EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Lehrstuhl für Datenbanksysteme
Wilhelm-Schickard-Institut für Informatik
Mathematisch-Naturwissenschaftliche Fakultät

Diplomarbeit in Informatik

A Ferry-Based Query Backend for the Links Programming Language

Alexander Ulrich

March 31, 2011

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Lehrstuhl für Datenbanksysteme
Wilhelm-Schickard-Institut für Informatik
Mathematisch-Naturwissenschaftliche Fakultät

Diplomarbeit in Informatik

A Ferry-Based Query Backend for the Links Programming Language

Ein Ferry-basiertes Query-Backend für die Programmiersprache Links

Author:	Alexander Ulrich
Supervisor:	Prof. Dr. Torsten Grust
Advisor:	Tom Schreiber, M.Sc.
Date:	March 31, 2011

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Tübingen, den 31. März 2011 Alexander Ulrich

Abstract

This thesis describes the implementation of a FERRY-based query backend for the LINKS programming language. In LINKS, queries are seamlessly embedded into the language: Queries formulated in a subset of the language are translated into single SQL queries. LINKS uses static checks to ensure that a type-correct query expression can be translated into an equivalent SQL query and allows abstraction over parts of a query. The queryizable subset of LINKS is, however, severely limited in terms of supported functions and the data type (limited to bags of flat records) of queries. The thesis begins with a description of the query facility and criticizes the limited functionality of the queryizable LINKS subset.

The FERRY framework deals with the compilation of pure, declarative languages based on list comprehensions into SQL queries. It provides features that LINKS queries are lacking: query results involving nested lists and computed by a small, statically bounded number of SQL queries, ordered lists semantics and a larger number of supported functions.

The thesis first reviews the compilation technique of the FERRY framework and adapts it to the specifics of LINKS. The queryizable subset of LINKS is higher-order and allows to treat functions as first-class values. To keep this property in the new query backend, the thesis extends the FERRY framework to deal with first-class functions. To make a larger part of a functional programmer's toolset available in queries, the framework is also extended to handle variants.

In a second step, the thesis describes the architecture of a query backend that embeds this algebraic compilation into the LINKS compiler. This involves changes to the typing mechanisms that statically check LINKS expressions for queryizability and query-specific optimizations.

The benefits of the new query backend are demonstrated with the help of a number of example queries. For simple queries, compact and idiomatic SQL code is generated. For more complex queries, the relocation of computations to a database leads to a substantial performce gain.

In summary, the new backend considerably extends the queryizable subset of LINKS and makes more database functionality available and at the same time keeps the safe, statically checked and compositional integration of queries.

Acknowledgements

This thesis would not have been possible without the help and support of multiple people to whom I would like to express my gratitude.

I am most grateful to my supervisor Torsten Grust for giving me the chance to work on this project, and for his guidance and encouragement. Torsten included me in his research group and provided me with the most comfortable working environment a student could wish for. Torsten also made very helpful comments on the manuscript.

Torsten Grust's database research group has been a most welcoming place. Of the team, I would first and foremost like to thank my thesis advisor Tom Schreiber, who was after all successful in dragging me into the database world. Tom has always been available and enthusiastic when I needed advice. Kudos to Jan Rittinger, the query plan whisperer[1], whose skills in digging into and making sense of Really Large Query Plans are impressive. Jan saved me multiple times when I gost lost in a plan during debugging. I would also like to thank Jeroen Weijers, George Giorgidze and Manuel Mayr for interesting and insightful discussions.

I am grateful to Sam Lindley from the University of Edinburgh, who gave helpful hints for the direction of this project and was always willing to help when I got into trouble with the LINKS compiler.

Finally, thanks to Anne and Sebastian for proof-reading and sane-keeping. Above all, I owe to my parents for believing in and supporting me all those years.

[1] With apologies to Robert Redford and Jan Rittinger.

Contents

1. Introduction

For the last decades, general programming languages and database query languages have been developed largely independent of each other. As of yet, querying is dominated by textual embedding of query code into host language programs (e.g. ODBC, JDBC). Typically, a programmer formulates the query in a query language like SQL and embeds the SQL text into the source code of the host program. Because of the semantic gap between the world of programming languages and the world of databases, this approach is, however, troublesome. Relational databases operate with relations of flat, unordered data, while usual programming languages feature a richer set of possibly ordered and nestable data structures. A programmer working with both has to keep these different representations in mind and convert between them, and also has to work with the different syntaxes of the two languages. In contrast to the essential abstraction capabilities of programming languages, such approaches typically – if at all – support only a very limited amount of abstraction over parts of a query. As Lindley and Wadler put it: "The problem is simple: two languages are more than twice as difficult to use as one language" [22].

Another approach that has gained prominence in recent years is that of *language-integrated query*, as pioneered e.g. by Microsoft's LINQ. The aim is to seamlessly embed queries into the host programming language by defining a subset of the host language that can be efficiently compiled into database query code. In this way, queries can be formulated using the native syntax of the host language, and the programmer can – conceptually – stay in the familiar data model of the host language. This data model has to be represented on the database by a suitable encoding.

One proponent of language-integrated queries is the LINKS programming language. Designed as a multi-tier language for web appplication development, subsets of LINKS can be compiled into JAVASCRIPT to be executed on a web client, and into SQL queries to be executed on a relational database. LINKS emphasizes the safe integration of queries: Queries are, like the rest of LINKS, strongly typed and static checks ensure that only those queries pass the type checker that can indeed be translated to SQL queries. LINKS allows abstraction over parts of a query and therefore allows to assemble queries at runtime in a safe way. Unfortunately, the expressiveness of the LINKS subset allowed in queries is severely limited. Queries must only compute flat values (i.e. lists of records of base types) and only a very limited amount of database functionality is available. These restrictions reduce databases more than necessary to a data storage role and do not allow to fully leverage their highly optimized query processing infrastructure.

The FERRY project explores how large a subset of a programming language can be translated into efficient queries with the goal of *database-supported program execution*. Databases should seamlessly take part in the evaluation of programs and especially serve as coprocessors for data-intensive computations. Centered around list comprehensions as a semantical basis, FERRY's compilation technique allows queries with nested and ordered results while generating only small, statically determined numbers of queries, and sup-

Language	Example
SQL	`SELECT x FROM y WHERE z`
LINKS programs	`fun f(x) { x + 2 }`
Internal representation of LINKS programs	concatMap(λx.x+2, l)

Table 1.1.: Different fonts used in listings.

ports a rich library of functions. With this feature set, it provides a good basis to overcome the limitations of the LINKS query facility.

To quote Philip Wadler, one of the designers of LINKS:

> "And there is another group now at Tübingen, Torsten Grust and his students, who are working on a system called FERRY [...] They handle many more constructs than we do [in LINKS] and they figured out how to generalize this to work over lists rather than just bags. So I think FERRY is very much worth paying attention to in terms of extending what you can do, in terms of how powerful a language you can compile into a database." [1]

This thesis aims to explore how LINKS can benefit from FERRY.

1.1. Roadmap

The remainder of this thesis is organized as follows: Chapter 2 briefly describes the LINKS programming language, especially it's integrated query facility . We point out certain defiencies of this query integration and outline which improvements the FERRY-based backend achieves. In Chapter 3, we describe the core of the FERRY compilation technique: the compilation of a pure, functional language into query plans over a relational algebra. Chapter 4 describes how this algebraic compilation is embedded in the LINKS compiler. Chapter 5 compares the current and the FERRY-based query backend with respect to the SQL queries they generate and assesses performance benefits that result from FERRY's ability to perform more computations on the database. Finally, in Chapter 6, we point to related work and conclude.

1.2. Notation

We use different fonts to differentiate between code of different programming languages and representations. Table 1.1 gives examples for the usage of fonts.

[1] Transcript of Philip Wadler's talk "Not lost in translation: How to write SQL in your own language". Available online: http://download.microsoft.com/download/7/6/A/76A69AE5-72B5-4005-BBD9-7EA5F4795014/29-PhilipWadler-Links.wmv

2. Background and Motivation

In this chapter, we first describe aspects of the LINKS language that are relevant for this thesis, concentrating on the query integration (Section 2.1). We then point out deficiencies of the current query integration and outline in what ways the FERRY-based query backend overcomes these deficiencies (Section 2.2).

During the course of this thesis we will refer to the current version of LINKS with the query facility as described in [4] with its code name "Murrayfield".

2.1. Queries in LINKS

Basically, LINKS is a functional, impure, strong and statically typed programming language. Its type system is based on Hindley-Milner type inference, allowing parametric polymorphism. The type system includes row variables, allowing polymorphically extensible records and polymorphic variant types. Rows map record fields or variant tags to types, and row variables range not over types but over complete rows of types [24]. For example, the *open* record type $(a : Int, b : Int \mid r)$ is to be read as "type of a record that includes *at least* the fields a of Int and b of Int" and the type $(\mid r)$ is a completely polymorphic record type. In contrast, a *closed* record type $(a : Int, b : Int)$ contains exactly the fields listed. For future reference, Figure 2.1 shows a simplified grammar of LINKS types. Types include base types, type variables, function or arrow types, list types as well as record and variant types. Record and variant types can be open or closed. LINKS includes a set of primitive functions and a standard library of functions implemented in LINKS itself. For an in-depth description of LINKS we refer to [4].

LINKS' aim is to reduce the impedance mismatch problem occuring in web application development [6]. A typical web application consists of three tiers: core logic running on the server (implemented e.g. in RUBY or JAVA), queries against a relational database (formulated in SQL) and code running on the web browser (implemented in JAVASCRIPT). This requires developers to use at least three different languages, all with different syntax, expressiveness and data models. LINKS eases this problem by providing an execution model which unifies the three tiers: A single LINKS program is executed on all three tiers by translating LINKS code into JAVASCRIPT code for the client and SQL code for the database at runtime, and by interpreting the core logic on the server. In the following, we concentrate on the translation of database queries expressed in LINKS code into SQL queries.

LINKS embeds database queries (mostly) seamlessly and safely into the language: Queries are formulated using the exact same syntax and typechecked in the same way as normal LINKS expressions. Like the rest of the language, queries are fully compositional, allowing to use arbitrary expressions as part of a query expression as long as they are queryizable (see below). Queries can contain free variables like any other LINKS expression. This

$$
\begin{array}{rrcll}
(\textit{types}) & \tau & ::= & Float \mid Char & (\textit{base types}) \\
 & & \mid & Int \mid Bool \mid String & \\
 & & \mid & [\tau] & (\textit{list type}) \\
 & & \mid & (\overrightarrow{l : \tau}) \mid (\overrightarrow{l : \tau} \mid R) \mid (\mid R) & (\textit{record type}) \\
 & & \mid & [\mid \overrightarrow{C : \tau} \mid] \mid [\mid \overrightarrow{C : \tau} \mid R \mid] \mid [\mid R \mid] & (\textit{variant type}) \\
 & & \mid & (\vec{\tau}) \to \tau & (\textit{function type}) \\
 & & \mid & V & (\textit{type variable}) \\
(\textit{row variables}) & R & ::= & r \mid r :: Base & \\
(\textit{type variables}) & V & ::= & \alpha \mid \alpha :: Base & \\
(\textit{field labels}) & l & & & \\
(\textit{variant tags}) & C & & &
\end{array}
$$

Figure 2.1.: Simplified grammar of LINKS types.

```
fun playersByAge(pred) {
  for (p <-- players) where (pred(p)) orderby (p.age)
    [(name = p.name)]
}
```

Listing 2.1: Query $Q1$.

makes it possible to abstract e.g. over sub-queries or predicates and thus to construct queries at runtime. LINKS queries center around list or table *comprehensions*. Semantically, a list comprehension `for (x <- l) e` maps an input list `l` to an output list by binding the iteration variable `x` consecutively to each element of `l` and evaluating the body expression `e` for each binding. The comprehension's body must be list typed and the overall result is constructed by concatenating the resulting lists from the evaluation of the body expression for all bindings. Comprehensions have been found to be a good basis for the embedding of queries into programming languages [20]. A simple typical query using LINKS' comprehension syntax might look as in Listing 2.1.

This example demonstrates abstraction over parts of a query: the predicate `pred` is used freely in the query. Any predicate that is appropriately typed and matches the criteria for SQL translatability (see below) can be used in this place and the type system statically assures that only queryizable predicates are used. The long arrow `<--` indicates that the generator of the comprehension uses a database table and not a list. Semantically, `t` is bound consecutively to each row in the database table referred to by the table handle `players`. **`where`** and **`orderby`** clauses are syntactic sugar for filtering and ordering the generating table. To simply fetch the whole content of a table into a list, the function **`fun`** `asList(t) { for (r <-- t) [r] }` is provided.

Which parts of a LINKS program actually are to be translated to an SQL query is determined by explicit or implicit annotations. Wrapping an expresssion `e` with an annotation **`query`** `{ e }` (a *query block*) forces the compiler to either translate it to a single SQL statement at runtime or fail at compile time if `e` can't be translated. Table comprehensions using the $<--$ arrow implicitly incur a query annotation.

At compile time, the LINKS compiler statically checks if the query-annotated expression

concatMap	$((\alpha) \to [\beta], [\alpha]) \to [\beta]$	monadic list mapping
map	$((\alpha) \to \beta, [\alpha]) \to [\beta]$	list mapping
filter	$((\alpha) \to Bool, [\alpha]) \to [\alpha]$	filter list by predicate
sortByBase	$((\alpha \to (\|\_ :: Base), [\alpha]) \to [\alpha]$	sorting
length	$([\alpha]) \to Int$	list length
empty	$([\alpha]) \to Bool$	test for empty list

Figure 2.2.: Effect-afflicted functions from the LINKS standard library that can be used in queries (Murrayfield).

is fit to be translated to a single SQL statement, i.e. *queryizable*. To be queryizable, a LINKS expression has to comply with multiple restrictions:

- It must not use recursion.
- It must be *pure*, i.e. it must not use any language constructs which generate side-effects.
- It must not use any primitive function for which no SQL equivalent exists.
- Both the type of the query and of any included table comprehensions has to be a *relation type*, i.e. a list of records of atomic or *base* types. This corresponds to the types that a single SQL query can return due to first normal form restrictions.

