Query Deforestation

Torsten Grust   **Marc H. Scholl**

68/1998

Faculty of Mathematics and Computer Science
University of Konstanz

`{Torsten.Grust, Marc.Scholl}@uni-konstanz.de`

Abstract

The construction of a declarative query engine for a DBMS includes the challenge of compiling algebraic queries into efficient execution plans that can be run on top of the persistent storage. This work pursues the goal of employing `foldr-build` deforestation for the derivation of efficient streaming programs – programs that do not allocate intermediate data structures to perform their task – from algebraic (combinator) query plans. The query engine is based on the insertion representation of collections and base types, making the deforestation idea amenable to a wide range of type constructors and operators. In addition to the removal of intermediate data structures, deforestation actually implements certain algebraic optimizations for free. Query deforestation is a generally applicable technique that puts the derivation of streaming programs in a compact formal framework that should be immediately applicable.

1 Motivation

The construction of a declarative query interface for a DBMS lets several challenges show up, among which the compilation of queries into equivalent access plans and their efficient execution on top of the persistent storage play a major role.

Compiled plans exhibit a tree-like structure in which the nodes represent operators that implement diverse tasks like selections, aggregations, joins, and grouping. It is non-trivial to efficiently schedule the flow of data from the leaves of the processing tree to its root which represents the query result. Assigning operators to single processes connected by IPC facilities (to control the flow of execution) as well as disk files (to communicate temporary results) and then let the operating system do the scheduling has been identified as too costly an option [11, 12]. The necessary process context switches, IPC cost, and I/O of temporary results dominate the overall query cost.

Different lines of research [7, 8, 16] led to the development of single-process query engines in which the query plan is compiled into *a single iterative or recursive procedure*. Whenever possible these approaches strive for a *stream-based* (or pipelined) execution to avoid I/O of temporary data. The query processor benefits from streaming since objects are addressed and brought in from persistent storage only once. Execution is in-memory and no intermediate writes to the persistent store and subsequent fault-ins occur.

However, given that the query compiler arranges the query plan as a sequence of separate operators, it is not obvious how an efficient streaming procedure can be automatically derived: operators are commonly individually designed as combinators that consume their input as a whole, transform it, and subsequently produce an intermediate result. Combinators are then arranged by functional composition to form query plans. These combinator-style query engines are highly modular, easily extensible, and are suitable targets for query compilation, but one bears the above-mentioned performance penalty. It is the core concern of the present work to *automatically* derive streaming programs for an extensive query engine (w.r.t. supported operators as well as data types) from arbitrary combinator queries. This will go well beyond common pipelining approaches which are geared to fuse specific combinator patterns only (e.g. adjacent *select-project-join* operators).

Graefe [11] proposed a solution in which operators had to be re-coded in a streaming-style: operators consume their input on demand (lazily) and element-wise. These operators then interact via a simple *end-of-stream?* and *next* call interface. Almost simultaenously [7, 8] employed an instance of Burlington and Durstall's [4] *unfold-transform-fold* program transformation strategy: the inner control structure of the combinators – expressed in a limited subset of LISP – were revealed (*unfolded*) with the goal of *fusing* neighbouring combinators. Rule sets described if and how fusing was possible. The strategy then tried to *fold* the fused program back into combinator form, a step that involved complex pattern matching and, for some cases, could only be done semi-automatically. [16] developed a very specific set of fusion rules for an imperative programming language in which plans had to be coded.

To arrange a query plan as a sequence of combinators actually coincides very closely with the so-called *listful style* of functional programming: complex list manipulations are expressed as compositions of simple but general functions (e.g. `map`), each generating an intermediate result list.[1] Given that the combinators adhere a simple syntactic criterion, such programs can be optimized by Wadler's *deforestation* [22]. A deforested program never allocates intermediate data structures.[2] Unfortunately, deforestation also involves a fold step, which increases the technique's complexity considerably.

The rather regular collection producer/consumer scenario of a query engine, however, does not demand a deforestation technique applicable to a full functional programming language: *cheap deforestation* [9, 15] becomes possible for programs that use the generic combinator `foldr` as a consumer. In the query compilation context this is actually not as restrictive as it may seem. In [13] we have shown that the combinators needed for the compilation of a full-fledged declarative object query language like OQL can entirely be expressed by `foldr`. Other proposals for expressive query languages also relied on variants of `foldr` [1, 21, 5].

Cheap (or `foldr-build`) deforestation is a one-step transformation that does not involve a fold step. It relies on the observation that a list built from the constructors `:` (*cons*) and `[]` (*nil*) which is subsequently reduced by a `foldr` may actually be reduced *during* its construction. Furthermore, due to deforestation, formerly separate program parts become adjacent. This may – and in effect does, as we will see – offer the opportunity for further optimizations which, e.g., the stream operator model of [11] does not provide.

Contribution. Deforestation provides a formal framework for the derivation of stream-based plans, a problem that has primarily been tackled on the implementation level only. We extend the formerly only list-based deforestation to work with left-associative algebras in general which makes the idea amenable to query engines for recent object query languages like OQL, their advanced bulk type systems, and operator sets. Using the very same formal setting, the present work complements the algebraic optimization described in [13] with a corresponding optimizing translation on the execution plan level. Query deforestation handles classes of (nested) programs which could not be expressed by previous work, e.g. the *unsafe programs* of [5]. Due to its simplicity (deforestation is based on a one-step transformation, rather than a large set of equivalence laws) the technique is easily integrated into an actual query optimizer.

Synopsis. In the sequel we will pursue the goal of a `foldr-build` deforestation of query execution plans for a source language like OQL.

