : G] | y \in e_1, G = f_1(f_2(\{x | x \in e_2, y.[A_1] \theta x.[A_1]\}))\} \\
&\equiv lhs
\end{aligned}$$

□

$$\sigma_p^g(e_1 \Gamma_{g;A_1 \theta A_2;f} e_2) \equiv e_1 \Gamma_{g;A_1 \theta A_2,p;f} e_2 \tag{61}$$

Proof

$$rhs \equiv \{y \circ [g : G] | y \in e_1, G = f(\{x | x \in e_2, y.[A_1]\theta x.[A_2], p\})\}$$

$$\begin{aligned}
lhs &\equiv \sigma_p^g(\{y \circ [g : G'] | y \in e_1, G' = f(\{x | x \in e_2, y.[A_1]\theta x.[A_2]\})\}) \\
&\equiv \rho_{g:g'}(\pi_{\overline{g}}(\chi_{g':\sigma_p(g)}(\{y \circ [g : G'] | y \in e_1, G' = f(\{x | x \in e_2, y.[A_1]\theta x.[A_2]\})\}))) \\
&\equiv \rho_{g:g'}(\pi_{\overline{g}}(\{y \circ [g : G'] \circ [g' : \sigma_p(G')] | y \in e_1, G' = f(\{x | x \in e_2, y.[A_1]\theta x.[A_2]\})\})) \\
&\equiv \rho_{g:g'}(\{y \circ [g' : \sigma_p(G')] | y \in e_1, G' = f(\{x | x \in e_2, y.[A_1]\theta x.[A_2]\})\}) \\
&\equiv \{y \circ [g : \sigma_p(G')] | y \in e_1, G' = f(\{x | x \in e_2, y.[A_1]\theta x.[A_2]\})\} \\
&\equiv \{y \circ [g : G''] | y \in e_1, G'' = \sigma_p(f(\{x | x \in e_2, y.[A_1]\theta x.[A_2]\}))\} \\
&\equiv \{y \circ [g : G''] | y \in e_1, G'' = f(\sigma_p(\{x | x \in e_2, y.[A_1]\theta x.[A_2]\}))\} \\
&\quad \text{if } f \text{ and } \sigma_p \text{ reorderable} \\
&\equiv \{y \circ [g : G] | y \in e_1, G = f(\{x | x \in e_2, y.[A_1]\theta x.[A_2], p\})\} \\
&\equiv rhs
\end{aligned}$$

□

$$\chi^{\alpha}_{g:f(e_2)}(e_1) \equiv f^{\alpha.g}(\chi^{\alpha}_{g:e_2}(e_1)) \tag{62}$$

Proof by induction over $|\alpha|$.

induction start: $|\alpha| = 0 \quad \surd$

presupposition: Equivalence holds for arbitraty α with $|\alpha| = n$.

induction step: $n + 1 \rightsquigarrow n$

Let $\alpha' = \alpha_1.\alpha$

$$\begin{aligned}
& \chi^{\alpha'}_{g:f(e_2)}(e_1) \\
\equiv\ & \chi^{\alpha_1.\alpha}_{g:f(e_2)}(e_1) \\
\equiv\ & \rho_{\alpha_1:\alpha_1'}(\pi_{\overline{\alpha_1}}(\chi_{\alpha_1':\chi^{\alpha}_{g:f(e_2)}(\alpha_1)}(e_1))) \\
\equiv\ & \rho_{\alpha_1:\alpha_1'}(\pi_{\overline{\alpha_1}}(\{y \circ [\alpha_1' : \chi^{\alpha}_{g:f(e_2)}(y.\alpha_1)] | y \in e_1\})) \\
\overset{I.V.}{\equiv}\ & \rho_{\alpha_1:\alpha_1'}(\pi_{\overline{\alpha_1}}(\{y \circ [\alpha_1' : f^{\alpha.g}(\chi^{\alpha}_{g:e_2}(y.\alpha_1))] | y \in e_1\})) \\
\equiv\ & \{y_{\overline{\alpha_1}} \circ [\alpha_1 : f^{\alpha.g}(\chi^{\alpha}_{g:e_2}(y.\alpha_1))] | y \in e_1\}
\end{aligned}$$