Note that these restrictions do not forbid free use of lambda abstractions. Only recursive functions are forbidden. This means that the queryizable subset of LINKS is higher-order.

The restrictions are enforced during the regular type checking on LINKS programs by two mechanisms. An *effects analysis* forbids the use of side-effects and recursive functions. Such constructs are annotated with an effect noqy (no query). Additionally, primitive functions which do not have a direct SQL equivalent also are annotated with this effect. In a nutshell, for an expression the analysis unifies any recursively obtained effects from the sub-expressions and possibly adds its own effect. If the effect set of a query-annotated expression contains the effect noqy, the expression is not queryizable. A set of functions from LINKS' standard library listed in Figure 2.2 maps directly to SQL constructs . Although these functions actually incur an effect because they are implemented recursively, they are treated as queryizable.

The restriction on the type of query blocks and table comprehensions is enforced by a *kinding discipline*. Again in a nutshell, *subkinds* can be attributed to type variables in order to restrict the set of types to which a type variable can be instantiated. For example, the subkind $Base$ restricts a type variable $\alpha :: Base$ to be instantiated only to base types, i.e. the base types from Figure 2.1. Subkinds can also be attributed to row variables, so that the type variable for every field of the row is restricted by this subkind. Subkinds are employed during unification to check that the inferred type of table comprehensions and query blocks has the necessary shape. For example, the type inferred from a query block is not unified with an unrestricted type variable α (which would not impose any restrictions on the type), but with the type $[(|r :: Base)]$, describing a list of records of base values. If the unification fails because the inferred type does not meet this shape, the query is rejected as not queryizable.

```
  SELECT players.name
    FROM players
   WHERE players.age > 20
         OR players.age < 10
ORDER BY players.age
```

Listing 2.2: SQL code generated by the Murrayfield query translator for Query $Q1$.

The combination of effects analysis and subkind restrictions on type variables provides a strong static assurance that a type-correct query-annotated LINKS expression can indeed be compiled to an SQL query. Especially, it statically ensures that free variables of queries are only given queryizable values at runtime so that query construction at runtime is safe. In fact, the transformation for a limited subset of LINKS expressions has been proven correct and complete [5]. The actual implementation of this transformation in the LINKS compiler leaves very little loopholes on which the transformation might fail.

We conclude this section by coming back to Query $Q1$. Assume that at runtime the function `playersByAge` is applied to the predicate

```
fun (p) { p.age > 20 || p.age < 10 }
```

The predicate does not have the noqy effect, so that it can be safely used in the query. At runtime, the query is then translated by means of a rewrite system [5] to the equivalent SQL expression in Listing 2.2.

2.2. Deficiencies and Remedy

LINKS has certainly advanced the safe and seamless integration of database queries to a desirable level. Especially the effects analysis to exclude side effects and general recursion and to categorize primitive functions into queryizable and non-queryizable ones is an elegant way to ensure that no language constructs are used in queries which can not be translated. The integration of queries into the strong typing discipline provides a good mean to check for errors in queries. However, the integration still leaves much to be desired in terms of expressibility, faithful realization of host language semantics and access to database features. Two points of view are to be considered: The queryizable subset of LINKS should serve as a versatile query language. This means that as much as possible of a database's functionality should be accessible from within LINKS. On the other hand, relational databases should serve as co-processors for data-intensive LINKS programs. Under this aspect, as much as possible of LINKS's expressibility should be available in queries.

Ordered and nested lists are an important part of LINKS' data model. However, the query translator directly maps queries formulated in LINKS and using ordered lists to SQL queries over inherently unordered relational databases. This means that the query facility has faces difficulties implementing the ordered list semantics. The formalized query translation does not cope with order at all, operating only over unordered bags [5]. Although the actual implementation of this translation allows the usage of order-aware constructs equivalent to SQL's **`ORDER BY`** and **`LIMIT ... OFFSET`** functionality, it still can't implement ordered list semantics faithfully. We illustrate this problem using the (admittedly

```
query {
  for (i <- [10, 20]
    for (p <-- players)
      [if (p.age > 20) (x = p.age + i) else (x = p.age - i)]
}
```

Listing 2.3: Query $Q2$

```
(SELECT
  (CASE WHEN age > 20 THEN age+10 ELSE age-10 END) AS x
 FROM players)
UNION ALL
(SELECT
  (CASE WHEN age > 20 THEN age+20 ELSE age-20 END) AS x
 FROM players)
```

Listing 2.4: SQL query generated from query $Q2$ (Listing 2.3).

rather artificial) query $Q2$ (Listing 2.3) which is translated to the SQL query shown in Listing 2.4.

The iteration over the literal outer list is implemented with the unordered set operator **UNION ALL**. Although the result might be delivered in the expected order, the order is semantically undefined.

In terms of functionality, the queryizable LINKS subset is not really satisfying. Under the two aspects mentioned above (LINKS as a query language and databases as co-processors for data-intensive programs), quite a number of points can be criticized:

- Only a modest amount of functions from the LINKS standard library can be used in queries. For example, most functions working on lists are implemented recursively and are therefore not queryizable.
- The queryizable subset of LINKS also lacks functionality when regarded as a query language. It does not provide access to common database functionality, including the usual aggregate functions `MAX`, `MIN` etc., computation of the unique elements of a list or table (**DISTINCT**) and grouping (**GROUP BY**). Ordering of tables is possible in an ascending order (`ASC`), but not in a descending order (`DESC`). All this functionality can of course be implemented by fetching the raw data from the database and computing the equivalent functions in native LINKS code. However, in general this will be much less efficient than a utilization of the highly optimized database functionality.
- Although LINKS goes to great lengths to ensure that queries which can not be translated are rejected statically, some loopholes remain: LINKS queries which are accepted by the type system can under rare circumstances not be translatable. These include regular expressions which can't always be mapped to the equivalent SQL operator, and the use of comparison operators on non-atomic values.
- The most severe deficit is the restriction of queries to lists of records of base values which is due to the translation to single SQL queries. Queries are allowed to produce

intermediate nested results and the translation takes care of un-nesting them. Query results however must not be nested, must not contain e.g. variant values and must not even be of just a base type. For instance, the simple query

```
query { length(asList(t)) }
```

is rejected and its result must be wrapped in a list and a record:

```
query { [(len=length(asList(t)))] }
```

The inability to perform computations in queries resulting from these deficiencies means that those computations must be carried out in the normal LINKS heap. This is unfavorable for multiple reasons. Generally more data has to be transferred from the database to the LINKS heap, meaning that costs for data serialization and inter-process communication increase. Consider an aggregate function like `max`, that reduces a whole table column or list to a single scalar value. If the aggregate is performed natively in LINKS, the complete column has to be transferred to the LINKS heap, compared to a single integer when the aggregate is computed by the database. The implementation of most common operations in databases is extremely efficient, profiting amongst others from indices and optimization based on statistical information about data. Computations in the heap can not make use of these facilities. Finally, the inability to compute nested results means that e.g. inner lists must be computed individually and assembled in the heap. This can result in a number of queries that can not only be large but – more problematic – depends on the size of the data.

We illustrate the effect of these deficiencies with the help of the often needed function groupBy $:: (((\alpha) \to \beta), [\alpha]) \to [(\beta, [\alpha])]$. As arguments it takes a list with element type α and a projection function which maps a list element to a grouping key of type β. The result consists of a list of tuples in which the first element is the grouping key and the second element is the list of elements which share this grouping key. Grouping is a common requirement in queries. `groupBy` can straight forwardly be implemented as follows:

```
fun groupBy(p, xs) {
  var ks = nub(map(p, l));
  for (k <- ks)
    var elts = for (x <- xs) where (p(x) == k) [x];
    [(k, elts)]
}
```

Listing 2.5: Function `groupBy`.

In general, two things are necessary to implement `groupBy`: the list of distinct grouping keys and the ability to construct the nested result. In SQL queries, **`DISTINCT`** can be used to reduce a table to its unique elements. In LINKS, equivalent functionality is not available in queries, so that the list of the result of the projection function for all elements would have to be fetched in a query. This list would then have to be – potentially inefficiently – reduced to the list of distinct keys. But even if this part could be performed on the database, the final result could still not be constructed because it is nested. This means that every inner list would need to be fetched on its own, resulting in one query for every distinct grouping key. In effect, the number of queries would depend on the size of the

concatMap	$((\alpha) \to [\beta], [\alpha]) \to [\beta]$	monadic list mapping
map	$((\alpha) \to \beta, [\alpha]) \to [\beta]$	list mapping
mapi	$((\alpha, Int) \to \beta, [\alpha]) \to [\beta]$	indexed list mapping
concat	$([[\alpha]]) \to [\alpha]$	list flattening
take, drop	$(Int, [\alpha]) \to [\alpha]$	keep list prefix/suffix
select	$([\alpha], Int) \to \alpha$	positional list access
zip	$([\alpha], [\beta]) \to [(\alpha, \beta)]$	two-way positional
unzip	$([(\alpha, \beta)]) \to ([\alpha], [\beta])$	merge and split
reverse	$([\alpha]) \to [\alpha]$	list reversal
filter	$((\alpha) \to Bool, [\alpha]) \to [\alpha]$	filter list by predicate
groupByFlat	$(((\alpha) \to \beta :: Flat), [\alpha]) \to [(\lvert\beta, [\alpha])]$	grouping
sortByflat	$((\alpha \to (\lvert\beta :: Flat), [\alpha]) \to [\alpha]$	sorting
length	$([\alpha]) \to Int$	list length
sum	$([Int]) \to Int$	list aggregation
min, max	$([Int]) \to Maybe(Int)$	
avg	$([Int]) \to Maybe(Float)$	
or, and	$([Bool] \to Bool$	
nubFlat	$([\alpha :: Flat]) \to [\alpha :: Flat]$	distinct elements of list
takeWhile, dropWhile	$((\alpha) \to Bool, [\alpha]) \to [\alpha]$	keep list prefix/suffix by predicate
empty	$([\alpha]) \to Bool$	test for empty list

Figure 2.3.: Effect-afflicted functions that can be used in queries with the FERRY-based backend. Functions already available in Murrayfield are marked gray.

data and for large data sets this would overwhelm the database with a flood of queries, a so called *query avalanche* [16].

Now for a commercial break: The FERRY-based query backend described in this thesis aims to overcome the deficiencies described above, while keeping the positive aspects. It will

- provide a translation of LINKS expressions to database queries that more faithfully adheres to the ordered list semantics of LINKS, using an explicit encoding of element order on the relational level.
- allow a query to return not only lists of records of base values, but *arbitrary nested* combinations of lists, records, variant values and base values.
- provide a guarantee that queries whose types contains nesting are translated to a small number of SQL queries that is *statically* determined by the queries' type.
- make database functionality such as aggregate functions and grouping available. Aggregate functions such as min which are undefined on empty lists or tables are handled in a safe way by returning a $Maybe$ type.
- make a substantially larger subset of the LINKS standard library queryizable. Figure 2.3 lists functions that are implemented using recursion and incur the noqy effect, but are queryizable with the new query backend. Table A.1 (Appendix A) provides a complete list of functions that are either primitive or implemented recursively in the LINKS *prelude* that can be used in queries.

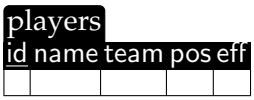


Figure 2.4.: Table schema for Query $Q3$.

- close some of the mentioned loopholes: provide support for the full set of LINKS regular expressions and provide relational implementations of comparison operators on non-atomic values.

At the same time, none of the desirable properties are dropped. Abstraction over queries is still possible in the same way. The newly queryizable subset of LINKS is higher-order, which makes it possible to handle e.g. lists of functions and higher-order functions such as takeWhile.

Not abandoning the *safety* aspect of query integration, the same mechanisms are used to ensure the queryizability of expressions. Effects analysis is used in the same way as before. Additionally, the (lightened) restriction on the type of queries is still enforced by the subkind mechanism, albeit not with the $Base$ subkind. This should make it very hard to write type-correct queries in LINKS that can't be translated, although we do not offer proves of correctness and completeness of the query translation.

We conclude this section with a motivating example query (Query $Q3$) that is intended to showcase some of the capabilities of the FERRY-based query backend. The query will be used as a running example during the course of the thesis. Given a table of basketball players, the query computes for each team an association list which contains for every position the efficiency of the most efficient player at this position. Figure 2.4 shows the table schema and Listing 2.6 lists Query $Q3$ formulated in LINKS.

```
fun eff(ps) { for (p <- ps) [p.eff] }
fun projTeam(p) { p.team }
fun projPos(p) { p.pos }
fun rosters(ps) { groupByFlat(projTeam, ps) }

query {
  for (r <- rosters(asList(players)) {
    var posn = groupByFlat(projPos, r.2);
    var maxps = for (pos <- posn) [(p.1, max(eff(pos.2)))];
    [(team = r.1, maxps = maxps)]
  }
}
```

Listing 2.6: Query $Q3$.