We will proceed as follows. Section 2 reviews the insertion representation for collections and base data types, making queries over sets, bags, as well as quantification and aggregation accessible to the originally list-based cheap deforestation. Query engine combinators will be mapped to a core language built around `foldr`. A translation of OQL into this framework is possible but this is not principal to the method and may be found in [13] and [6]. Section 3 examines `foldr-build` deforestation of queries and will show it to cover and extend, despite its generality, special

[1] A lazy functional language may never built these lists as a whole but their elements have to be allocated and garbage-collected anyway.

[2] Deforestation removes intermediate data structures, which Wadler collectively refers to as trees. Hence the name deforestation.

transformations like those developed in [7], [5], and [17]. We will discuss connections to these related efforts when the particular details are in context. Section 4 sheds light on correctness and implementation issues. How this idea might evolve is briefly touched in Section 5. A Haskell-style [14] representation language will be used for the sake of readability.

2 Query Representation

Any finite value of type α *list* may be built by finitely many applications of *list*'s two constructors `[]` and `(:)`. This is a consequence of the initiality of the algebra (α *list*, `[]`, `:`) and justifies the definition of functions that use pattern matching on the patterns `[]` resp. $x : xs$ to take lists apart again. The principle extends to other algebras. Finite values of type α *set* may be built by finitely many insertions into the empty set, similarly for the *bag* type constructor. We will adopt this *insertion representation* for collections [20, 3] and – for the sake of notational uniformity – denote the constructors by ($\overset{\tau}{:}$, $[]^\tau$) for type constructor τ. $\overset{bag}{:}$ as well as $\overset{set}{:}$ are left-commutative, the latter also left-idempotent, i.e. $x \overset{set}{:} y \overset{set}{:} xs = y \overset{set}{:} x \overset{set}{:} xs$ and $x \overset{set}{:} x \overset{set}{:} xs = x \overset{set}{:} xs$ ($\overset{\tau}{:}$ associates to the right[3]). This will have consequences for the well-definedness of `foldr` to which we return in Section 4.

Aggregation and quantification may be understood in this model as well. The aggregates `sum`, `min` are represented as the algebras $sum = (int, \mathtt{0}, \mathtt{+})$ resp. $min = (int, \infty_{int}, \mathtt{min})$. Quantifiers `all` and `exists` are implemented as $all = (bool, \mathtt{True}, \mathtt{\&\&})$ and $exists = (bool, \mathtt{False}, \mathtt{||})$: universal quantification over a collection of booleans starts with constructor `True` and inductively adds a collection member using boolean conjunction `(&&)` to compute the result. Below, we will prefer to write the usual constructors for the non-collection algebras, e.g. we will use (`True`, `&&`) instead of ($[]^{all}$, $\overset{all}{:}$) although both denote the same. See [13] for a mapping of OQL's complete family of aggregates and quantifiers along the same lines.

The query engine's algebraic combinators are now as naturally expressed as familiar list operations. At this point, Figure 1 shows selection (`filter`) and existential quantification (`exists`) only as there is nothing really new to be learned. Instead, to make them amenable to deforestation, we will give `foldr`-based definitions of the engine's combinators shortly. Note that `filter` and `exists` exclusively use the constructors of τ resp. of the *exists* algebra to produce their result. The combinators carry the collection type τ they operate on as an annotation. For the time being the annotations may be understood as type abstractions in the sense of second-order λ-calculus.

$$
\begin{array}{lcl}
\mathtt{filter}^\tau & :: & (\alpha \to bool) \to \alpha\ \tau \to \alpha\ \tau \\
\mathtt{filter}^\tau\ \mathtt{p}\ []^\tau & = & []^\tau \\
\mathtt{filter}^\tau\ \mathtt{p}\ (x \overset{\tau}{:} xs) & = & \mathtt{if\ p}\ x\ \mathtt{then}\ x \overset{\tau}{:} (\mathtt{filter\ p}\ xs)\ \mathtt{else\ filter\ p}\ xs \\
\mathtt{exists}^\tau & :: & (\alpha \to bool) \to \alpha\ \tau \to bool \\
\mathtt{exists}^\tau\ \mathtt{p}\ []^\tau & = & \mathtt{False} \\
\mathtt{exists}^\tau\ \mathtt{p}\ (x \overset{\tau}{:} xs) & = & \mathtt{p}\ x\ \mathtt{||}\ \mathtt{exists}^\tau\ \mathtt{p}\ xs
\end{array}
$$

Figure 1: Recursive definitions of combinators `filter` and `exists`.

`filter` and `exists` exhibit a pattern of simple recursion and reduction that can be found in all algebraic combinators. The generic functional `foldr` (variants of which are referred to as *pump* [1], *flattenmap* [17], or *sri* [20] in the literature) implements this pattern, which implies that a complete query engine could be built solely based on an efficient implementation of `foldr` (Section 4 will come back to this point):

$$
\begin{array}{lcl}
\mathtt{foldr}^\tau & :: & (\alpha \to \beta \to \beta) \to \beta \to \alpha\ \tau \to \beta \\
\mathtt{foldr}^\tau\ \mathtt{f\ z}\ []^\tau & = & \mathtt{z} \\
\mathtt{foldr}^\tau\ \mathtt{f\ z}\ (x \overset{\tau}{:} xs) & = & \mathtt{f}\ x\ (\mathtt{foldr}^\tau\ \mathtt{f\ z}\ xs)
\end{array}
$$

[3] We sometimes abbreviate $x_1 \overset{\tau}{:} x_2 \overset{\tau}{:} \cdots \overset{\tau}{:} x_n \overset{\tau}{:} []^\tau$ by $[x_1, x_2, \ldots, x_n]^\tau$ in what follows. As in Haskell, we will write $(\oplus)$ for the prefix application of infix operator $\oplus$, and `` `f` `` if a binary function `f` is used as an infix operator.