$$f^{\alpha'.g}(\chi^{\alpha'}_{g:e_2}(e_1))$$

$$
\begin{aligned}
&\equiv && f^{\alpha_1.\alpha.g}(\chi^{\alpha_1.\alpha}_{g:e_2}(e_1))\\
&\equiv && f^{\alpha_1.\alpha.g}(\rho_{\alpha_1:\alpha_1'}(\pi_{\overline{\alpha_1}}(\chi_{\alpha_1':\chi^{\alpha}_{g:e_2}(\alpha_1)}(e_1))))\\
&\equiv && f^{\alpha_1.\alpha.g}(\rho_{\alpha_1:\alpha_1'}(\pi_{\overline{\alpha_1}}(\{y \circ [\alpha_1' : \chi^{\alpha}_{g:e_2}(y.\alpha_1)] | y \in e_1\})))\\
&\equiv && f^{\alpha_1.\alpha.g}(\rho_{\alpha_1:\alpha_1'}(\{y_{\overline{\alpha_1}} \circ [\alpha_1' : \chi^{\alpha}_{g:e_2}(y.\alpha_1)] | y \in e_1\}))\\
&\equiv && f^{\alpha_1.\alpha.g}(\{y_{\overline{\alpha_1}} \circ [\alpha_1 : \chi^{\alpha}_{g:e_2}(y.\alpha_1)] | y \in e_1\})\\
&\equiv && f^{\alpha_1.\alpha.g}(\{y_{\overline{\alpha_1}} \circ [\alpha_1 : \chi^{\alpha}_{g:e_2}(y.\alpha_1)] | y \in e_1\})\\
&\equiv && \rho_{\alpha_1:\alpha_1'}(\pi_{\overline{\alpha_1}}(\chi_{\alpha_1':f^{\alpha.g}(\alpha_1)}(\{y_{\overline{\alpha_1}} \circ [\alpha_1 : \chi^{\alpha}_{g:e_2}(y.\alpha_1)] | y \in e_1\})))\\
&\equiv && \rho_{\alpha_1:\alpha_1'}(\pi_{\overline{\alpha_1}}(\{y_{\overline{\alpha_1}} \circ [\alpha_1 : \chi^{\alpha}_{g:e_2}(y.\alpha_1)] \circ [\alpha_1' : f^{\alpha.g}(\chi^{\alpha}_{g:e_2}(y.\alpha_1))] | y \in e_1\}))\\
&\equiv && \rho_{\alpha_1:\alpha_1'}(\{y_{\overline{\alpha_1}} \circ [\alpha_1' : f^{\alpha.g}(\chi^{\alpha}_{g:e_2}(y.\alpha_1))] | y \in e_1\})\\
&\equiv && \{y_{\overline{\alpha_1}} \circ [\alpha_1 : f^{\alpha.g}(\chi^{\alpha}_{g:e_2}(y.\alpha_1))] | y \in e_1\}
\end{aligned}
$$

□

$$
\sigma^{\alpha.g}_p(e_1 \Gamma^{\alpha}_{g;A_1\theta A_2;f} e_2) \equiv e_1 \Gamma^{\alpha}_{g;A_1\theta A_2,p;f} e_2 \tag{63}
$$

Proof by induction over $|\alpha|$.

induction start: $|\alpha| = 0$ $\surd$

presupposition: Equivalence holds for arbitraty α with $|\alpha| = n$.