The query uses grouping (`groupByFlat`) and an aggregate function (`max`). The type of the query block reads

$$[(team : String, maxps : [(String, [|\ Just : Int \mid Nothing\ |])])]$$

and includes nested lists as well as a variant type. Note that the computation is performed completely on the database. *From the type* we can determine (see Sections 3.1.4 and 3.2.4) that the compilation of the query will result in a bundle of four SQL queries from whose tabular results the result value is assembled. The construction of the team rosters (lists of

players of a team) resembles the "teamRosters" query used as an example for the capabilities of the original LINKS query integration in [5]. However, whereas "teamRosters" only used a nested *intermediate* result, this time the overall result is nested.

3. The FERRY Framework

This chapter describes the core of the LINKS query backend: The compilation of a pure input language based on list comprehensions into bundles of query plans over a relational algebra.

We start with a language representing the hitherto feature set of FERRY (Section 3.1) and review the FERRY compilation technique. With this feature set, only a smaller subset of the LINKS language could be used in queries than is desirable. We therefore extend the input language with algebraic data types (Section 3.2) and with first-class functions (Section 3.3) and show how to handle these features in the loop-lifting framework. Section 3.4) is concerned with the compilation of additional language constructs: comparison operators applied to non-atomic values and string matching with regular expressions.

3.1. FERRY So Far

In this section we review the FERRY compilation technique. We briefly describe the data model for the relational representation of values (Section 3.1.2), the underlying compilation technique called *loop-lifting* (Section 3.1.5) and the table algebra which serves as a target language of the compilation (Section 3.1.6). Sections 3.1.7 and 3.1.8 are concerned with the actual compilation, describing the data structures used in the process and the inference rules defining the compilation.

3.1.1. Input Language

As a starting point, we use the simple input language SL whose grammar is defined in Figure 3.1. The language is meant to represent the features of the FERRY framework that have been described so far and to serve as an already useful basis for the LINKS query backend. It will be extended with additional features during the course of this chapter.

SL is treated as an untyped language because the compilation is not type-directed. However, as it is used to compile LINKS queries, every query must have passed the type checker in the frontend of the LINKS compiler and can therefore be considered type-correct. When speaking of types, we use the LINKS type notation sketched in Section 2.1. The language is pure, including no constructs that generate side effects. Also, explicit recursion is not possible. These properties allow the compilation to relational algebra in the first place.

The language features values of the usual atomic types ($Int, Float, String, Char, Bool$), the empty *unit* value $()$ as well as lists and records. Lists and records can be nested in arbitrary combinations. Access to database tables is provided by table handles. Table handles are semantically not different from lists of records of atomic values. Language constructs are provided to create, project, extend and restrict records. List constructs include constructors for empty and singleton lists. Non-trivial lists are created by the append operator, which takes a sequence of lists as arguments and flattens them into one list.

$$
\begin{array}{rrcl}
(\textit{terms}) & T & ::= & \mathsf{for}\ (x\ \leftarrow\ T)\ T \\
 & & | & \mathsf{for}\ (x\ \leftarrow\ T)\ \mathsf{orderby}\ (T)\ T \\
 & & | & \mathsf{let}\ x = T\ \mathsf{in}\ T \\
 & & | & \mathsf{if}\ T\ \mathsf{then}\ T\ \mathsf{else}\ T \\
 & & | & \mathsf{table}\ t : (\mathsf{col}_1, \ldots, \mathsf{col}_n) \\
 & & | & [T] \ \mid\ [] \\
 & & | & \mathsf{append}(\vec{T}) \\
 & & | & x \ \mid\ c \\
 & & | & T \circ T \\
 & & | & P(\vec{T}) \\
 & & | & T.l \ \mid\ (\overrightarrow{l = T}) \ \mid\ (\overrightarrow{l = T} \mid T) \\
 & & | & (T \setminus \vec{l}) \\
 & & | & () \\
 & & & \\
(\textit{constants}) & c & & \\
(\textit{variables}) & x & & \\
(\textit{operators}) & \circ & ::= & +, -, <, \tilde{}, \hat{}\hat{}, \ldots \\
(\textit{primitive functions}) & P & ::= & \mathsf{take} \mid \mathsf{drop} \mid \mathsf{zip} \mid \mathsf{unzip} \mid \mathsf{length} \\
 & & | & \mathsf{sum} \mid \mathsf{reverse} \mid \mathsf{or} \mid \mathsf{and} \mid \mathsf{nubFlat} \\
 & & | & \mathsf{concat}
\end{array}
$$

Figure 3.1.: Grammar of the input language SL.

The let $x = t$ in $\ldots$ construct binds the expression t to the variable x. The major construct to deal with lists and database tables are list or table comprehensions which are expressed with the for construct. It provides a way to iterate over a list or table and apply an expression to every element. The results of the individual iterations are again combined into a list. An orderby expression o can be added to impose an arbitrary order on the list that is generated by the comprehension. Note that the for comprehension is semantically equivalent to the concatMap function, not to the map function. As a consequence, the body of a comprehension must be list-typed. The results of all iterations are flattened into one list. This makes it possible to filter lists and tables with a combination of for and if expressions:

```
for (i <- [10, 20, 30, 40])
    if (i < 20) then [i] else []
```

A set of primitive operators is provided, including the usual arithmetic and comparison operators as well as string concatenation (^^) and regular expression matching (~). The language does not include function definitions or λ-abstractions. Only primitive functions can be applied. A set of primitive functions which are directly compiled to algebra code is included. Further primitive functions are introduced in Sections 3.2 and 3.3. First-class λ-abstractions are added in Section 3.3. Note that only functions which do not map directly to SQL:1999 equivalents and actually involve algebraic code generation are shown here. A complete list of supported primitive functions is included in Appendix A.

3.1.2. Relational Data Model

The basics of the relational data model used in the *loop-lifting* approach have been introduced in the context of XQuery [18] and an extension from flat to nested lists was given in [16]. In this section, we describe this data model briefly.

Atomic Values and Flat Lists

(a) $[10, 20, 30, 40]$ (b) 10 (c) $[]$

Figure 3.2.: Relational encoding of flat lists and atomic values.

The central built-in data structure of the LINKS language are lists. Lists are represented as tables in which each element of the list maps to one row. Depending on the type of the list elements, elements are represented by one or multiple item columns. However, while lists in LINKS are ordered, relational databases are inherently unordered. To support the ordering semantics of LINKS on a relational database, an additional pos column is added which represents the order of the list explicitly. As an example, Figure 3.2a shows the encoding of the LINKS list value [10, 20, 30, 40] where the pos column encodes the order and the item_1 column contains the actual data. Empty lists are represented by empty tables.

To keep the data model uniform, the representation of non-list values is also extended with a pos column. Figure 3.2b shows the representation of the atomic value 10. Note that a non-list value of type α and a singleton list of type $[\alpha]$ share the exact same representation. They can, however, be distinguished by their *implementation type* (Section 3.1.3).

Nesting of Lists

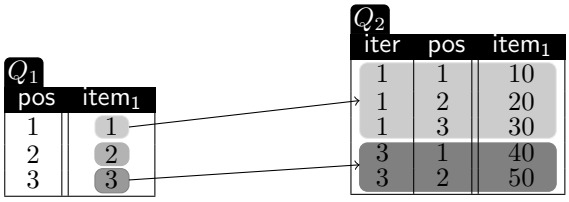


Figure 3.3.: Relational encoding of the nested list $[[10, 20, 30], [], [40, 50]]$.

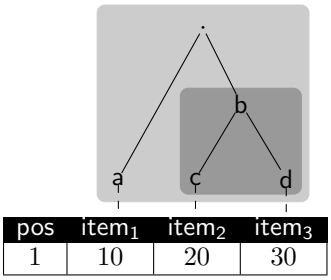
But how about lists that are nested in data structures, e.g. lists of lists? The scheme so far uses flat, first normal form tables and assumes that list elements are atomic values which can be represented by one column. If the list element itself is a list, this is not possible.

Instead, the inner lists are represented in the outer table via a column containing surrogate values. These values serve as foreign keys which reference rows in an *inner table*. Essentially, the outer table encodes the shape of the list and the inner table encodes the

content. The inner table combines the representation of all inner lists referenced by the outer list in one table. If the elements of the inner lists actually need to be accessed, the surrogate values are used for a foreign key join with the inner table to access the list elements. If one of the elements of the outer list is an empty list, the appropriate surrogate value in the outer table simply does not reference any rows in the inner table.

Figure 3.3 shows an example for the relational encoding of a nested list. Column item_1 of the outer table Q_1 contains surrogate values which are aligned with the values in the iter column of the inner table Q_2. The value 2 in Q_1 represents an empty list and therefore does not reference any rows in Q_2.

Records



pos	item_1	item_2	item_3
1	10	20	30

Figure 3.4.: Relational representation of the record $(a = 10, b = (c = 20, d = 30))$ (differently shaded areas depict the inner and outer records).

Next to lists, the second built-in data structure in LINKS are records. A record's fields can consist itself of records. Records can in general be nested to an arbitrary depth, forming a tree structure. However, the relational representation of (nested) records needs only to be concerned with the leaf fields of the record, which do not contain further records but actual data. Each leaf field is mapped to one column of the corresponding table. Therefore, on the relational level, no overhead is introduced to maintain the records structure. See Figure 3.4 for an example of a nested record and its relational representation. If a leaf field contains an atomic value, it is represented by an item column which contains just this value. If the field contains a value which can not be represented by one column (e.g. a list), the column contains surrogate values referencing an inner table just as in the representation of nested lists described above. During the compilation process, a separate data structure described in Section 3.1.7 is used to keep track of the record layout and the mapping of leaf fields to relational columns.

Database Tables

Relationally, an actual database table is represented in exactly the same way as a list of records. This uniform representation allows for a seamless integration of values stemming from the LINKS runtime and data from database tables. The obligatory pos column encodes some order on the table. The record's fields conform to the columns of the database table. Note that as we deal only with flat, first normal form relational databases, the item columns represent only atomic values in this case.

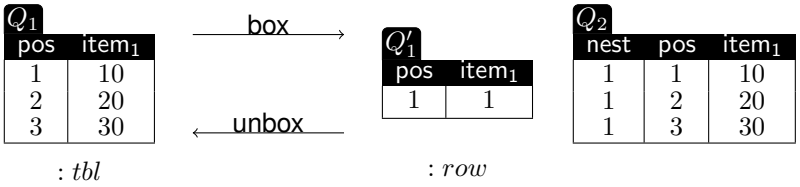


Figure 3.5.: Boxed and unboxed representations of the list $[10, 20, 30]$ and the corresponding implementation types.

3.1.3. Implementation Type

Section 3.1.2 explained the relational representation of lists: while lists of atomic values are represented by a single table, nested lists take the form of two tables which describe the shape of the outer list and the contents of the inner list respectively. These two representations make further work necessary. Consider a list of atomic values with type $[\alpha :: Base]$ and a list of lists of atomic values with type $[[\alpha :: Base]]$. They have different representations for the inner list, although the types are the same. This leads to a problem concerning the compilation rules: the rule handling the list constructor would need to know about its enclosing expression to choose the appropriate list representation. The compilation would not be compositional.

FERRY solves this problem by means of a static analysis phase on expressions of the input language which introduces calls to box and unbox functions. These functions do not change the semantics of the expressions but change the data representation of lists as necessary. A call to box on a list-typed expression splits the respective query plan into two plans, representing shape and content of the list. In contrast, a call to unbox on a boxed list typed expression merges the outer and inner plans. Only those expressions can be unboxed in which there is only a single inner list, because otherwise the first normal form would be violated. See Figure 3.5 for an example of the effects of box and unbox.

The analysis works by attributing an *implementation type* $\tau \in \{row, tbl\}$ to expressions. Data types are mapped to implementation types depending on how they are represented relationally: atomic values and records are implemented as single rows and have implementation type row. List-typed expressions can have one or the other type: unboxed lists are represented by multiple rows and have implementation type tbl, whereas a boxed list needs only a single row for the outer table and has implementation type row. Data constructors and functions require their argument sub-expressions to be of a certain implementation type. For instance, the list constructor [] requires its argument to be of implementation type row, so that every row can represent one list element. If the inferred implementation type does not match the required implementation type, calls to box and unbox need to be introduced to change the implementation type of the sub-expression. Such mismatches are handled by the function

$$\Downarrow_{\tau_1}^{\tau_2}(e) = \begin{cases} \mathsf{box}(e) & \tau_1 = row, \tau_2 = tbl \\ \mathsf{unbox}(e) & \tau_1 = tbl, \tau_2 = row \\ e \end{cases}$$

which uses box and unbox to force sub-expressions to the required implementation type.

Note that changes of the implementation type are only possible for list-typed expressions and therefore only list-typed expressions will be boxed or unboxed.

The set of inference rules that define this static analysis phase is shown in Figure 3.6. Rule 4 infers the implementation type of a let-bound expression and adds a binding to the typing environment Γ. In contrast, variables which are bound by for comprehensions always have the implementation type row (Rules 14, 15), because iteration variables are bound to list elements, which are always of implementation type row (Rule 11).

If not both branches are of implementation type tbl, the implementation type of the if branches is forced to row (Rule 10). Both branches have the same data type. But if this data type is a list type, the list representation created by the branches must not be the same. Therefore, the implementation type of the overall if expression can not just be determined by looking at the implementation type of a single branch.

This does however not lead to inefficient code. For an if branch expression b, consider the combination of the required implementation type imposed by the enclosing expression and the implementation type of the branch body, which is always forced to row by boxing:

$$\begin{array}{lcl} (row, row) & \rightarrow & b \\ (tbl, tbl) & \rightarrow & \mathsf{unbox}(\mathsf{box}(b)) \\ (row, tbl) & \rightarrow & \mathsf{box}(b) \\ (tbl, row) & \rightarrow & \mathsf{unbox}(b) \end{array}$$

Only if both the required type and the body type are tbl there is superfluous boxing. The generated unbox/box cascade can be easily removed when compiling expressions to algebra.