Evaluation of $\texttt{foldr}^{\tau}\ \texttt{f}\ \texttt{z}\ (x_1 \overset{\tau}{:} x_2 \overset{\tau}{:} \cdots \overset{\tau}{:} x_n \overset{\tau}{:} \texttt{[]}^{\tau})$ results in x_1 ‘f‘ (x_2 ‘f‘ (... ‘f‘ (x_n ‘f‘ z) $\cdots$), i.e. the collection argument's constructor ($\overset{\tau}{:}$) is replaced by the binary function f, and $\texttt{[]}^{\tau}$ by z likewise. With $\texttt{f} = \lambda x\ xs$ `->` p x || xs and z = False we obtain exists p.

The transformation of the recursive combinator definitions into an equivalent non-recursive foldr-based form is rather straightforward and may even be done by a machine [15]. Figure 2 lists the query operators that we will encounter in the sequel. Many more may be similarly defined. We will have to say something about the well-definedness of the operators in Section 4.

$$
\begin{array}{ll}
\texttt{filter}^{\tau}\ \texttt{p}\ s & = \texttt{foldr}^{\tau}\,(\lambda x\ xs\ \texttt{->}\ \texttt{if p}\ x\ \texttt{then}\ x \overset{\tau}{:} xs\ \texttt{else}\ xs)\ \texttt{[]}^{\tau}\ s \\
\texttt{map}^{\tau}\ \texttt{f}\ s & = \texttt{foldr}^{\tau}\,(\lambda x\ xs\ \texttt{->}\ \texttt{f}\ x \overset{\tau}{:} xs)\ \texttt{[]}^{\tau}\ s \\
\texttt{exists}^{\tau}\ \texttt{p}\ s & = \texttt{foldr}^{\tau}\,(\lambda x\ xs\ \texttt{->}\ \texttt{p}\ x \,||\, xs)\ \texttt{False}\ s \\
\texttt{all}^{\tau}\ \texttt{p}\ s & = \texttt{foldr}^{\tau}\,(\lambda x\ xs\ \texttt{->}\ \texttt{p}\ x\,\&\&\,xs)\ \texttt{True}\ s \\
\texttt{sum}^{\tau}\ s & = \texttt{foldr}^{\tau}\,(\lambda x\ xs\ \texttt{->}\ x + xs)\ \texttt{0}\ s \\
\texttt{count}^{\tau}\ s & = \texttt{foldr}^{\tau}\,(\lambda x\ xs\ \texttt{->}\ \texttt{1} + xs)\ \texttt{0}\ s \\
\texttt{join}^{\tau\sigma}\ \texttt{p f}\ s_1\ s_2 & = \texttt{foldr}^{\tau}\,(\lambda x\ xs\ \texttt{->}\ \texttt{foldr}^{\sigma}\,(\lambda y\ ys\ \texttt{->} \\
 & \qquad\qquad \texttt{if p}\ x\ y\ \texttt{then f}\ x\ y \overset{\tau}{:} ys\ \texttt{else}\ ys)\ xs\ s_2)\ \texttt{[]}^{\tau}\ s_1 \\
\texttt{semijoin}^{\tau\sigma}\ \texttt{p}\ s_1\ s_2 & = \texttt{foldr}^{\tau}\,(\lambda x\ xs\ \texttt{->} \\
 & \qquad\qquad \texttt{if exists}^{\sigma}\ (\texttt{p}\ x)\ s_2\ \texttt{then}\ x \overset{\tau}{:} xs\ \texttt{else}\ xs)\ \texttt{[]}^{\tau}\ s_1 \\
\texttt{nestjoin}^{\tau\sigma}\ \texttt{p f}\ s_1\ s_2 & = \texttt{foldr}^{\tau}\,(\lambda x\ xs\ \texttt{->}\ (\texttt{foldr}^{\sigma}\,(\lambda y\ ys\ \texttt{->} \\
 & \qquad\qquad \texttt{if p}\ x\ y\ \texttt{then f}\ x\ y \overset{\sigma}{:} ys\ \texttt{else}\ ys)\ \texttt{[]}^{\sigma}\ s_2) \overset{\tau}{:} xs)\ \texttt{[]}^{\tau}\ s_1
\end{array}
$$

Figure 2: foldr-based definitions of algebraic combinators.

Aside from nestjoin the combinators are pretty standard and should explain themselves, e.g. join p f implements a binary join with respect to predicate p and result-building function f, i.e. we obtain a collection of pairs with $\texttt{f} = \lambda x\ y$ `->` (x,y). Operator nestjoin combines join and grouping: for each object in s_1, a group of objects from s_2 w.r.t. predicate p is built (no duplicate removal). nestjoin has already been proven to be especially useful in query processors for complex object models [18, 19]. Despite the rather complex control structure of nestjoin, deforestation will enable us to fuse it with adjacent combinators automatically.

In [13] we have shown that a set of combinators built along these lines can be used to implement a query language like OQL efficiently. This former work actually approached the problem from the opposite direction: a mapping of OQL to *monad comprehensions* was developed, which, in turn, could easily be mapped to algebraic combinators including foldr. Comprehensions deforest well as [9] already pointed out. We will briefly sketch a suitable generalization of this observation in Section 4 as it may be used to rapidly prototype a complete query language compiler.

Figure 3 summarizes the core representation language for the query engine to finalize this section. The semantics is that of a non-strict functional language extended by additional collection type constructors. Records are represented as tuples on this level.

3 Cheap Deforestation

The widely accepted approach of defining a query algebra as a set of combinators that may be orthogonally combined and rearranged facilitates algebraic optimization but does not fit well with the idea of deducing streaming programs. Combinator algebras implicitly accept the need for temporary storage to communicate results between operators, while stream-based execution plans avoid the allocation of intermediate collections whenever possible.