Induktionsschritt: $n+1 \leadsto n$

Let $\alpha' = \alpha_1.\alpha$, $|\alpha'| = n+1$

$$
\begin{aligned}
& && \sigma^{\alpha'.g}_p(e_1 \Gamma^{\alpha'}_{g;A_1\theta A_2;f} e_2)\\
&\equiv && \sigma^{\alpha_1.\alpha.g}_p(e_1 \Gamma^{\alpha_1.\alpha}_{g;A_1\theta A_2;f} e_2)\\
&\equiv && \sigma^{\alpha_1.\alpha.g}_p(\rho_{\alpha_1:\alpha_1'}(\pi_{\overline{\alpha_1}}(\chi_{\alpha_1':\alpha_1 \Gamma^{\alpha}_{g;A_1\theta A_2;f} e_2}(e_1))))\\
&\equiv && \sigma^{\alpha_1.\alpha.g}_p(\rho_{\alpha_1:\alpha_1'}(\pi_{\overline{\alpha_1}}(\{y \circ [\alpha_1' : y.\alpha_1 \Gamma^{\alpha}_{g;A_1\theta A_2;f} e_2] | y \in e_1\})))\\
&\equiv && \sigma^{\alpha_1.\alpha.g}_p(\rho_{\alpha_1:\alpha_1'}(\{y_{\overline{\alpha_1}} \circ [\alpha_1' : y.\alpha_1 \Gamma^{\alpha}_{g;A_1\theta A_2;f} e_2] | y \in e_1\}))\\
&\equiv && \sigma^{\alpha_1.\alpha.g}_p(\{y_{\overline{\alpha_1}} \circ [\alpha_1 : y.\alpha_1 \Gamma^{\alpha}_{g;A_1\theta A_2;f} e_2] | y \in e_1\})\\
&\equiv && \sigma^{\alpha_1.\alpha.g}_p(\{y_{\overline{\alpha_1}} \circ [\alpha_1 : y.\alpha_1 \Gamma^{\alpha}_{g;A_1\theta A_2;f} e_2] | y \in e_1\})\\
&\equiv && \rho_{\alpha_1:\alpha_1'}(\pi_{\overline{\alpha_1}}(\chi_{\alpha_1':\sigma^{\alpha.g}_p(\alpha_1)}(\{y_{\overline{\alpha_1}} \circ [\alpha_1 : y.\alpha_1 \Gamma^{\alpha}_{g;A_1\theta A_2;f} e_2] | y \in e_1\})))\\
&\equiv && \rho_{\alpha_1:\alpha_1'}(\pi_{\overline{\alpha_1}}(\{y_{\overline{\alpha_1}} \circ [\alpha_1 : y.\alpha_1 \Gamma^{\alpha}_{g;A_1\theta A_2;f} e_2] \circ [\alpha_1' : \sigma^{\alpha.g}_p(y.\alpha_1 \Gamma^{\alpha}_{g;A_1\theta A_2;f} e_2)] | y \in e_1\}))\\
&\equiv && \rho_{\alpha_1:\alpha_1'}(\{y_{\overline{\alpha_1}} \circ [\alpha_1' : \sigma^{\alpha.g}_p(y.\alpha_1 \Gamma^{\alpha}_{g;A_1\theta A_2;f} e_2)] | y \in e_1\})\\
&\equiv && \{y_{\overline{\alpha_1}} \circ [\alpha_1 : \sigma^{\alpha.g}_p(y.\alpha_1 \Gamma^{\alpha}_{g;A_1\theta A_2;f} e_2)] | y \in e_1\}\\
&\overset{I.V.}{\equiv} && \{y_{\overline{\alpha_1}} \circ [\alpha_1 : y.\alpha_1 \Gamma^{\alpha}_{g;A_1\theta A_2,p;f} e_2] | y \in e_1\}
\end{aligned}
$$

$$
\begin{array}{ll}
& e_1 \Gamma^{\alpha'}_{g;A_1\theta A_2,p;f} e_2 \\
\equiv & e_1 \Gamma^{\alpha_1.\alpha}_{g;A_1\theta A_2,p;f} e_2 \\
\equiv & \rho_{\alpha_1:\alpha_1'}(\pi_{\overline{\alpha_1}}(\chi_{\alpha_1':\alpha_1\Gamma^{\alpha}_{g;A_1\theta A_2,p;f}e_2}(e_1))) \\
\equiv & \rho_{\alpha_1:\alpha_1'}(\pi_{\overline{\alpha_1}}(\{y \circ [\alpha_1' : y.\alpha_1 \Gamma^{\alpha}_{g;A_1\theta A_2,p;f} e_2] | y \in e_1\})) \\
\equiv & \rho_{\alpha_1:\alpha_1'}(\{y_{\overline{\alpha_1}} \circ [\alpha_1' : y.\alpha_1 \Gamma^{\alpha}_{g;A_1\theta A_2,p;f} e_2] | y \in e_1\}) \\
\equiv & \{y_{\overline{\alpha_1}} \circ [\alpha_1 : y.\alpha_1 \Gamma^{\alpha}_{g;A_1\theta A_2,p;f} e_2] | y \in e_1\}
\end{array}
$$