As an example for boxing, consider the expression $(a = 1, b = [10, 20])$. Records have implementation type row and therefore the sub-expressions of the fields a and b are required to be of implementation type row, so that all fields can be combined into a single row (Rule 6). However, the list constructor of field b leads to a table-based list representation with implementation type tbl and does not fit into a single row (Rule 11). This mismatch of implementation types is resolved by the $\boxed{\downarrow}$ function in Rule 6, which introduces a box call on the list-typed expression so that it's implementation type changes to row and meets the record constructor's requirement: $(a = 1, b = \mathsf{box}([10, 20]))$.

3.1.4. Avalanche Safety

From the relational data model and boxing phase, it should now be clear why the number of generated query plans is determined solely by the return type of the query and not by the size of lists or database tables (the so-called *avalanche safety* from Section 2). Every query generates at least one query plan. Every occurence of a list type constructor in the return type that is not the outermost type constructor adds a query plan. Elements contained in a list or record data structure need to be of type row. Therefore, every occurence of a list constructor (which per se leads to an unboxed representation) will cause a mismatch of implementation types which in turn causes a box call and a splitting of the query plan.

$$\frac{}{\{\ldots, x \to \tau, \ldots\} \vdash x = x : \tau} \quad (1) \qquad \frac{}{\Gamma \vdash c = c : row} \quad (2)$$

$$\frac{\Gamma \vdash r = r' : row}{\Gamma \vdash r.l = r'.l : row} \quad (3) \qquad \frac{\Gamma \vdash t = t' : \tau_t \quad \Gamma \cup \{x \to \tau_t\} \vdash e = e' : \tau_e}{\Gamma \vdash \mathsf{let}\ x = t\ \mathsf{in}\ e = \mathsf{let}\ x = t'\ \mathsf{in}\ e' : \tau_e} \quad (4)$$

$$\frac{\Gamma \vdash e_1 = e_1' : row \quad \Gamma \vdash e_2 = e_2' : row}{\Gamma \vdash e_1 \circ e_2 = e_1' \circ e_2' : row} \quad (5)$$

$$\frac{^{i=1,\ldots,n}|\Gamma \vdash e_i = e_i' : \tau_i}{\Gamma \vdash (l_1 = e_1, \ldots, l_n = e_n) = (l_1 = \boxplus^{\tau_1}_{row}(e_1'), \ldots, l_n = \boxplus^{\tau_n}_{row}(e_n')) : row} \quad (6)$$

$$\frac{^{i=1,\ldots,n}|\Gamma \vdash e_i = e_i' : \tau_i \quad r = r' : row}{\Gamma \vdash (l_1 = e_1, \ldots, l_n = e_n \mid r) = (l_1 = \boxplus^{\tau_1}_{row}(e_1'), \ldots, l_n = \boxplus^{\tau_n}_{row}(e_n') \mid r') : row} \quad (7)$$

$$\frac{\Gamma \vdash r = r' : row}{\Gamma \vdash (r \backslash \vec{l}) = (r \backslash \vec{l}) : row} \quad (8) \qquad \frac{}{\Gamma \vdash () = () : row} \quad (9)$$

$$\frac{\Gamma \vdash c = c' : \tau_c \quad \Gamma \vdash t = t' : \tau_t \quad \Gamma \vdash e = e' : \tau_e \quad \tau = \begin{cases} tbl & \tau_t = \tau_e = tbl \\ row & \end{cases}}{\Gamma \vdash \mathsf{if}\ c\ \mathsf{then}\ t\ \mathsf{else}\ e = \mathsf{if}\ c'\ \mathsf{then}\ \boxplus^{\tau_t}_{\tau}(t')\ \mathsf{else}\ \boxplus^{\tau_e}_{\tau}(e') : \tau} \quad (10)$$

$$\frac{\Gamma \vdash e = e' : \tau}{\Gamma \vdash [e] = [\boxplus^{\tau}_{row}(e')] : tbl} \quad (11) \qquad \frac{}{\Gamma \vdash [] = [] : tbl} \quad (12)$$

$$\frac{^{i=1,\ldots,n}|\Gamma \vdash l_i = l_i' : \tau_i}{\Gamma \vdash \mathsf{append}(l_1, \ldots, l_n) = \mathsf{append}(\boxplus^{\tau_1}_{tbl}(l_1'), \ldots, \boxplus^{\tau_n}_{tbl}(l_n')) : tbl} \quad (13)$$

$$\frac{\Gamma \vdash l = l' : \tau_l \quad \Gamma \cup \{x \to row\} \vdash e = e' : \tau_e}{\Gamma \vdash \mathsf{for}\ (x \leftarrow l)\ e = \mathsf{for}\ (x \leftarrow \boxplus^{\tau_l}_{tbl}(l'))\ \boxplus^{\tau_e}_{tbl}(e') : tbl} \quad (14)$$

$$\frac{\Gamma \vdash l = l' : \tau_l \quad \Gamma \cup \{x \to row\} \vdash e = e' : \tau_e \quad o = o' : \tau_o}{\begin{array}{rl} \Gamma \vdash & \mathsf{for}\ (x \leftarrow l)\ \mathsf{orderby}\ (o)\ e \\ = & \mathsf{for}\ (x \leftarrow \boxplus^{\tau_l}_{tbl}(l'))\ \mathsf{orderby}\ (o)\ \boxplus^{\tau_e}_{tbl}(e') : tbl \end{array}} \quad (15)$$

$$\frac{}{\Gamma \vdash \mathsf{table}\ t : (\mathsf{col}_1, \ldots, \mathsf{col}_n) = \mathsf{table}\ t : (\mathsf{col}_1, \ldots, \mathsf{col}_n) : tbl} \quad (16)$$

$$\frac{\Gamma \vdash l = l' : \tau \quad p \in \{\mathsf{take}, \mathsf{drop}, \mathsf{concat}\}}{\Gamma \vdash p(i, l) = p(i, \boxplus^{\tau}_{tbl}(l') : tbl)} \quad (17) \qquad \frac{\Gamma \vdash l = l' : \tau \quad p \in \{\mathsf{nubFlat}, \mathsf{reverse}\}}{\Gamma \vdash p(l) = p(\boxplus^{\tau}_{tbl}(l') : tbl)} \quad (18)$$

$$\frac{\Gamma \vdash l_1 = l_1' : \tau_1 \quad \Gamma \vdash l_2 = l_2' : \tau_2}{\Gamma \vdash \mathsf{zip}(l_1, l_2) = \mathsf{zip}(\boxplus^{\tau_1}_{tbl}(l_1'), \boxplus^{\tau_2}_{tbl}(l_2') : tbl)} \quad (19)$$

$$\frac{\Gamma \vdash l = l' : \tau \quad p \in \{\mathsf{unzip}, \mathsf{length}, \mathsf{sum}, \mathsf{and}, \mathsf{or}, \mathsf{empty}\}}{\Gamma \vdash p(l) = p(\boxplus^{\tau}_{tbl}(l') : row)} \quad (20)$$

Figure 3.6.: Inference rules that introduce box() and unbox() calls.

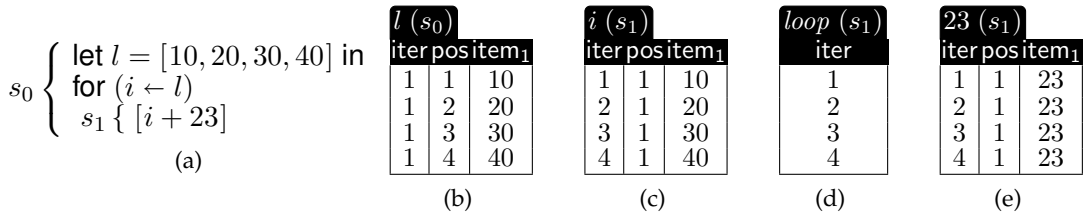


Figure 3.7.: Loop-lifting of iterations.

3.1.5. Loop Lifting

In this section, we briefly review the compilation technique named *loop-lifting*, which forms the conceptual core of the algebraic compilation. Loop-lifting has originally been developed to compile XQUERY expressions to relational algebra , but since been extended to handle general programming languages (see Section 6.1).

The central construct in LINKS to work with lists and database tables is the for comprehension: for $(i \leftarrow l)$ e evaluates the body expression e under different bindings of the iteration varaible i to all elements of the list or table l. As LINKS' type system guarantees all expressions in a query context to be free of side-effects (Section 2.1), the different evaluations of e are mutually independent.

The basic idea of loop lifting is to keep all values which are assigned to the iteration variable in one table and evaluate the body expression in a set-oriented manner on all those values at once. As relational database management systems are specialists in bulk operations applied to whole sets of rows, this allows for efficient evaluation of iteration constructs on a relational database.

Conceptually, every comprehension introduces a new *iteration scope* s_i, in which its body expression is evaluated. The iteration scope contains n iterations, one for each of the n elements that the iteration variable is bound to. As the top-level expression of the query is not enclosed in a comprehension, it is evaluated in a singleton iteration scope s_0, which contains one default iteration. The algebraic compilation produces code that *loop-lifts* the body expression according to the current iteration scope, so that it is evaluated for all iterations in the scope and thus for all bindings of the iteration variable.

To support the iteration concept, the relational representation of values (Section 3.1.2) is augmented with a column iter, which contains iteration identifiers. These identifiers establish to which iteration of the current scope each value belongs. Iteration identifiers are only kept in the outermost table. However, inner tables can share the same column layout including iter: Inner tables use the iter to store the surrogates that refer to the outer table (column nest in Figure 3.3). The *iteration context loop* is used in the compilation to encode the iteration scope. It consists just of the iter column and is used to lift the body expressions.

To illustrate the concepts, consider the example in Figure 3.7 which refers to to the compilation of the expression in Figure 3.7a. The list l is compiled in the enclosing singleton scope s_0, leading to the representation in Figure 3.7b. The for comprehension turns this table into a representation of the bindings for i (Figure 3.7c). Each of the rows represents one binding of the iteration variable i. It is to be read as "In iteration 1, i assumes the value

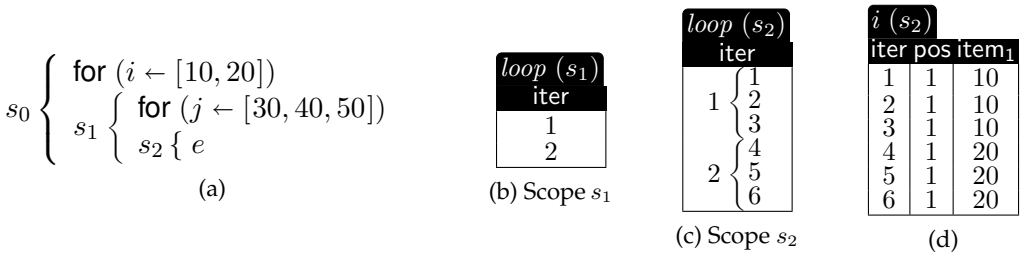
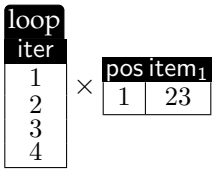


Figure 3.8.: Nested iteration scopes.

10, in iteration 2 . . . ". The body expression $[i + 23]$ is evaluated in the iteration scope s_1, represented by the *loop* table (Figure 3.7d). The expression 23 in the body of the comprehension is lifted to the iteration scope s_1. Conceptually, this is performed with a cartesian product as follows:



This leads to the loop-lifted representation of the constant 23 (3.7e). The iteration identifiers in the iter columns can now be used to relate the loop-lifted constant to the corresponding values of i and compute the column operator $\oplus$ on the operand columns.

Comprehensions can contain further nested comprehensions in their body. To see how nested comprehensions map to nested iteration scopes, consider the expression in Figure 3.8a. The inner comprehension is evaluated in the scope s_1 generated by the outer comprehension. This means that the inner comprehension is evaluated two times, once for each of the two iterations in s_1. The body e of the inner comprehension in scope s_2 is thus evaluated three times for each of the two iterations in s_1. As all of these iterations are independent and are performed over the same body expression e differing only in the values that the iteration variables i and j are bound to, all bindings for j are assembled into a single table. Consequently, e is evaluated in an innermost iteration scope s_2 with 2×3 iterations, meaning that e is lifted not only once but twice. Figures 3.8b and 3.8c depict how the scopes s_1 and s_2 are related and Figure 3.8d shows the representation of i lifted to iteration scope s_2.

3.1.6. Target Language: Relational Algebra

The intermediate language used as a result of the inference rules is a table algebra described e.g. in [16]. It is a dialect of the common relational algebra. The algebra was designed to reflect the capabilities of modern relational database management systems in an abstract way. From this intermediate language, a code generator creates code for actual database systems, i.e. SQL:1999 queries (see Section 4.4).