For some combinator compositions it is quite easy to see how to get rid of intermediate results, though. Consider the simple OQL query

$$
\begin{array}{rl}
\texttt{select} & x\texttt{.f} \\
\texttt{from} & x\ \texttt{in}\ es \\
\texttt{where} & \texttt{p}(x)
\end{array}
\qquad (Q_1)
$$

e	$\rightarrow$	v	variables
	$\mid$	`c`	constants
	$\mid$	$(e_1, \ldots, e_n)$	tuples ($n \geq 1$)
	$\mid$	$e_1\ e_2$	application
	$\mid$	λvs `->` e	abstraction
	$\mid$	$\mathtt{foldr}^{\tau}$ `f z`	reduction
	$\mid$	$\mathtt{[]}^{\tau} \mid \overset{\tau}{:}$	constructors
	$\mid$	`if` e_1 `then` e_2 `else` e_3	conditional
	$\mid$	`(`op`)` $e_1 \ldots e_n$	primitives ($n \geq 0$, $op =$ `+`, `*`, `<`, `>`, $\ldots$)
	$\mid$	`let` *decls* `in` e	local definitions
decl	$\rightarrow$	v `=` e	variable binding

Figure 3: Query engine core representation language.

and its algebraic equivalent `map f (filter p` es`)`. Instead of allocating the temporary result of the `filter p` pass over es, a streaming plan would scan es, and *immediately* apply `f` but only to those objects x that satisfy `p` (and otherwise just drop x). The corresponding query engine program does not allocate intermediate results and would read

$$\mathtt{foldr}^{\tau}\ (\lambda x\ xs\ \texttt{->}\ \mathtt{if\ p}\ x\ \mathtt{then\ f}\ x \overset{\tau}{:} xs\ \mathtt{else}\ xs)\ \mathtt{[]}^{\tau}\ es$$

Having detected this particular chance for streaming, we could supply the plan generator with a rewrite rule to realize this specific transformation.

However, how are we supposed to deduce a streaming program from a larger composition of arbitrary algebraic operators? The set of rewrite rules would have to account for all possible pairs of combinators (thus suffering from combinatorial explosion) and would be subject to major change on every modification of the query algebra.

This is where the regular collection consumption pattern expressed by `foldr` comes into play. Recall that $\mathtt{foldr}^{\tau}$ `f z` traverses its argument of type $\alpha\ \tau$ and replaces collection constructor $(\overset{\tau}{:})$ by `f` resp. $\mathtt{[]}^{\tau}$ by `z` as it goes. If this replacement could be done at *plan generation time*, i.e. not during query execution but statically, we would get rid of that `foldr` at all! Cheap deforestation [9] does exactly this for $\tau = \mathit{list}$, but the principle nicely adapts to the insertion representation for collection and base types, as we will now see.

To implement this idea we first need a means to gain a handle of all constructor occurences in a core language expression, g say. We can achieve this by compile-time β-abstraction of g w.r.t. the constructors used[4]: $\lambda \mathtt{c\ n}$ `->` $g[\text{`c`} / \overset{\tau}{:}][\mathtt{n}/\ \mathtt{[]}^{\tau}]$. The result of applying $\mathtt{foldr}^{\tau}$ `f z` to g may now readily be computed at plan generation time:

$$\mathtt{foldr}^{\tau}\ \mathtt{f\ z}\ g \underset{(\star)}{=} (\lambda \mathtt{c\ n}\ \texttt{->}\ g[\text{`c`} / \overset{\tau}{:}][\mathtt{n}/\ \mathtt{[]}^{\tau}])\ \mathtt{f\ z} \underset{\beta}{=} g[\text{`f`} / \overset{\tau}{:}][\mathtt{z}/\ \mathtt{[]}^{\tau}]$$

Note that rightmost expression is a streaming program for the composition $\mathtt{foldr}^{\tau}$ `f z` g: instead of constructing a temporary collection using $(\overset{\tau}{:})$ and $\mathtt{[]}^{\tau}$, g uses `f` and `z` instead to *reduce its input during consumption.*

The correctness of the deforestation step at $(\star)$ clearly depends on the prerequisite that g exclusively uses $(\overset{\tau}{:})$ and $\mathtt{[]}^{\tau}$ to produce its result since we need to replace *all* constructors in order to implement streaming correctly. Fortunately, this condition is met by all algebraic combinators in Figure 2. [9], which gives a proof for $(\star)$, uses a (non-Hindley-Milner) type constraint on g to elegantly enforce correctness.

With the definition of $\mathtt{build}^{\tau}\ g = g\ (\overset{\tau}{:})\ \ \mathtt{[]}^{\tau}$, a function g that has been abstracted over the constructors (i.e. $g = g'\ (\overset{\tau}{:})\ \ \mathtt{[]}^{\tau}$ where g' has been derived by β-abstraction of g as described

[4] $e[x/y]$ denotes e with all free occurences of y replaced by x.

```
map^τ f s           = build^τ (λc n -> foldr^τ (λx xs -> f x `c` xs) n s)
filter^τ p s        = build^τ (λc n -> foldr^τ (λy ys ->
                         if p y then y `c` ys else ys) n s)
all^τ p s           = build^all (λc n -> foldr^τ (λx xs -> p x `c` xs) n s)
sum^τ s             = build^sum (λc n -> foldr^τ (λx xs -> x `c` xs) n s)
nestjoin^τσ p f s1 s2 = build^τ (λc1 n1 -> foldr^τ (λx xs ->
                         build^σ (λc2 n2 -> (foldr^σ (λy ys ->
                          if p x y then f x y `c2` ys else ys) n2 s2) `c1` xs) n1 s1))
```

Figure 4: Combinators abstracted over their constructors.

above) may be written as $\texttt{build}^\tau\ g'$ from now on. This allows us to render ($\star$) more concisely as

$$\texttt{foldr}^\tau\ \texttt{f z}\ (\texttt{build}^\tau\ g') \;=\; g'\ \texttt{f z}$$

This `foldr`-`build` cancellation is the only transformation query deforestation relies on. No fold step as in [8] is required.