$\square$

$$
\Gamma^{\alpha}_{g;\theta A;f_1 \circ f_2}(e) \equiv f_1^{\alpha.g}(\Gamma^{\alpha}_{g;\theta A;f_2}(e)) \tag{64}
$$

Proof by induction over $|\alpha|$.

induction start: $|\alpha| = 0 \quad \surd$

presupposition: Equivalence holds for arbitraty α with $|\alpha| = n$.

induction step: $n + 1 \rightsquigarrow n$

Let $\alpha' = \alpha_1.\alpha, \quad |\alpha'| = n + 1$

$$
\begin{array}{ll}
& f_1^{\alpha'.g}(\Gamma^{\alpha'}_{g;\theta A;f_2}(e)) \\
\equiv & f_1^{\alpha_1.\alpha.g}(\Gamma^{\alpha_1.\alpha}_{g;\theta A;f_2}(e)) \\
\equiv & f_1^{\alpha_1.\alpha.g}(\rho_{\alpha_1:\alpha_1'}(\pi_{\overline{\alpha_1}}(\chi_{\alpha_1':\Gamma^{\alpha}_{g;\theta A;f_2}(\alpha_1)}(e)))) \\
\equiv & f_1^{\alpha_1.\alpha.g}(\rho_{\alpha_1:\alpha_1'}(\pi_{\overline{\alpha_1}}(\{y \circ [\alpha_1' : \Gamma^{\alpha}_{g;\theta A;f_2}(y.\alpha_1)] | y \in e\}))) \\
\equiv & f_1^{\alpha_1.\alpha.g}(\rho_{\alpha_1:\alpha_1'}(\{y_{\overline{\alpha_1}} \circ [\alpha_1' : \Gamma^{\alpha}_{g;\theta A;f_2}(y.\alpha_1)] | y \in e\})) \\
\equiv & f_1^{\alpha_1.\alpha.g}(\{y_{\overline{\alpha_1}} \circ [\alpha_1 : \Gamma^{\alpha}_{g;\theta A;f_2}(y.\alpha_1)] | y \in e\}) \\
\equiv & \rho_{\alpha_1:\alpha_1'}(\pi_{\overline{\alpha_1}}(\chi_{\alpha_1':f_1^{\alpha.g}(\alpha_1)}(\{y_{\overline{\alpha_1}} \circ [\alpha_1 : \Gamma^{\alpha}_{g;\theta A;f_2}(y.\alpha_1)] | y \in e\}))) \\
\equiv & \rho_{\alpha_1:\alpha_1'}(\pi_{\overline{\alpha_1}}(\{y_{\overline{\alpha_1}} \circ [\alpha_1 : \Gamma^{\alpha}_{g;\theta A;f_2}(y.\alpha_1)] \circ [\alpha_1' : f_1^{\alpha.g}(\Gamma^{\alpha}_{g;\theta A;f_2}(y.\alpha_1))] | y \in e\})) \\
\equiv & \rho_{\alpha_1:\alpha_1'}(\{y_{\overline{\alpha_1}} \circ [\alpha_1' : f_1^{\alpha.g}(\Gamma^{\alpha}_{g;\theta A;f_2}(y.\alpha_1))] | y \in e\}) \\
\equiv & \{y_{\overline{\alpha_1}} \circ [\alpha_1 : f_1^{\alpha.g}(\Gamma^{\alpha}_{g;\theta A;f_2}(y.\alpha_1))] | y \in e\} \\
\overset{I.V.}{\equiv} & \{y_{\overline{\alpha_1}} \circ [\alpha_1 : \Gamma^{\alpha}_{g;\theta A;f_1 \circ f_2}(y.\alpha_1)] | y \in e\}
\end{array}
$$

$$
\begin{array}{ll}
& \Gamma^{\alpha'}_{g;\theta A;f_1 \circ f_2}(e) \\
\equiv & \Gamma^{\alpha_1.\alpha}_{g;\theta A;f_1 \circ f_2}(e)
\end{array}
$$