The algebra, which is listed in Table 3.1, consists of purist RISC-style operators which perform only single, well-defined tasks. For example, the selection operator σ does not evaluate a condition but relies on the presence of a boolean column which is used as the

Operator	Semantics
⛁_R	scan database-resident table R
a b c – – –	literal table with columns a, b, c
$\pi_{\mathsf{a}_1:\mathsf{b}_1,\ldots,\mathsf{a}_n:\mathsf{b}_n}$	project onto columns b_i, then rename b_i into a_i
σ_a	select rows with column $a = true$
$\_ \times \_$	Cartesian product
$\_ \overline{\bowtie}_{\mathsf{a},\mathsf{b}} \_$	equi join on columns a, b
$\_ \bowtie_{b_{1\cdot1}\theta_1 b_{2\cdot1}\wedge\cdots\wedge b_{1\cdot n}\theta_n b_{2\cdot n}} \_$	theta-join ($\theta \in \{=, <, >, <=, >=, \neq\}$)
δ	eliminate duplicate rows
$\_ \dot\cup \_$	disjoint set union
$\_ \setminus \_$	set difference
$@_{\mathsf{a}:v}$	attach constant value v in column a
$\circledcirc_{\mathsf{a}:\langle \mathsf{b}_1,\ldots,\mathsf{b}_n\rangle}$	attach result of n-ary operator application in column a ($\circledcirc \in \{+, -, \&\&, \neg, \ldots\}$)
$\#_{\mathsf{a}:\langle \mathsf{b}_1,\ldots,\mathsf{b}_n\rangle/\mathsf{c}}$	group rows by c, then attach row number (in $\mathsf{b}_1, \ldots, \mathsf{b}_n$ order) in column a
$\varrho_{\mathsf{a}:\langle \mathsf{b}_1,\ldots,\mathsf{b}_n\rangle/\mathsf{c}}$	group rows by column c, then attach row rank (in $\mathsf{b}_1, \ldots, \mathsf{b}_n$ order) in column a
$\text{GRP}_{\mathsf{a}_1:\circ_1(\mathsf{b}_1),\ldots,\mathsf{a}_n:\circ_n(\mathsf{b}_n)/\mathsf{b}_{grp}}$	group rows by column c and store result of aggregate function $\circ_i$ of column b_i in a_i ($\circ_i \in \{\text{COUNT}, \text{MAX}, \ldots\}$)

Table 3.1.: Operators of the table algebra used as an intermediate language.

criterion on which to select rows. The condition column for the selection has to be created separately via comparison and boolean operators. The set operators $\cup$ and $\setminus$ do not perform implicit duplicate elimination as it is common in the usual relational algebra. Instead, duplicates have to be eliminated explicitly with the distinct operator δ.

Some operators are not found in textbook relational algebrae and need to be explained. The attach operator $@_{a:c}$ attaches a new column a which contains the constant c in all rows. It could, in principle, be replaced by a cross product with a literal table. However, the specialized operator shortens the exposition and signals to subsequent optimization phases that the rows of the input table are only combined with a one-row one-column table, knowledge which aids the optimization.

The numbering operators $\#$ and ϱ are, among other things, used to generate pos and iter columns and thus to encode order on lists and tables and to generate new iteration contexts. They are also used to generate surrogate values as foreign keys for nested tables. The rownum operator $\#_{\mathsf{a}:\langle \mathsf{b}_1,\_,\mathsf{b}_n\rangle/\mathsf{c}}$ attaches a new column a which contains a numbering for the rows of the input table based on the ordering $\langle \mathsf{b}_1, \ldots, \mathsf{b}_n\rangle$. This numbering is created independently for all groups induced by the grouping column c. The consecutive numbers for each group start with 1.

Similarly, the row ranking operator $\varrho_{\mathsf{a}:\langle \mathsf{b}_1,\ldots,\mathsf{b}_n\rangle/\mathsf{c}}$ creates a new column a which contains the *rank* of the columns according to the order $\langle \mathsf{b}_1, \ldots, \mathsf{b}_n\rangle$ and grouped by the column c. All distinct ordering values according to the ordering are given consecutive numbers starting with 1.

The difference between row *numbering* and row *ranking* can be summed up as follows: If multiple rows in the same group have the same order according to the ordering $\langle \mathsf{b}_1, \ldots, \mathsf{b}_n\rangle$, they are assigned different *numbers* by the $\#$ operator, whereas the ϱ operator assigns the same *rank* to them.

3.1.7. Compilation Target

In general, the algebraic compilation phase emits query plans over the table algebra described in Section 3.1.6. A query plan abstractly represents a query against a relational database system which computes (parts of) the result. Query plans come in the form of directed acyclic graphs (DAGs) whose nodes are operators of the table algebra. The leaves of these DAGs can either be literal tables or references to database tables. Query plans make up the "code" that implements the semantics of the compiled SL program on the algebra level.

The query plans are in general DAGs and not just trees because of sharing: Sharing can occur when variables that are bound by a let expression, by a λ-abstraction or by a for comprehension are used multiple times. Additionally, the compilation function regularly uses the same sub-plan in multiple locations. By using a DAG, the duplication of these shared sub-plans is avoided. The external code generation phase (Section 4.4) considers this information about sharing and avoids repeated computation of the same results.

Due to the flat representation of nesting and variant values explained in Section 3.1.2, the algebraic compilation might actually emit multiple query plans. For example, the compilation of a nested list of type $[[Int]]$ emits two query plans: An outer plan that generates the outer list and represents inner lists via surrogate values, and an inner plan that generates the contents of all inner lists. Of course, every inner list might again have a type that

incurs nesting. In general, the algebraic compilation produces a tree of nested query plans which reflect the possibly nested structure of the result.

As an invariant, all plans that are produced by the compilation rules share a common column scheme $(\mathsf{iter}, \mathsf{pos}, \mathsf{item}_1, \ldots, \mathsf{item}_n)$, where iter contains the iteration identifiers of the current iteration context, pos contains the positions of list elements and the item_i columns contain the payload data. This invariant layout ensures the compositionality of the compilation rules.

All inference rules that define the compilation function $\Rightarrow$ from SL to algebraic code produce the same result: a row (q, cs, ts). q is the actual (outer) query plan. The remaining components are auxilliary data structures which are used to manage the column layout (cs) and the child query plans which compute inner lists (ts). The cs and ts components of the triple are described in detail in Sections 3.1.7 and 3.1.7.

Component cs

As described in Section 3.1.2, hierarchical record values are mapped onto flat column sequences. The generated table only contains the leaf fields of the record. As part of the compilation result row, the cs component is used to keep track of columns and their function, the actual record layout and the mapping of record fields to column names. cs is defined as a recursive type

$$\begin{array}{rll} \text{type } cs = & | & Col \text{ of } column \\ & | & Map \text{ of } [(field * cs)] \\ & | & Unit \text{ of } column \end{array}$$

A cs component of the form Col $\mathsf{item_i}$ indicates that the column $\mathsf{item_i}$ contains either atomic values or surrogate values representing nested lists. The constructor $Unit$ is used to mark columns which represent unit values. The constructor Map is used to keep track of the record layout if the table represents a record. It contains a list of pairs of field names and further cs components. These nested cs components might again be Map entries indicating a nested record or they can stand for leaf fields, i.e. $Unit$ or Col.

As an example, the cs component

$$Map\ [(a, Col\ 1), (b, Map\ [(c, Col\ 2), (d, Col\ 3)])]$$

describes the layout and column mapping of the nested record $(a = 10, b = (c = 20, d = 30))$, whose relational encoding was shown in Figure 3.4.

Component ts

The ts component stores inner plans that represent inner lists. It consists of a mapping from column names to inner plans. For every column that contains surrogate values, the corresponding inner plan is stored.

As an example, the ts component

$$\{\mathsf{item}_1 \rightarrow (q_1, cs_1, ts_1), \mathsf{item}_2 \rightarrow (q_2, cs_2, ts_2)\}$$

stores inner plans for two surrogate columns item_1 and item_2.

$\|l\|$	length of a list
$\mathsf{appendMappings}(cs, (l_i = cs_i))$	add a mapping to cs
$[cs]_<$	extract the sorted list of column names from cs
$\mathsf{shift}(cs, i)$	increase/decrease all column names stored in cs by i
$\mathsf{shiftTS}(ts, i)$	increase/decrease all column names used as keys in ts by i
$\mathsf{filterFields}(fields, cs)$	remove any field in the list $fields$ from cs
$\mathsf{keepOnly}(columns, ts)$	remove all entries for columns not in the list $columns$ from ts
$\mathsf{min}(cs)$	return the column name item_i from cs such that i is minimal

Table 3.2.: Helper functions to maintain the cs and ts components.

Helper Functions

The compilation process relies on some helper functions to maintain the cs, ts and vs components of the result. These functions are listed in table 3.2.

3.1.8. Compilation Rules

Notation

The compilation of SL expressions to table algebra plans is performed by the syntax driven function $\Mapsto$ in a fully compositional way. The compilation function is inductively defined by *inference rules* which describe the compilation of specific parts of the input language SL. An inference rule generally has the form

$$\frac{\begin{array}{c} premise_1 \\ \dots \\ premise_n \end{array}}{\Gamma; loop \vdash e \Mapsto (q, cs, ts)}$$

This judgement indicates that, given an environment Γ, an iteration context $loop$ and premises $1, \dots, n$, the SL expression e maps to the row (q, cs, ts) defined in Section 3.1.7. The environment Γ maps variables to (q, cs, ts) triples. When compiling an expression e, the environment Γ contains bindings for all variables which occur free in e. The iteration context $loop$ is a plan with a single column iter containing all iteration identifiers of the current iteration scope.

The compilation function descends top-down over the input expression tree, passing environments and iteration contexts down to the compilation of possible sub-expressions. The results for sub-expressions in the form of (q, cs, ts) triples are passed back up and merged in the current rule.

The compilation of the root expression of the input SL program which delivers the overall result starts with

$$\{\}; \boxed{\begin{array}{c}\mathsf{iter}\\ 1\end{array}} \vdash e \Mapsto (q, cs, ts)$$

As no bindings have been processed, the environment Γ is empty. The top-level expression of a query is evaluated exactly once. That is, e is equivalent to $\mathsf{for}\ (x \leftarrow [1])\ [e]$, where

x does not occur free in e. For this reason, the default iteration context contains exactly one iteration identifier.

3.1.9. Merging of Plans

The function $\overset{sq}{\Rightarrow}$ which is defined by inference rule SEQCONS is responsible for merging multiple plans from different branches of an expression into one plan.

$$\frac{\begin{array}{c} q_{1\cdot ext} \equiv @_{\mathsf{ord}:1}(q_1) \\ i{=}2,\ldots,n \;\Big|\; q_{i\cdot ext} \equiv \left(@_{\mathsf{ord}:i}\left(q_i\right)\right) \cup q_{(i-1)\cdot ext} \\ q_{nk} \equiv \#_{\mathsf{item}':\langle \mathsf{iter},\mathsf{ord},\mathsf{pos}\rangle}\left(q_{n\cdot ext}\right) \\ q \equiv \pi_{\mathsf{iter},\mathsf{pos}:\mathsf{pos}',\ \mathsf{keys}(ts_n)\,\cup\,\mathsf{keys}(vs_n):\mathsf{item}'}\left(q_{nk}\right) \\ ts_1,\ldots,ts_n \overset{ts}{\Rightarrow} (q_{ts}, cs_{ts}, ts_{ts}) \end{array}}{(q_1, cs_1, ts_1),\ldots,(q_n, cs_n, ts_n) \overset{sq}{\Rightarrow} (q, cs, ts)} \quad \text{(SEQCONS)}$$

The function $\overset{sq}{\Rightarrow}$ can only be used if the assumption is met that the iteration identifiers in the iter column of the plans $q_1, \ldots, q_n$ are pairwise disjoint, i.e. that an iteration value from one plan occurs in no other one. The assumption holds for rules which *fragment* iteration contexts, e.g. IFTHENELSE, CASE and APPLYCLOSURE. This restriction guarantees that positions in the pos column do not overlap for each iteration. Therefore, positions don't need to be recomputed. In contrast, the rule APPEND in Section 3.1.10 is used to merge plans with overlapping iterations.

The basic merging of the plans is done by recursively merging two plans with a disjoint union: For example, a disjoint union is employed to merge q_n and the result of the recursive merge of q_{n-1} to q_1. Before the unions, a column ord is attached with a distinct value for each original plan. This column encodes the concatenation order of the n original plans and is used during the merging of inner plans.

If one of the item columns contains surrogate values, those surrogate values will no longer be unique after the union. Therefore, new surrogate values are computed and stored temporarily in the column newkey. After the inner plans have been merged (see next paragraph), the new values in the newkey column are projected to all actual surrogate columns.

Having merged the outer plans into one plan q, we are still left with the individual $ts_1, \ldots, ts_n$ components. Those have to be merged into a single ts component by merging the plans represented by the entries of the components. For this purpose, the helper function $\overset{ts}{\Rightarrow}$ is employed.

To keep the presentation comprehensible, we only show a special case of the merging of ts components in Rule APPENDTS: In the rule shown it is assumed that only one of the item columns contains surrogate values and that all other columns contain normal data. It should be obvious how to extend this to the general case.