The application to the streaming plan problem is immediate: the `foldr`-based combinators are abstracted over all algebraic constructors they use to build their result and then reexpressed via `build`. Figure 4 summarizes the `build`-forms of some operators which are derived alike for all combinators (some are omitted here for the sake of brevity). Note that `nestjoin` produces its result using ($\overset{\tau}{:}$, $[]^\tau$) and ($\overset{\sigma}{:}$, $[]^\sigma$), so `build` has to be used twice to fully abstract the constructors away.

The implementation of query deforestation is inherent to the method since it is based on a single syntactic transformation to be applied exhaustively. This is in contrast to, e.g., [17] where extensive sets of rewriting rules are derived from a generic `fold` fusion rule. Such rule sets imply the need for a more or less sophisticated rule application strategy. Additionally, [17] accompanies specific rules with rather complex provisos (e.g. strictness or distributivity of functions) that cannot easily be asserted, let alone be checked syntactically [2].

A number of examples might help to reveal the potentials of query deforestation.

3.1 Deforestation Examples

To come back to our introductory query Q_1, deforestation computes the streaming plan for $\texttt{map}^\tau$ `f` ($\texttt{filter}^\tau$ `p` es) as follows. The definitions of `map` and `filter` are unfolded to give

```
build^τ (λc n ->
    foldr^τ (λx xs -> f x `c` xs) n
        (build^τ (λc1 n1 ->
            foldr^τ (λy ys -> if p y then y `c1` ys else ys) n1 es)))
```

The `foldr`-`build` rule is then applied once (the affected `foldr`-`build` pair is marked grey):

```
build^τ (λc n ->
    foldr^τ (λy ys -> if p y then f y `c` ys else ys) n es)
```

Finally, unfolding the outer `build` gives the desired streaming program we have shown before:

```
foldr^τ (λy ys -> if p y then f y :^τ ys else ys) []^τ es
```

Now consider `semijoin p (filter q` s_1`)` s_2. Unfolding the combinators gives

```
build^τ (λc n ->
    foldr^τ (λx xs ->
        if build^exists (λc1 n1 -> foldr^σ (λz zs -> p z x `c1` zs) n1 s2)
        then x `c` xs else xs) n
            (build^τ (λc2 n2 ->
                foldr^τ (λy ys -> if q y then y `c2` ys else ys) n2 s1)))
```

foldr-build cancellation and unfolding the remaining builds results in

$$\texttt{foldr}^{\tau}(\lambda y\ ys\ \texttt{->}\ \texttt{if q}\ y\ \texttt{\&\& foldr}^{\sigma}(\lambda z\ zs\ \texttt{->}\ \texttt{p}\ z\ y\ \texttt{||}\ zs)\ \texttt{False}\ s_2\ \texttt{then}\ y\ \overset{\tau}{\texttt{:}}\ ys\ \texttt{else}\ ys)\ \texttt{[]}^{\tau}\ s_1$$

No temporary result is allocated at all. The plan scans s_1 and evaluates the semijoin condition only for those objects which satisfy q. Objects finding a join partner w.r.t. p contribute to the result, others are dropped. This is the most efficient plan we can hope for (provided that neither p or q are supported by an index).

Streaming plans for queries like $\texttt{filter}^{\tau}$ p ($\texttt{filter}^{\tau}$ q s) and $\texttt{map}^{\tau}$ f ($\texttt{map}^{\sigma}$ g s) are found along the same lines. For cases like these, deforestation happens to implement well-known algebraic optimization heuristics: the first query is effectively transformed into $\texttt{filter}^{\tau}(\lambda x\ \texttt{->}\ \texttt{q}\ x\ \texttt{\&\&}\ \texttt{p}\ x)\ s$ (two adjacent selections are collapsed into a conjunctive selection) while the second deforests into $\texttt{map}^{\tau}\ (\texttt{f} \circ \texttt{g})\ s$.

The complete combinator algebra is subject to deforestation, due to the uniform representation of collection construction, quantification, and aggregation.

Quantifiers. Deforestation successfully fuses quantifiers with other operators. Deforesting the query $\texttt{exists}^{\tau}$ q ($\texttt{filter}^{\tau}$ p s) gives

$$\texttt{foldr}^{\tau}(\lambda y\ ys\ \texttt{->}\ \texttt{if p}\ y\ \texttt{then q}\ y\ \texttt{||}\ ys\ \texttt{else}\ ys)\ \texttt{False}\ s$$

which is equivalent (using the definitions of if-then-else and (&&)) to

$$\texttt{foldr}^{\tau}(\lambda y\ ys\ \texttt{->}\ (\texttt{p}\ y\ \texttt{\&\&}\ \texttt{q}\ y)\ \texttt{||}\ ys)\ \texttt{False}\ s \quad = \quad \texttt{exists}^{\tau}(\lambda y\ \texttt{->}\ \texttt{p}\ y\ \texttt{\&\&}\ \texttt{q}\ y)\ s$$

If we replace exists by all in the original query we get $\texttt{all}^{\tau}(\lambda y\ \texttt{->}\ (\texttt{not}(\texttt{p}\ y)\ \texttt{||}\ \texttt{q}\ y))\ s$ instead.

Rewritings like these have been developed and added as separate rules to optimizer rule sets [19]. Deforestation implements them “for free” for any combinator composition that allows it.