$$
\begin{aligned}
&\equiv \rho_{\alpha_1:\alpha_1'}(\pi_{\overline{\alpha_1}}(\chi_{\alpha_1':\Gamma^\alpha_{g;\theta A;f_1\circ f_2}(\alpha_1)}(e)))\\
&\equiv \rho_{\alpha_1:\alpha_1'}(\pi_{\overline{\alpha_1}}(\{y\circ[\alpha_1':\Gamma^\alpha_{g;\theta A;f_1\circ f_2}(y.\alpha_1)]|y\in e\}))\\
&\equiv \rho_{\alpha_1:\alpha_1'}(\{y_{\overline{\alpha_1}}\circ[\alpha_1':\Gamma^\alpha_{g;\theta A;f_1\circ f_2}(y.\alpha_1)]|y\in e\})\\
&\equiv \{y_{\overline{\alpha_1}}\circ[\alpha_1:\Gamma^\alpha_{g;\theta A;f_1\circ f_2}(y.\alpha_1)]|y\in e\}
\end{aligned}
$$

□

$$
e_1\Gamma^\alpha_{g;\theta A;f_1\circ f_2}e_2 \equiv f_1^{\alpha.g}(e_1\Gamma^\alpha_{g;\theta A;f_2}e_2) \tag{65}
$$

Proof by induction over $|\alpha|$.

induction start: $|\alpha| = 0 \quad \surd$

presupposition: Equivalence holds for arbitraty α with $|\alpha| = n$.

induction step: $n+1 \rightsquigarrow n$

Let $\alpha' = \alpha_1.\alpha, \quad |\alpha'| = n+1$

$$
\begin{aligned}
& f_1^{\alpha'.g}(e_1\Gamma^{\alpha'}_{g;\theta A;f_2}e_2)\\
\equiv\ & f_1^{\alpha_1.\alpha.g}(e_1\Gamma^{\alpha_1.\alpha}_{g;\theta A;f_2}e_2)\\
\equiv\ & f_1^{\alpha_1.\alpha.g}(\rho_{\alpha_1:\alpha_1'}(\pi_{\overline{\alpha_1}}(\chi_{\alpha_1':\alpha_1\Gamma^\alpha_{g;\theta A;f_2}e_2}(e_1))))\\
\equiv\ & f_1^{\alpha_1.\alpha.g}(\rho_{\alpha_1:\alpha_1'}(\pi_{\overline{\alpha_1}}(\{y\circ[\alpha_1':y.\alpha_1\Gamma^\alpha_{g;\theta A;f_2}e_2]|y\in e_1\})))\\
\equiv\ & f_1^{\alpha_1.\alpha.g}(\rho_{\alpha_1:\alpha_1'}(\{y_{\overline{\alpha_1}}\circ[\alpha_1':y.\alpha_1\Gamma^\alpha_{g;\theta A;f_2}e_2]|y\in e_1\}))\\
\equiv\ & f_1^{\alpha_1.\alpha.g}(\{y_{\overline{\alpha_1}}\circ[\alpha_1:y.\alpha_1\Gamma^\alpha_{g;\theta A;f_2}e_2]|y\in e_1\})\\
\equiv\ & \rho_{\alpha_1:\alpha_1'}(\pi_{\overline{\alpha_1}}(\chi_{\alpha_1':f_1^{\alpha.g}(\alpha_1)}(\{y_{\overline{\alpha_1}}\circ[\alpha_1:y.\alpha_1\Gamma^\alpha_{g;\theta A;f_2}e_2]|y\in e_1\})))\\
\equiv\ & \rho_{\alpha_1:\alpha_1'}(\pi_{\overline{\alpha_1}}(\{y_{\overline{\alpha_1}}\circ[\alpha_1:y.\alpha_1\Gamma^\alpha_{g;\theta A;f_2}e_2]\circ[\alpha_1':f_1^{\alpha.g}(y.\alpha_1\Gamma^\alpha_{g;\theta A;f_2}e_2)]|y\in e_1\}))\\
\equiv\ & \rho_{\alpha_1:\alpha_1'}(\{y_{\overline{\alpha_1}}\circ[\alpha_1':f_1^{\alpha.g}(y.\alpha_1\Gamma^\alpha_{g;\theta A;f_2}e_2)]|y\in e_1\})\\
\equiv\ & \{y_{\overline{\alpha_1}}\circ[\alpha_1:f_1^{\alpha.g}(y.\alpha_1\Gamma^\alpha_{g;\theta A;f_2}e_2)]|y\in e_1\}\\
\overset{I.V.}{\equiv}\ & \{y_{\overline{\alpha_1}}\circ[\alpha_1:y.\alpha_1\Gamma^\alpha_{g;\theta A;f_1\circ f_2}e_2]|y\in e_1\}
\end{aligned}
$$