Given the result q_o of the merging of the outer plans, Rule APPENDTS merges the inner plans $q_1, \ldots, q_n$. This follows basically the same pattern as the merging of outer plans (SEQCONS). Each plan is equipped with an ord column which stores the concatenation

$$\frac{\begin{array}{c} q_{1 \cdot ext} \equiv @_{\mathsf{ord}:1}\,(q_1) \\ i{=}2,\ldots,n \,|\, q_{i \cdot ext} \equiv @_{\mathsf{ord}:i}(q_i) \uplus \left(q_{(i-1) \cdot ext}\right) \\ q_{nk} \equiv \#_{\mathsf{newkey}:\langle \mathsf{iter},\mathsf{ord},\mathsf{pos} \rangle}\,(q_{n \cdot ext}) \\ q_i \equiv \pi_{\begin{array}{l}\mathsf{iter}:\mathsf{newkey}',\\ \mathsf{pos},[cs_1]_<,\\ \mathsf{keys}(ts_1):\mathsf{newkey}\end{array}} \left(\left(\pi_{\begin{array}{l}\mathsf{ord}':\mathsf{ord},\\ \mathsf{newkey}':\mathsf{newkey},\\ \mathsf{oldkey}:\mathsf{item}_1\end{array}} (q_o) \right) \bowtie_{\begin{array}{l}\mathsf{iter}=\mathsf{oldkey}\\ \wedge \mathsf{ord}=\mathsf{ord}'\end{array}} q_{nk} \right) \\ q_i \vdash ts_1, \ldots, ts_n \overset{ts}{\Rightarrow} ts \end{array}}{q_o \vdash \{1 \to (q_1, cs_1, ts_1)\}, \ldots, \{1 \to (q_n, cs_n, ts_n)\} \overset{ts}{\Rightarrow} (q_i, cs_1, ts)} \quad \text{(APPENDTS)}$$

order of the n plans. The plans are then merged recursively with a pairwise disjoint union. Note that the plan order stored in the ord column is the same used in the outer plans ord column.

However, after bringing together the inner plans $q_1, \ldots, q_n$, the values in the iter column might overlap. Additionally, the surrogate values used in the outer plan to reference the iter column of the inner plans have been recomputed. As a consequence, we have to re-align the surrogate values used in the inner and outer plan. A join between the inner and outer plan just on columns item_1 (old surrogate values from q_o) and iter (old surrogate values from q_i) would not be sufficient, as those values are no longer unique after the merge. But as the foreign-key relationship between the outer and the inner plan is still stable for rows from corresponding original inner and outer tables, we can use the same order expressed in the ord columns to relate the corresponding rows. For this reason, ord is used as an additional join criterion. The freshly computed surrogate values in the newkey columns are then used in the iter column of the inner plan.

The just merged plans can once again contain nested plans in their ts components. These have to be merged recursively, which is done by another application of the $\overset{ts}{\Rightarrow}$ function. For the same reason, new surrogate values are computed for the inner plans of q_i itself. A projection makes these keys available in all surrogate columns of the inner plan.

Figure 3.9 shows the algebra code that is generated when appending plans which have nested plans. The merging of the outer plan is shown on the lefthand side, the merging of the inner plans on the righthand side. An intermediate plan with the freshly computed surrogate values in column newkey is related via an equijoin with the corresponding rows of the merged inner plan on the righthand side. For both plans, the fresh surrogate values are then used in all surrogate columns.

3.1.10. Data construction and Destruction

Atomic Values

To compile an atomic constant value in a *loop-lifted* way, the value has to be attached to all iterations of the current iteration context. Additionally, to establish the invariant layout of the resulting algebraic plan, every row has to be extended by a pos column which is always 1. Since we are dealing with atomic values here, there is no nesting to be considered and so the ts component is empty.

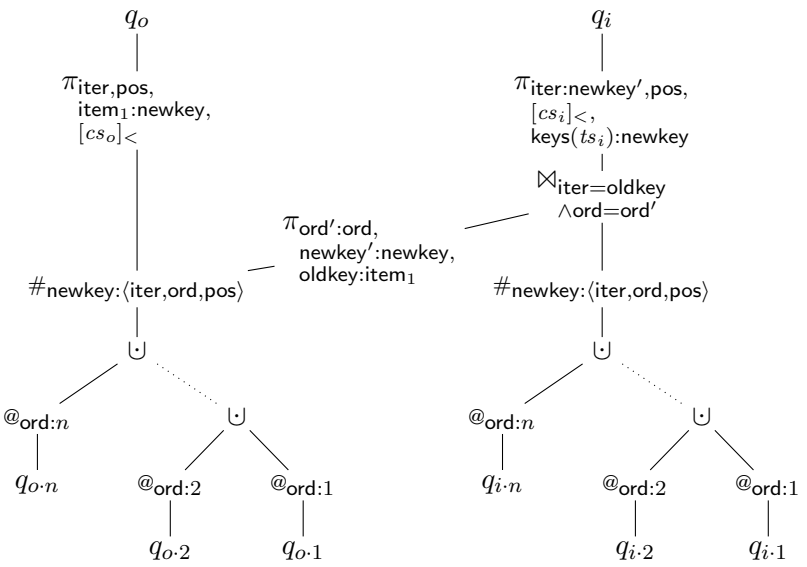


Figure 3.9.: Merging of outer and inner plans.

$$\frac{q \equiv @_{\text{item}_1:c}\,(@_{\text{pos:1}}\,(loop))}{\Gamma;\, loop \vdash c \Rightarrow (q, Col\ 1, \emptyset)} \quad \text{(ATOMICCONSTANT)}$$

$$\frac{q \equiv @_{\text{item}_1:1} @_{\text{pos:1}}\,(loop)}{\Gamma;\, loop \vdash () \Rightarrow (q, Unit\ 1, \emptyset)} \quad \text{(UNIT)}$$

The relational representation of the $()$ value is obtained by adding an item_1 column which contains a fake value. No operations are defined on the unit type, so it will never occur in any computation.

Records

$$\frac{\begin{array}{c}\Gamma;\, loop \vdash e_1 \Rightarrow (q, cs, ts) \\ cs' \equiv Map\ [l_1 = cs]\end{array}}{\Gamma;\, loop \vdash (l_1 = e_1) \Rightarrow (q, cs', ts)} \quad \text{(SINGLETONRECORD)}$$

Inference rule SINGLETONRECORD compiles record literals that consist only of a single field. The rule first compiles the field expression e_1 and uses the result as the overall result. The only difference is that the cs component is wrapped in a cs' component that records the mapping for the field l_1.

The inference rule RECORD compiles records with multiple fields by recursing over the fields. The rule starts by constructing a singleton record from the first field. This singleton record is then extended recursively with the remaining fields. To extend a record by one

$$\frac{\begin{array}{c}\Gamma; \mathit{loop} \vdash (l_1 = e_1) \Mapsto (q_{r_1}, cs_{r_1}, ts_{r_1}) \\ \left|\begin{array}{l} \Gamma; \mathit{loop} \vdash e_i \Mapsto (q_i, cs_i, ts_i, vs_i) \\ cs'_i \equiv \mathsf{shift}(cs_i, |cs_{r_{i-1}}|) \\ cs_{r_i} \equiv \mathsf{appendMappings}(cs_{r_{i-1}}, \mathit{Map}\ [l_i = cs'_i]) \\ ts_{r_i} \equiv ts_{r_{i-1}} \oplus \mathsf{incrKeys}(ts_i, |cs_{r_{i-1}}|) \\ q \equiv \pi_{\mathsf{iter},\mathsf{pos},\atop [cs_{r_i}]_<} \left(q_{r_{i-1}} \,\bar{\bar{\boxtimes}}_{\mathsf{iter},\mathsf{iter}'} \left(\pi_{\mathsf{iter}:\mathsf{iter}',\atop [cs'_i]_< : [cs_i]_<} (q_i) \right) \right) \end{array}\right. \quad i{=}2,\ldots,n \end{array}}{\Gamma; \mathit{loop} \vdash (l_1 = e_1, l_2 = e_2, \ldots, l_n = e_n) \Mapsto (q_{r_n}, cs_{r_n}, ts_{r_n})} \quad (\textsc{Record})$$

field ($l_i = e_i$), the expression e_i has to be compiled first. The column names used in q_i have to be shifted so that they do not overlap with the column names used in the already existing record. Also, the keys in the ts_i components have to be updated with the function incrKeys to account for the shifting. The ts component of the extended record is then constructed by appending the components of the original record with those of the extension field. Because of the shifting, the keys in the components do not overlap. Finally, to get the algebraic plan of the extended record, an equijoin on the iter columns aligns rows from the extended record with those of the extension field from the same iteration.

Note that the *extension* of a record

$$(l_{n+1} = e_{n+1}, \ldots, l_{n+k} = e_{n+k} | r)$$

can be done in exactly the same way as the construction of a record literal. The only difference is that the recursion does not start by constructing a singleton record from the first field but with the already existing record r that is to be extended.

$$\frac{\begin{array}{c}\Gamma; \mathit{loop} \vdash r \Mapsto (q_r, cs_r, ts_r) \\ cs = \mathsf{filterFields}([l_1, \ldots, l_n], cs_r) \\ ts = \mathsf{keepOnly}([cs]_<, ts_r) \\ q \equiv \pi_{\mathsf{iter},\mathsf{pos},[cs]_<} (q_r) \end{array}}{\Gamma; \mathit{loop} \vdash (r \setminus l_1, \ldots, l_n) \Mapsto (q, cs, ts)} \quad (\textsc{EraseRecord})$$

Erasing a set of fields from a record can be done by filtering all fields from the cs structure that are to be removed. This is done by the function filterFields. A projection with the remaining fields only keeps those columns for the fields which are still in the record. If any of the removed columns contains surrogate values referencing an inner table, the corresponding entries in the ts component are removed by the function keepOnly.

$$\frac{\begin{array}{c}\Gamma; \mathit{loop} \vdash r \Mapsto (q_r, cs_r, ts_r) \\ cs_l = cs_r[l] \quad cs = \mathsf{shift}(-\mathsf{min}([cs_r]_<), cs_r) \\ ts = \mathsf{shiftKeys}(-\mathsf{min}([cs_r]_<), ts_r) \\ q \equiv \pi_{\mathsf{iter},\mathsf{pos},[cs]_< : [cs_l]_<} (q_r) \end{array}}{\Gamma; \mathit{loop} \vdash r.l \Mapsto (q, cs, ts)} \quad (\textsc{Project})$$

Projection from a record is done by projecting only those item columns that belong to the record field that is to be projected. This list of columns is obtained by looking up the field in the cs component. To maintain the invariant that the payload columns start with item_1, column names are shifted appropriately if necessary.

Lists

$$\frac{}{\Gamma;\, loop \vdash [] \Rightarrow \left(\begin{array}{|c|c|c|}\hline \mathsf{iter} & \mathsf{pos} & \mathsf{item}_1 \\ \hline & & \\ \hline \end{array}, Col\ 1, \emptyset, \emptyset \right)} \quad \text{(EMPTYLIST)}$$

A literal empty list value is represented by a literal empty table. It might appear suspicious that the relational representation of $[]$ statically has one item column, although $[]$ is polymorphic and can be combined with values of a different relational structure (e.g. $(10, 20) :: []$). This can lead to incompatible column layouts of the child of binary relational operators like $\cup$, $\setminus$ etc. This is a drawback of the untyped nature of the input language SL, which makes it impossible to infer the correct column layout for the empty list. The problem can however easily be avoided by a separate *pruning* phase on the query plans following the algebraic compilation, which removes all occurences of literal empty tables. For example, the algebraic term

$$q_l \cup \begin{array}{|c|c|c|}\hline \mathsf{iter} & \mathsf{pos} & \mathsf{item}_1 \\ \hline & & \\ \hline \end{array}$$

is equivalent to the sub-plan q_l.

$$\frac{\Gamma;\, loop \vdash e \Rightarrow (q, cs, ts)}{\Gamma;\, loop \vdash [e] \Rightarrow (q, cs, ts)} \quad \text{(SINGLETONLIST)}$$

The relational representation of a singleton list is just the same as the representation of the value that is contained in the list. Therefore, the inference rule SINGLETONLIST just has to compile the expression e to get the representation of the singleton list itself.

Creation of Nesting

As described in Section 3.1.3, the boxing phase annotates the program with calls to the box and unbox functions to split or merge the plan according to the appropriate list representation. These functions are handled by the inference rules BOX and UNBOX.

$$\frac{\begin{array}{c}\Gamma;\, loop \vdash e \Rightarrow (q_e, cs_e, ts_e) \\ q \equiv @_{\mathsf{pos}:1}\left(\pi_{\mathsf{iter},\mathsf{item}_1:\mathsf{iter}}\left(loop\right)\right)\end{array}}{\Gamma;\, loop \vdash \mathsf{box}(e) \Rightarrow (q, Col\ 1, \{1 \Rightarrow (q_e, cs_e, ts_e)\}, \emptyset)} \quad \text{(BOX)}$$

The inference rule BOX creates the encoding of nested lists via inner tables and foreign-key relationships described in Section 3.1.2. Surrogate keys in the outer table are derived

from the current *loop* plan. For the inner table, the expression to be boxed is compiled in the current iteration context and stored in the ts component. Since the surrogate values in the item_1 column of the outer table and the iter values of the inner table both are created from the same *loop* plan, they are aligned.

$$\frac{\begin{array}{c}\Gamma; \mathit{loop} \vdash e \Rightarrow (q_e, cs_e, \{1 \Rightarrow (q_i, cs_i, ts_i)\}) \\ q \equiv \pi_{\mathsf{iter:iter'},\mathsf{pos},[cs_i]_<}\left(\left(\pi_{\mathsf{iter'}:\mathsf{iter},\mathsf{c}:\mathsf{item}_1}\left(q_e\right)\right) \mathbin{\overline{\overline{\bowtie}}}_{\mathsf{c}=\mathsf{iter}} q_i\right)\end{array}}{\Gamma; \mathit{loop} \vdash \mathsf{unbox}(e) \Rightarrow (q, cs_i, ts_i)} \quad \text{(UNBOX)}$$

If the content of an inner list actually needs to be accessed, it has to be unboxed from the inner table. To this end, the inference rule UNBOX first compiles the expression that is to be unboxed. As this expression is an inner list, the ts component contains a (q, cs, ts) triple representing this inner list. An equijoin between the surrogate values of the outer plan and the iter column of the inner plan lifts the rows of the inner plan that are referenced by the outer plan to the current iteration scope. At the same time, it removes all inner rows which are not referenced from the outside.