Aggregates. The principle naturally extends to aggregates. As an example how aggregation fuses with adajcent combinators let us deforest $\texttt{sum}^{\tau}$ ($\texttt{map}^{\tau}$ f s). Unfolding the query gives

$$\begin{array}{l}\texttt{build}^{sum}(\lambda\texttt{c n ->}\\ \quad \texttt{foldr}^{\tau}\texttt{c n (build}^{\tau}(\lambda\texttt{c}_1\ \texttt{n}_1\ \texttt{->}\\ \quad\quad \texttt{foldr}^{\tau}(\lambda x\ xs\ \texttt{->}\ \texttt{f}\ x\ \texttt{`c}_1\texttt{`}\ xs)\ \texttt{n}_1\ s)))\end{array}$$

While this still allocates a intermediate result for the map, its deforested equivalent

$$\texttt{foldr}^{\tau}(\lambda x\ xs\ \texttt{->}\ \texttt{f}\ x\ \texttt{+}\ xs)\ \texttt{0}\ s$$

does not. The collection s is reduced to an *int* value during its consumption. Similar results are readily obtained for count, min, and max (for sum and count this does not hold for $\tau = set$; see Section 4).

We conclude the list of examples with an OQL query involving aggregation, selection, join, and grouping:

$$\begin{array}{rl}\texttt{select} & \texttt{sum}(e\texttt{.sal})\\ \texttt{from} & e\ \texttt{in}\ emp\texttt{,}\ d\ \texttt{in}\ dept\\ \texttt{where} & e\texttt{.dno = }d\texttt{.no and }d\texttt{.budg < 10000}\\ \texttt{group by} & d\texttt{.no}\end{array} \qquad (Q_2)$$

[7] devotes a whole paper to develop a specialized laborious transformation for a simpler query (no join involved). Query deforestation immediately comes up with a streaming program that computes the sum aggregates during the grouping phase so that the actual groups need not be allocated.

`nestjoin` implements the join as well as the grouping so that an equivalent combinator plan for this query is

```
map^bag sum^bag (nestjoin^bagbag (λe d -> e.dno = d.no) (λe d -> e.sal)
   (filter^bag (λb -> b.budg < 10000) dept) emp)
```

The groups computed by `nestjoin` are subsequently aggregated by `map sum` (employing the higher-order nature of the algebra).

Deforestation then proceeds as follows (we omit some steps to save space):

```
foldr^bag (λs ss -> foldr^bag (+) 0 s :^bag ss) []^bag
   build^bag (λc n ->
      foldr^bag (λd ds ->
         build^bag (λc1 n1 ->
            foldr^bag (λe es ->
               if e.dno = d.no then e.sal `c1` es else es) n1 emp) `c` ds)
         n (build^bag (λc2 n2 ->
            foldr^bag (λb bs -> if b.budg < 10000 then b `c2` bs else bs) n2 dept))
```

Deforestation of `map` results in the `sum` and `nestjoin` being adjacent:

```
foldr^bag (λd ds ->
   foldr^bag (+) 0 (build^bag (λc1 n1 ->
      foldr^bag (λe es -> if e.dno = d.no then e.sal `c1` es else es) n1 emp)) :^bag ds)
   []^bag (build^bag (λc2 n2 ->
         foldr^bag (λb bs -> if b.budg < 10000 then b `c2` bs else bs) n2 dept)
```

This opens the possibility to finally merge the aggregate into the grouping phase to give the resulting streaming plan in which aggregation, join, and grouping have been completely fused:

```
foldr^bag (λd ds ->
   if d.budg < 10000
      then (foldr^bag (λe es -> if e.dno = d.no then e.sal + es else es) 0 emp) :^bag ds
      else ds) []^bag dept
```

We have found deforestation to significantly extend the work of [7] as well as [8] which, furthermore, was solely list-based. Cheap deforestation avoids complications found there (e.g. the so-called *fold* and *union* steps), leads to results faster, and is applicable to programs which were not expressible in former work on `fold` fusion. This includes the *unsafe programs* of [5], queries in which a nested subquery traverses partial results computed by an outer query. Instances of this class of programs, elements of which are aggregate queries that compute running sums or list reversal, are not accessible to the *fold promotion theorems* developed in [5] or [17] and thus cannot be fused with adjacent expressions.

4 Optimization and Implementation Issues

For the sake of simplicity we have post-poned questions of correctness and further implementation issues until now. This section makes up for it. We also sketch an alternative shortcut (in the sense of "rapid prototyping") to declarative query language implementation based on comprehensions, which, using the methods presented here, may still lead to an efficient implementation.

Observe that the query engine cannot rely on a fixed order iteration whenever a plan uses an iterator to access a set or bag: due to ($\overset{bag}{:}$) and ($\overset{set}{:}$) being left-commutative the persistent storage has, as opposed to lists, the freedom to retrieve the elements in any order. The same element may even be encountered more than once in the case of $\texttt{foldr}^{set}$ (which of course is due to the left-idempotence of ($\overset{set}{:}$)).

This has an impact on the well-definedness of $\texttt{foldr}^{\mathcal{T}}$ f z: whenever $(\overset{\mathcal{T}}{:})$ is left-idempotent/left-commutative, 'f' has to exhibit at least the same properties. Only then we are able to assign a meaning to $\texttt{foldr}^{\mathcal{T}}$ f z which is independent of iteration order or multiplicity of elements. No prerequisites are necessary for $\texttt{foldr}^{list}$ to be well-behaved.

Although asking whether 'f' has the needed properties is undecidable [2] this does not affect query deforestation too seriously:

(a) From a theoretical point of view, given that the query compiler produces well-behaved algebraic expressions only (which is the case for the OQL compiler discussed in [13], for example), $\texttt{foldr}^{\mathcal{T}}$ f z $(\texttt{build}^{\mathcal{T}} g)$ say, we may replace $(\overset{\mathcal{T}}{:})$ by 'f' in g without sacrificing well-definedness since 'f' is left-commutative/left-idempotent whenever $(\overset{\mathcal{T}}{:})$ is. In other words, the deforestation rule $\texttt{foldr}^{\mathcal{T}}$ f z $(\texttt{build}^{\mathcal{T}} g) \;=\; g$ f z is safe.