$$
\begin{aligned}
& e_1\Gamma^{\alpha'}_{g;\theta A;f_1\circ f_2}e_2\\
\equiv\ & e_1\Gamma^{\alpha_1.\alpha}_{g;\theta A;f_1\circ f_2}e_2\\
\equiv\ & \rho_{\alpha_1:\alpha_1'}(\pi_{\overline{\alpha_1}}(\chi_{\alpha_1':\alpha_1\Gamma^\alpha_{g;\theta A;f_1\circ f_2}e_2}(e_1)))\\
\equiv\ & \rho_{\alpha_1:\alpha_1'}(\pi_{\overline{\alpha_1}}(\{y\circ[\alpha_1':y.\alpha_1\Gamma^\alpha_{g;\theta A;f_1\circ f_2}e_2]|y\in e_1\}))\\
\equiv\ & \rho_{\alpha_1:\alpha_1'}(\{y_{\overline{\alpha_1}}\circ[\alpha_1':y.\alpha_1\Gamma^\alpha_{g;\theta A;f_1\circ f_2}e_2]|y\in e_1\})\\
\equiv\ & \{y_{\overline{\alpha_1}}\circ[\alpha_1:y.\alpha_1\Gamma^\alpha_{g;\theta A;f_1\circ f_2}e_2]|y\in e_1\}
\end{aligned}
$$

$\square$

$$e_1\Gamma^{\alpha_1}_{g_1;p_1}(e_2\Gamma^{\alpha_2}_{g_2;p_2}e_3) \equiv (e_1\Gamma^{\alpha_1}_{g_1;p_1}e_2)\Gamma^{\alpha_1.g_1.\alpha_2}_{g_2;p_2}e_3 \tag{66}$$

Proof by induction over $|\alpha_1|$:

induction start: $|\alpha_1| = 0 \quad \surd$

by presupposition : Equivalence holds for arbitraty α_1 with $|\alpha_1| = n$.