Appending Lists

$$\frac{\begin{array}{c}\Gamma; \mathit{loop} \vdash l_1 \Rightarrow (q_1, cs_1, ts_1) \\ q_{1 \cdot ext} \equiv @_{\mathsf{ord}:1}(q_1) \\ {\scriptstyle i=2,\ldots,n}\left|\begin{array}{l}\Gamma; \mathit{loop} \vdash l_i \Rightarrow (q_i, cs_i, ts_i, vs_i) \\ q_{i \cdot ext} \equiv \left(@_{\mathsf{ord}:i}\left(q_i\right)\right) \cup q_{(i-1) \cdot ext}\end{array}\right. \\ q_{nk} \equiv \#_{\mathsf{newkey}:\langle \mathsf{iter},\mathsf{ord},\mathsf{pos}\rangle}\left(\varrho_{\mathsf{pos'}:\langle \mathsf{ord},\mathsf{pos}\rangle}\left(q_{n \cdot ext}\right)\right) \\ q \equiv \pi_{\mathsf{iter},\mathsf{pos}:\mathsf{pos'},\mathsf{keys}(ts_n) \cup \mathsf{keys}(vs_n):\mathsf{item'}}\left(q_{nk}\right) \\ q_{nk} \vdash ts_1, \ldots, ts_n \overset{ts}{\Rrightarrow} ts\end{array}}{\Gamma; \mathit{loop} \vdash \mathsf{append}(l_1, \ldots, l_n) \Rightarrow (q, cs, ts)} \quad \text{(APPEND)}$$

So far, inference rules for empty and – possibly nested – singleton lists were presented. To actually construct non-trivial lists, the inference rule APPEND handles append expressions.

Appending lists works roughly in the same way as the merging of plans described in Section 3.1.9. The plans $q_1, \ldots, q_n$ representing the individual lists are equipped with an ord column to encode the order of the original plans and then merged recursively with a disjoint union. To append the inner plans in the ts components, new surrogate values are computed by the row numbering operator $\#$ in the column newkey. The surrogate values in newkey are used by the auxillary function $\overset{ts}{\Rrightarrow}$ to append the inner plans.

There are, however, two differences. First, whereas SEQCONS works on plans, APPEND has to compile the SL expressions $l_1, \ldots, l_n$ for the lists to be appended first. Second, SEQCONS assumes that the iteration identifiers used in the original plans are pairwise disjoint, so that the positions in the pos column do not overlap and don't need to be recomputed. APPEND uses the same *loop* plan to compile all expressions $l_1, \ldots, l_n$ so that all plans $q_1, \ldots, q_n$ contain the exact same iteration identifiers in the iter column. Therefore,

for each iterations, the positions of $q_1, \ldots, q_n$ overlap and positions need to be recomputed. Column pos orders the contents of the tables encoded by the original plans $q_1, \ldots, q_n$ and ord orders the original plans, so that pos and ord together impose the correct overall order on the merged plan. Row ranking (ϱ) over pos and ord is used to express this order in a single column pos′ which is then used instead of the original pos column.

Database Tables

$$\frac{\begin{array}{c} q \equiv loop \times \left(\varrho_{\mathsf{pos}:\langle c_1,\ldots,c_n\rangle}\left(\pi_{\mathsf{item}_1:\mathsf{col}_1,\ldots,\mathsf{item}_n:\mathsf{col}_n}\left(t\right)\right)\right) \\ cs \equiv Map\left[c_1 \rightarrow Col\ \mathsf{item_1}, \ldots, c_2 \rightarrow Col\ \mathsf{item_n}\right] \end{array}}{\Gamma; loop \vdash \mathsf{table}\ t : (\mathsf{col}_1, \ldots, \mathsf{col}_n) \Rightarrow (q, cs, \emptyset)} \quad \text{(TABLE)}$$

The actual purpose of a query block is to perform computations on data stemming from a database. Inference rule TABLE compiles a reference to a database table t with columns $c_1, \ldots, c_n$. To create the invariant layout of the resulting plan, a projection renames the columns of the table to the usual $\mathsf{item}_1, \ldots, \mathsf{item_n}$ schema. To support order semantics, a pos column is added to the table by the rowrank operator ϱ. The resulting plan has to be loop-lifted to the current iteration scope, which is achieved with a cross product between the table and the current iteration context $loop$. The resulting plan contains for each iteration all rows of t. As the table is represented as a list of records where the table columns are the record fields, the cs structure contains mappings for all the columns of the table. Since the columns of a database table can only contain atomic values, the ts component is empty.

3.1.11. Control Structures

Having described the inference rules which create the relational encoding of the available data types, we now turn to the relational handling of control structures: the list or table comprehension for and the conditional if.

Iteration

$$\frac{\begin{array}{c} \Gamma; loop \vdash l \Rightarrow (q_l, cs_l, ts_l)\ ① \\ q_l' \equiv \#_{\mathsf{inner}:\langle \mathsf{iter},\mathsf{pos}\rangle}\left(q_l\right) \quad q_v \equiv @_{\mathsf{pos}:1}\left(\pi_{\mathsf{iter}:\mathsf{inner},[cs_l]_<}\left(q_l'\right)\right)\ ② \\ map \equiv \pi_{\mathsf{outer}:\mathsf{iter},\mathsf{inner}}\left(q_l'\right) \quad loop' \equiv \pi_{\mathsf{iter}}\left(q_v\right) \\ \Gamma' \equiv \left\{\ldots, x \Rightarrow \left(\pi_{\mathsf{iter}:\mathsf{inner},\mathsf{pos},[cs_l]_<}\left(q_x \ \bar{\bar{\bowtie}}_{\mathsf{iter},\mathsf{outer}}\ map\right), cs_x, ts_x\right), \ldots\right\} \\ \Gamma_v \equiv \Gamma' \cup \left\{v \Rightarrow (q_v, cs_l, cs_l, ts_l)\right\} \\ \Gamma_v; loop' \vdash e \Rightarrow (q_e, cs_e, ts_e)\ ③ \\ q \equiv \pi_{\mathsf{iter}:\mathsf{outer},\mathsf{pos}:\mathsf{pos}',[cs_e]_<}\left(\varrho_{\mathsf{pos}':\langle \mathsf{iter},\mathsf{pos}\rangle}\left(q_e \ \bar{\bar{\bowtie}}_{\mathsf{iter},\mathsf{inner}}\ \left(\pi_{\mathsf{outer},\mathsf{inner}} map\right)\right)\right)\ ④ \end{array}}{\left\{\ldots, x \rightarrow (q_c, cs_x, ts_x), \ldots\right\}; loop \vdash \mathsf{for}\ (v \leftarrow l)\ e \Rightarrow (q, cs_e, ts_e)} \quad \text{(FOR)}$$

The central construct of the FERRY language is the iteration construct for, which is handled by the inference rule FOR. FOR follows the loop-lifting approach to handle iterations

as outlined in Section 3.1.5. A new iteration context is created from the list that is being iterated over by computing new iteration identifiers with the rownum (#) operator. A projection on these new iteration identifiers creates an iteration context $loop'$ representing the new iteration scope. The plan map provides a mapping between the original iteration scope (identifiers in column outer) and the newly created scope (identifiers in column inner). It will be used to map the iteration result back into the original iteration scope.

The new iteration identifiers are used in the iter column of list l. Additionally, a new pos column is created which contains the constant 1 for all iterations, denoting single elements for all iterations. The representation of the input list of length n is thus turned into a plan which represents n independent iterations, each iteration containing one element of the original list l.

Entries in the environment Γ still depend on the original iteration context $loop$. An equijoin between plans in Γ and the map plan lifts those plans into the new (inner) iteration scope. The thereby obtained lifted environment Γ' is then extended with a binding for the plan q_v, so that the variable v which occurs freely in the body of the iteration is available when compiling this body. The environment Γ_v thus contains a binding of the variable v to a plan which computes the values for v for all iterations.

Given the new iteration context $loop'$ and the lifted and extended environment Γ', the body expression of the iteration is compiled. For each iteration, the resulting plan q_e computes the result of evaluating the body expression e for the value of the iteration variable v in this iteration. These results have to be mapped back into the original iteration scope to form the result list of the comprehension. This mapping is achieved with an equijoin between the result plan q_e and the plan map, which aligns the corresponding iterations of the inner and outer iterations. So far, the pos column still consists of the constant 1. To obtain the correct representation of the overall list-typed result of the for expression, new positions are computed by the row ranking operator ϱ. This step has the effect of flattening the lists that result from the individual iterations into one result list according to the concatMap semantics.

We illustrate how the FOR rule works by showing the intermediate tables which occur when compiling the expression [1]

$$\text{let } y = 10 \text{ in for } (x \leftarrow \text{append}([10],[20],[30],[40]))\ [x + y + 20]$$

Figure 3.10 shows the query plan that results from the compilation of the for iteration. As the complete plan is too large to be included here, we omit the compilation of the list and the let-binding. Table q_l results from the compilation of the list and q_y is the plan from the environment that results from the let-binding. Tables on the left and right sides of the plan are intermediate results that correspond to the individual steps of Rule FOR: the map that relates iteration identifiers from the inner and outer iteration contexts, the environment entry q'_y that is lifted into the new iteration scope, the new iteration context $loop'$, the lifted list q_v, the result of the body of the comprehension and the overall result after it has been lifted back into the original iteration scope via an equijoin with map.

[1] For the sake of the example, we ignore the fact that y could be inlined into the body of the comprehension.

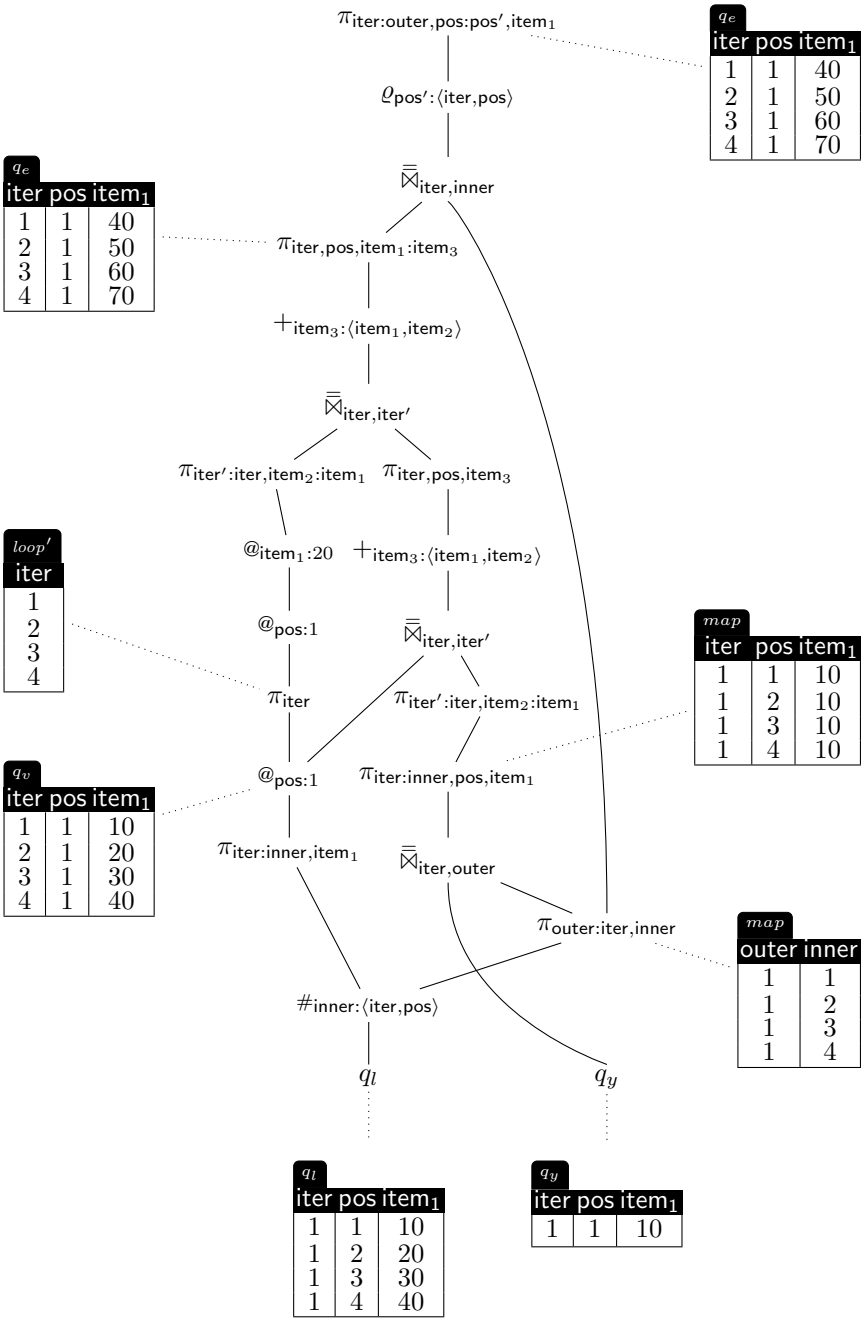


Figure 3.10.: Query plan resulting from list comprehension (Example in Section 3.1.11).

Explicitly Ordered Iteration

So far, the order expressed in pos columns was derived either from the order of the arguments of list construction (APPEND) or the ordering of database tables according to the values of their rows (TABLE). To impose an order based on an arbitrary ordering expression, an orderby clause can be added to a for loop. In a loop

$$\text{for } (v \leftarrow l) \text{ orderby } (o)\ e$$

the ordering expression o is evaluated for each binding of the iteration variable v in the same iteration scope as the body expression e. The results of the body expression are then ordered using the result of the ordering expressions as a criterion.