(b) Almost all implementations of query engines are actually list-based. This is mainly due to the apparent cost of duplicate elimination inherent in $(\overset{set}{:})$ and actually enables the engine to reason about (sorting) orders of their object streams. Problems with ill-behaved expressions do not occur in such systems since $\texttt{foldr}^{list}$ is always well-defined. These query engines typically use a separate dupelim combinator to enforce set semantics if needed.

Query engines of this type are naturally represented in our setup as dupelim may be expressed by the quey engine core language as well. The query engine then has the option to delay duplicate elimination based on the equivalence

$$\texttt{build}^{set}\; g \;\overset{\rightarrow}{\equiv}\; \texttt{dupelim}(\texttt{build}^{list}\; g)$$

(read in the direction of the arrow, this allows to "push duplicate elimination out" of the plan and thus enables efficient list-based processing of g). This rule is of particular significance in our context since the implementation of dupelim typically involves sorting or memoization which disrupts the fully stream-based execution of query plans.

Asymmetry of join. As the foldr representation of join p f s_1 s_2 (of all binary operators actually) is not symmetric in s_1 and s_2 the plan generator should keep an eye on which argument assign to s_2, over which the inner foldr (loop) iterates. Clearly s_2 should be assigned the cheaper (in terms of an appropriate cost model) of the two inputs. Should it happen that the inputs have to be exchanged, the plan generator may do so by means the rule (let flip f x y = f y x)

$$\texttt{join p f}\; s_1\; s_2 \;=\; \texttt{join (flip} \circ \texttt{p) (flip} \circ \texttt{f)}\; s_2\; s_1$$

Comprehensions for rapid prototyping. While the compilation of OQL into efficient algebraic combinator plans can be complicated, the language is easily translated into *monad comprehensions* via a syntactic mapping [13]. The mapping is based on a generalization of list comprehensions (known from functional programming languages) to monadic types.[5]

In the comprehension $[\, e \mid q_1, \ldots, q_n]^{\mathcal{T}}$ the *qualifiers* q_i are either *generators* $x \leftarrow q$ or *predicates* p. A generator q_i sequentially binds variable x to the elements of its range q; x is bound in $q_{i+1}, \ldots, q_n$ and e. The binding of x is propagated until a predicate evaluates to False. The *head* e is evaluated under all bindings that pass and the function results are then accumulated using $(\overset{\mathcal{T}}{:})$. Notice that the relational calculus may be understood as a specific instance of the monad comprehension calculus.

As we cannot provide more than an intuition of the OQL to comprehension translation here we refer the interested reader to [6] and [13]. To give just two examples, queries Q_1 and Q_2 from Section 3 respectively translate into

$$Q_1 \;=\; [\, \texttt{f}\; x \mid x \leftarrow es, \texttt{p}\; x]^{bag}$$

and

$$Q_2 \;=\; [\, [\, e.\texttt{sal} \mid e \leftarrow emp, d' \leftarrow dept, e.\texttt{dno} = d'.\texttt{no}, d.\texttt{no} = d'.\texttt{no}]^{sum} \mid d \leftarrow dept, d.\texttt{budg} < 10000]^{bag}$$

[5] The types we used for the (insertion) representation of queries are easily made monad instances.

A second syntactic translation scheme, $\mathcal{MC}$ in Figure 5 (a generalization of a similar scheme presented in [9]), already completes the prototypical OQL compiler. $\mathcal{MC}$ uses structural recursion on the list of qualifiers to translate comprehensions into nested `foldr`s, which is admittedly naive and, at least in general, cannot compete with an algebraically optimized plan. Since $\mathcal{MC}$ emits expressions in the desired `foldr-build` form, subsequent deforestation can, however, turn these expressions into streaming plans, which partly makes up for the naivety of the previous phases.

$$
\begin{array}{lcl}
\mathcal{MC}\ \texttt{[}\,e \mid qs\texttt{]}^{\mathcal{T}} & = & \texttt{build}^{\mathcal{T}}(\lambda\texttt{c n ->}\ \mathcal{C}\ (\texttt{[}\,e \mid qs\texttt{]})\ \texttt{c n}) \\
\mathcal{MC}\ e & = & e \\
 & & \\
\mathcal{C}\ \texttt{[}\,e \mid \texttt{]}\ \texttt{c n} & = & (\mathcal{MC}\ e)\ \texttt{`c`}\ \texttt{n} \\
\mathcal{C}\ \texttt{[}\,e \mid p, qs\texttt{]}\ \texttt{c n} & = & \texttt{if}\ \mathcal{MC}\ p\ \texttt{then}\ \mathcal{C}\ \texttt{[}\,e \mid qs\texttt{]}\ \texttt{c n else n} \\
\mathcal{C}\ \texttt{[}\,e \mid x \leftarrow q, qs\texttt{]}\ \texttt{c n} & = & \texttt{foldr}\ (\lambda x\ xs\ \texttt{->}\ \mathcal{C}\ \texttt{[}\,e \mid qs\texttt{]}\ \texttt{c}\ xs)\ \texttt{n}\ q
\end{array}
$$

Figure 5: A mapping of comprehensions to the query engine's core language.