induction step: $n+1 \rightsquigarrow n$

Let $\alpha'_1 = \alpha_{11}.\alpha_1, \quad |\alpha'_1| = n+1$

$$\begin{array}{ll}
 & (e_1\Gamma^{\alpha'_1}_{g_1;p_1}e_2)\Gamma^{\alpha'_1.g_1.\alpha_2}_{g_2;p_2}e_3 \\
\equiv & (e_1\Gamma^{\alpha_{11}.\alpha_1}_{g_1;p_1}e_2)\Gamma^{\alpha_{11}.\alpha_1.g_1.\alpha_2}_{g_2;p_2}e_3 \\
\equiv & \rho_{\alpha_{11}:\alpha'_{11}}(\pi_{\overline{\alpha_{11}}}(\chi_{\alpha'_{11}:\alpha_{11}\Gamma^{\alpha_1}_{g_1;p_1}e_2}(e_1)))\Gamma^{\alpha_{11}.\alpha_1.g_1.\alpha_2}_{g_2;p_2}e_3 \\
\equiv & \rho_{\alpha_{11}:\alpha'_{11}}(\pi_{\overline{\alpha_{11}}}(\{y \circ [\alpha'_{11} : y.\alpha_{11}\Gamma^{\alpha_1}_{g_1;p_1}e_2] | y \in e_1\}))\Gamma^{\alpha_{11}.\alpha_1.g_1.\alpha_2}_{g_2;p_2}e_3 \\
\equiv & \rho_{\alpha_{11}:\alpha'_{11}}(\{y_{\overline{\alpha_{11}}} \circ [\alpha'_{11} : y.\alpha_{11}\Gamma^{\alpha_1}_{g_1;p_1}e_2] | y \in e_1\})\Gamma^{\alpha_{11}.\alpha_1.g_1.\alpha_2}_{g_2;p_2}e_3 \\
\equiv & \{y_{\overline{\alpha_{11}}} \circ [\alpha_{11} : y.\alpha_{11}\Gamma^{\alpha_1}_{g_1;p_1}e_2] | y \in e_1\}\Gamma^{\alpha_{11}.\alpha_1.g_1.\alpha_2}_{g_2;p_2}e_3 \\
\equiv & \rho_{\alpha_{11}:\alpha'_{11}}(\pi_{\overline{\alpha_{11}}}(\chi_{\alpha'_{11}:\alpha_{11}\Gamma^{\alpha_1.g_1.\alpha_2}_{g_2;p_2}e_3}(\{y_{\overline{\alpha_{11}}} \circ [\alpha_{11} : y.\alpha_{11}\Gamma^{\alpha_1}_{g_1;p_1}e_2] | y \in e_1\}))) \\
\equiv & \rho_{\alpha_{11}:\alpha'_{11}}(\pi_{\overline{\alpha_{11}}}(\{y_{\overline{\alpha_{11}}} \circ [\alpha_{11} : y.\alpha_{11}\Gamma^{\alpha_1}_{g_1;p_1}e_2] \circ [\alpha'_{11} : y.\alpha_{11}\Gamma^{\alpha_1}_{g_1;p_1}e_2\Gamma^{\alpha_1.g_1.\alpha_2}_{g_2;p_2}e_3] | y \in e_1\})) \\
\equiv & \rho_{\alpha_{11}:\alpha'_{11}}(\{y_{\overline{\alpha_{11}}} \circ [\alpha'_{11} : y.\alpha_{11}\Gamma^{\alpha_1}_{g_1;p_1}e_2\Gamma^{\alpha_1.g_1.\alpha_2}_{g_2;p_2}e_3] | y \in e_1\}) \\
\equiv & \{y_{\overline{\alpha_{11}}} \circ [\alpha_{11} : (y.\alpha_{11}\Gamma^{\alpha_1}_{g_1;p_1}e_2)\Gamma^{\alpha_1.g_1.\alpha_2}_{g_2;p_2}e_3] | y \in e_1\} \\
\overset{I.V.}{\equiv} & \{y_{\overline{\alpha_{11}}} \circ [\alpha_{11} : y.\alpha_{11}\Gamma^{\alpha_1}_{g_1;p_1}(e_2\Gamma^{\alpha_2}_{g_2;p_2}e_3)] | y \in e_1\}
\end{array}$$

$$\begin{array}{ll}
 & e_1\Gamma^{\alpha'_1}_{g_1;p_1}(e_2\Gamma^{\alpha_2}_{g_2;p_2}e_3) \\
\equiv & e_1\Gamma^{\alpha_{11}.\alpha_1}_{g_1;p_1}(e_2\Gamma^{\alpha_2}_{g_2;p_2}e_3) \\
\equiv & \rho_{\alpha_{11}:\alpha'_{11}}(\pi_{\overline{\alpha_{11}}}(\chi_{\alpha'_{11}:\alpha_{11}\Gamma^{\alpha_1}_{g_1;p_1}(e_2\Gamma^{\alpha_2}_{g_2;p_2}e_3)}(e_1))) \\
\equiv & \rho_{\alpha_{11}:\alpha'_{11}}(\pi_{\overline{\alpha_{11}}}(\{y \circ [\alpha'_{11} : y.\alpha_{11}\Gamma^{\alpha_1}_{g_1;p_1}(e_2\Gamma^{\alpha_2}_{g_2;p_2}e_3)] | y \in e_1\})) \\
\equiv & \rho_{\alpha_{11}:\alpha'_{11}}(\{y_{\overline{\alpha_{11}}} \circ [\alpha'_{11} : y.\alpha_{11}\Gamma^{\alpha_1}_{g_1;p_1}(e_2\Gamma^{\alpha_2}_{g_2;p_2}e_3)] | y \in e_1\}) \\
\equiv & \{y_{\overline{\alpha_{11}}} \circ [\alpha_{11} : y.\alpha_{11}\Gamma^{\alpha_1}_{g_1;p_1}(e_2\Gamma^{\alpha_2}_{g_2;p_2}e_3)] | y \in e_1\}
\end{array}$$

$\square$