Implementing `foldr`. To conclude this section, note that neither the definition of `foldr` nor the definitions of other combinators necessarily involves pattern matching on collections. Indeed `foldr` may be reexpressed in terms of Graefe's operator model as

$$
\begin{array}{lll}
\texttt{foldr f z}\ xs & = & \texttt{if}\ (\textit{end-of-stream?}\ xs)\ \texttt{then z} \\
 & & \texttt{else let}\ (x, xs')\ \texttt{=}\ \textit{next}\ xs\ \texttt{in f}\ x\ (\texttt{foldr f z}\ xs')
\end{array}
$$

The linear recursion implemented by `foldr` may then easily be replaced by iteration during the final mapping to the query engine's (typically imperative) implementation language.

5 Outlook

While the algebraic optimization phase of query language compilers is reasonably well understood and backed up by formal methods this does not hold for plan generators in general. We have proposed the use of query deforestation to overcome this deficiency: generalized to types with insertion representation, `foldr-build` deforestation provides a formal setting in which the derivation of efficient streaming programs from algebraic combinator queries is achieved by program transformation techniques much like in algebraic optimizers. Complex object algebra expressions as well as comprehensions are acceptable as input to the deforestation process which, at the same time, appears to be simpler than previous proposals.

Further program optimizations developed by the functional programming community are applicable as the query engine core actually uses an extended λ-calculus as its representation language. We already got promising results from experiments that employed techniques like λ-lifting, static argument transformation, and partial evaluation. Since deforestation also applies to tree-like data types, this may open possibilities to interleave the actual query execution with index lookups (e.g., in B^+-trees) in an efficient manner. Again, this will provide formal access to a problem area that has only been tackled on the implementation level so far.

To provide a proof of concept we have already implemented a query compiler for OMDG's OQL based on comprehensions and `foldr`. Its final stage emits query-specific C++ code. The compiler is currently in use in our prototypical CROQUE object base system [10] in which the C++ query code is compiled, dynamically linked, and then loaded at runtime to provide an ad-hoc query facility. The compiler stages prior to C++ code generation can be test-driven via a web interface at URL `http://www.informatik.uni-konstanz.de/~grust/cgi-bin/OQL-foldr/`.

References

[1] C. Beeri and Y. Kornatzky. Algebraic Optimization of Object-Oriented Languages. In *Proc. of the 3rd Int'l Conference on Database Theory (ICDT)*, pages 72–88, Paris, France, December 1990.

[2] V. Breazu-Tannen and R. Subrahmanyam. Logical and Computational Aspects of Programming with Sets/Bags/Lists. In *Proc. of th 18th Int'l Colloquium on Automata, Languages and Programming*, pages 60–75, Madrid, Spain, July 1991.

[3] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of Programming with Complex Objects and Collection Types. *Theoretical Computer Science*, 149(1):3–48, 1995.

[4] R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[5] L. Fegaras. Efficient Optimization of Iterative Queries. In *Proc. of the 4th Int'l Workshop on Database Programming Languages (DBPL)*, pages 200–225, August 1993.

[6] L. Fegaras and D. Maier. Towards an Effective Calculus for Object Query Languages. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, May 1995.

[7] J.C. Freytag and N. Goodman. Translating Aggregate Queries into Iterative Programs. In *Proc. of the 12th Int'l Conference on Very Large Data Bases (VLDB)*, pages 138–146, Kyoto, Japan, August 1986.

[8] J.C. Freytag and N. Goodman. On the Translation of Relational Queries into Iterative Programs. *ACM Transactions on Database Systems*, 14(1):1–27, March 1989.

[9] A.J. Gill. *Cheap Deforestation for Non-strict Functional Languages.* Ph.D. dissertation, Department of Computing Science, University of Glasgow, January 1996.

[10] D. Gluche, T. Grust, C. Mainberger, and M.H. Scholl. Incremental Updates for Materialized OQL Views. In *Proc. of the 5th Int'l Conference on Deductive and Object-Oriented Databases (DOOD'97)*, pages 52–66, Montreux, Switzerland, December 1997.

[11] G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, pages 102–111, Atlantic City, New Jersey, USA, 1990.

[12] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[13] T. Grust, J. Kröger, D. Gluche, A. Heuer, and M.H. Scholl. Query Evaluation in CROQUE—Calculus and Algebra Coincide. In *Proc. of the 15th British National Conference on Databases (BNCOD)*, pages 84–100, London, Birkbeck College, July 1997.

[14] P. Hudak, S.L. Peyton Jones, P. Wadler, et al. Report on the Programming Language Haskell. *ACM SIGPLAN Notices*, 27(5), 1992.

[15] J. Launchbury and T. Sheard. Warm Fusion: Deriving Build-Catas from Recursive Definitions. In *Proc. of the ACM Conference on Functional Programming and Computer Architecture (FPCA)*, La Jolla, California, June 1995.

[16] D.F. Lieuwen and D.J. DeWitt. A Transformation Based Approach to Optimizing Loops in Database Programming Languages. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, pages 91–100, San Diego, California, June 1992.

[17] A. Poulovassilis and C. Small. Formal Foundations for Optimising Aggregation Functions in Database Programming Languages. In *Proc. of the 6th Int'l Workshop on Database Programming Languages (DBPL)*, Estes Park, Colorado, USA, 1997.

[18] C. Rich, A. Rosenthal, and M.H. Scholl. Reducing Duplicate Work in Relational Join(s): A Unified Approach. In *Proc. of the Int'l Conference on Information Systems and Management of Data (CISMOD)*, pages 87–102, Delhi, India, October 1993.

[19] H.J. Steenhagen. *Optimization of Object Query Languages.* Ph.D. dissertation, Department of Computer Science, University of Twente, 1995.

[20] D. Suciu and L. Wong. On Two Forms of Strutural Recursion. In *Proc. of the 5th Int'l Conference on Database Theory (ICDT)*, pages 111–124, Prague, Czech Republic, January 1995.

[21] B. Vance. Towards an Object-Oriented Query Algebra. Technical Report 91–008, Oregon Graduate Institute of Science & Technology, January 1992.

[22] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.