The ordering expression is compiled using the same environment Γ_v and the same iteration context $loop'$ as in the compilation of the body expression:

$$\Gamma_v; loop' \vdash o \Rightarrow (q_o, cs_o, \emptyset)$$

The resulting plan q_o is joined with the map plan, leading to an (outer, inner item$_{m+1}$, ..., item$_{m+n}$) schema (where m is the number of item columns of the result plan q_e):

$$map' \equiv \pi_{\text{item}_{m+1}:\text{item}_1,\ldots,\text{item}_{m+n}:\text{item}_n} \left(map \ \overline{\overline{\bowtie}}_{\text{inner,iter}}\ q_o \right)$$

The actual ordering can subsequently be integrated into the backmapping step of the original Rule FOR. As usual, the backmapping joins map with the result of the body expression on iter and inner, aligning the iteration identifiers from the body iteration scope with the corresponding iteration identifiers from the enclosing iteration scope in column outer. Additionally, this aligns the result rows from q_e with the corresponding results of the ordering expressions, whose column names have to be shifted appropriately to avoid overlaps. Finally, with an application of the row ranking operator ϱ, new positions are computed based primarily on the ordering columns and, secondarily, on the original pos column. If the result plan q_e has m item columns, the backmapping step now reads

$$q \equiv \pi_{\substack{\text{iter:outer,pos:pos}', \\ [cs_e]_<}} \left(\varrho_{\substack{\text{pos}':<\text{item}_1,\ldots,\text{item}_{,} \\ \text{iter,pos}>}} \left(q_e \ \overline{\overline{\bowtie}}_{\text{iter,inner}}\ map \right) \right)$$

This scheme assumes that the ordering expression o results in a plan in which each item column represents atomic values. This assumption holds indeed, because LINKS' type system basically restricts the results of the ordering restrictions to values of type $\alpha :: Flat$, i.e. atomic values or (nested) records (see Section 4.2.2).

If Conditions

Inference rule IFTHENELSE handles if expressions in a loop-lifted way. First, the condition expression c is compiled and yields a plan containing one boolean column item$_1$ which contains for every iteration the result of the condition. With a selection on true and false values in this column and a projection on the iter column, we split the original iteration context into two *sub-contexts*: The iteration context $loop_{then}$ contains all iterations for which the condition evaluates to true and the iteration context $loop_{else}$ contains all iterations for

$$\frac{\begin{array}{c}\Gamma; loop \vdash c \Rightarrow (q_c, Col\ 1, \emptyset) \\ loop_{then} \equiv \pi_{\mathsf{iter}}\left(\sigma_{\mathsf{item}_1}\left(q_c\right)\right) \\ loop_{else} \equiv \pi_{\mathsf{iter}}\left(\sigma_{\mathsf{item}_2}\left(\neg_{\mathsf{item}_2:\langle\mathsf{item}_1\rangle}\left(q_c\right)\right)\right) \\ \Gamma_{then} \equiv \left\{\ldots, x \rightarrow \left(loop_{then} \;\overline{\overline{\bowtie}}_{\mathsf{iter},\mathsf{iter}'}\; \left(\pi_{\mathsf{iter}':\mathsf{iter}}\left(q_x\right)\right), cs_x, ts_x\right), \ldots\right\} \\ \Gamma_{else} \equiv \left\{\ldots, x \rightarrow \left(loop_{else} \;\overline{\overline{\bowtie}}_{\mathsf{iter},\mathsf{iter}'}\; \left(\pi_{\mathsf{iter}':\mathsf{iter}}\left(q_x\right)\right), cs_x, ts_x\right), \ldots\right\} \\ \Gamma_{then}; loop_{then} \vdash t \Rightarrow (q_t, cs_t, ts_t) \\ \Gamma_{else}; loop_{else} \vdash e \Rightarrow (q_e, cs_e, ts_e) \\ (q_t, cs_t, ts_t), (q_e, cs_e, ts_e) \overset{seqcons}{\Rightarrow} (q, cs, ts)\end{array}}{\Gamma; \{\ldots, x \rightarrow (q_c, cs_x, ts_x), \ldots\} \vdash \mathsf{if} c\ \mathsf{then}\ t\ \mathsf{else}\ e \Rightarrow (q, cs, ts)} \quad \text{(IFTHENELSE)}$$

which the condition evaluates to false. Consequently, the then branch of the conditional has to be evaluated in the iteration scope represented by $loop_{then}$ and the else branch in the iteration scope represented by $loop_{else}$. However, the entries in the environment Γ still contain all iterations from the original context. An equijoin between the environment entries and the new iteration contexts $loop_{then}$ and $loop_{else}$ leads to new environments Γ_{then} and Γ_{else} which are restricted to the iteration sub-contexts of the two branches.

The results of both branches must be merged to form the overall result of the conditional expression. Since the two iteration sub-contexts are disjoint, the result plans from both branches could in principle be merged with a simple disjoint union $\uplus$. However, the item columns of the results might contain surrogate values which reference inner tables in the ts and vs columns. Care must be taken so that these surrogate values do not overlap if the branch results are merged. Therefore, after the outer plans are merged, new unique surrogate values are computed and the corresponding iter values in the inner plans are updated accordingly. This is taken care off by the helper inference rule SEQCONS (Section 3.1.9).

let-Bindings

$$\frac{\begin{array}{c}\Gamma; loop \vdash t \Rightarrow (q_t, cs_t, ts_t) \\ \Gamma \cup \{x \rightarrow (q_t, cs_t, ts_t)\}; loop \vdash e \Rightarrow (q_e, cs_e, ts_e)\end{array}}{\Gamma; loop \vdash \mathsf{let}\ x = t\ \mathsf{in}\ e \Rightarrow (q_e, cs_e, ts_e)} \quad \text{(LET)}$$

Rule LET first compiles the expression t and adds the result to the environment Γ. In the extended environment, the tail expression e is compiled.

3.1.12. Operators

Binary operators $\circ$ on atomic values (e.g. $+, -, <, >, \ldots$) in SL expressions are mapped to their database equivalents $\circledcirc$, which take two columns of a table as input and add a third column containing the result. In essence, these column operations correspond to vector operations.

To apply such an operation to two arguments, the arguments e_1 and e_2 have to be compiled first, producing two plans q_1 and q_2. The tables represented by q_1 and q_2 are then

$$\frac{\begin{array}{c}\Gamma; loop \vdash e_1 \Rightarrow (q_{e_1}, cs_{e_1}, \emptyset) \\ \Gamma; loop \vdash e_2 \Rightarrow (q_{e_2}, cs_{e_2}, \emptyset) \\ q \equiv \pi_{\substack{\mathsf{iter},\mathsf{pos},\\ \mathsf{item}_1:\mathsf{res}}} \left(\circledcirc \mathsf{res} : \langle \mathsf{item}_1, \mathsf{item}_2 \rangle \left(q_{e_1} \;\bar{\bar{\bowtie}}_{\mathsf{iter}=\mathsf{iter}'} \left(\pi_{\substack{\mathsf{iter}':\mathsf{iter},\\ \mathsf{item}_2:\mathsf{item}_1}} \right) \right) \right)\end{array}}{\Gamma; loop \vdash e_1 \circ e_2 \Rightarrow (q, cs_e, \emptyset)} \quad \text{(BINOP)}$$

joined into a single table by an equijoin which brings together rows stemming from the same iterations. The operator itself is then applied to the combined table and produces a column containing the result of the operation, which is then used as the item_1 column of the result table.

We omit the trivial rule UNOP, which handles just the unary negation function not via the relational negation operator $\neg$.

3.1.13. Primitive Functions

We describe only a subset of the inference rules for primitive functions here to demonstrate how functions are implemented algebraically. Rules for the remaining functions and, or, take and drop can be found in Appendix A.

Aggregate Functions

$$\frac{\begin{array}{c}p \in \{\mathsf{length}, \mathsf{sum}\} \quad \circ \in \{\text{COUNT}, \text{SUM}\} \quad \Gamma; loop \vdash l \Rightarrow (q_l, cs_l, ts_l) \\ q \equiv @_{\mathsf{pos}:1} \left(\left(\text{GRP}_{\mathsf{item}_1 : \circ(\mathsf{item}_1)/\mathsf{iter}} (q_l) \right) \cup \left(@_{\mathsf{item}_1:0} \left(loop \setminus \pi_{\mathsf{iter}} (q_l) \right) \right) \right)\end{array}}{\Gamma; loop \vdash p(l) \Rightarrow (q, Col\ 1, \emptyset)} \quad \text{(AGGR)}$$

The rule AGGR handles the not-failing aggregate functions length and sum by mapping them to their relational counterparts, the operators COUNT and SUM. The rule needs to consider the special case that some iterations might contain empty lists. Recall that empty lists are represented by empty iterations (i.e. iterations for which there is no row). Those empty iterations are obtained by computing the set difference between the plans *loop* and q_l, representing the iteration context and the input list respectively. For the empty iterations, the static result 0 is attached, whereas for the non-empty iterations the aggregate function is computed.

The empty iterations are obtainable in this way because the iteration context *loop* is always complete, i.e., it contains all iterations, even the empty ones. If there is only a single iteration containing the empty list, the iteration context *loop* will nevertheless contain the default iteration 1. The only way to obtain a relation with multiple iterations of which some are empty is to iterate over a list of lists, where some of the inner lists are empty. In this case, the iteration context *loop* was derived from the newly created iteration identifiers, which are computed based on the outer plan. The table computed by this outer plan contains an entry for every list element, even if this list element is an empty list. Therefore, the iteration context *loop* used in the body of this loop contains one entry for every element of the outer list, be it empty or not.

$$\frac{\Gamma; loop \vdash l \Rightarrow (q_l, cs_l, ts_l) \quad q' \equiv \pi_{\mathsf{iter},\mathsf{pos}:\mathsf{pos}',[cs_l]_<}\left(\varrho_{\mathsf{pos}':\langle pos:desc\rangle}\,(q_l)\right)}{\Gamma; loop \vdash \mathsf{reverse}(l) \Rightarrow (q', cs_l, ts_l)} \quad \text{(REVERSE)}$$

Because the relational data model includes explicit positions for list elements in the pos column, order-aware functions like the reverse function can be implemented quite straight forwardly: To reverse a list, only a new pos column has to be computed which reverses the values of the original pos column. This can be done with an application of the row ranking operator ϱ which takes the original ascending pos column as a descending ranking criterion.

Positional Merging and Splitting

$$\frac{\begin{array}{c}\Gamma; loop \vdash l_1 \Rightarrow (q_1, cs_1, ts_1) \\ \Gamma; loop \vdash l_2 \Rightarrow (q_2, cs_2, ts_2) \\ cs_2' \equiv \mathsf{shift}(cs_2, |cs_1|) \quad ts_2' \equiv \mathsf{shiftTS}(ts_2, |cs_1|) \\ cs \equiv Map\,[(1, cs_1), (2, cs_2')] \quad ts \equiv ts_1 + ts_2' \\ q \equiv \pi_{\mathsf{iter},\mathsf{pos},[cs]_<}\left(q_1 \underset{\substack{\mathsf{iter}=\mathsf{iter}' \\ \wedge \mathsf{pos}=\mathsf{pos}'}}{\bowtie} \left(\pi_{iter':\mathsf{iter},\mathsf{pos}':\mathsf{pos},\,(q_2)}\right)\right)\end{array}}{\Gamma; loop \vdash \mathsf{zip}(l_1, l_2) \Rightarrow (q, cs, ts)} \quad \text{(ZIP)}$$

Rule ZIP positionally aligns two lists l_1 and l_2 and produces a list of pairs. Algebraically, this can be achieved in a rather elegant way with a single thetajoin which aligns list elements with the same position. We just have to observe that there might be multiple iterations involved, so the pos columns might not be distinct. Therefore, the iter columns have to be part of the join criterion. The semantic of the LINKS zip function is to discard excess elements if one of the lists is longer than the other one. This is done implicitly by the thetajoin, because for elements of one list which have positions greater than the largest position of the other list, no join partner will be found and they will be discarded. Additionally, the column names used for the second list must be shifted to avoid overlaps and the ts_2 component must be updated accordingly. To store the row structure of the result, an appropriate cs component is created.

Even less is necessary to unzip a list of pairs, i.e. to create a pair of lists which contain the first and second elements respectively. As these lists are stored in a record, they are represented by nested plans. Creating the inner plans consists just of removing the columns which belong to the other pair component with a projection. The outer list is created from scratch from the iteration context *loop*: The iter column is used to encode the surrogate values in the columns item_1 and item_2 and the position is constantly 1. This works because the outer plan just represents rows, not lists. Therefore, there is only one row per iteration. The rest of the rule is employed with maintaining the auxilliary data structures: the original ts component is restricted to the columns that are included in the plans q_1 and q_2, creating ts_1

$$\frac{\begin{array}{c}\Gamma; loop \vdash l \Rightarrow (q_z, Map\ [(1, cs_1), (2, cs_2)], ts_z) \\ cs_2' \equiv \mathsf{shift}(cs_2, -|cs_1|) \\ q \equiv \pi_{\substack{\mathsf{iter,pos,} \\ \mathsf{item_1{:}iter,} \\ \mathsf{item_2{:}iter}}} (@_{\mathsf{pos}:1}\ (loop)) \\ q_1 \equiv \pi_{\mathsf{iter,pos},[cs_1]_<}(q_z) \quad q_2 \equiv \pi_{\mathsf{iter,pos},[cs_2']_<[cs_2]_<}(q_z) \\ ts_1 \equiv \mathsf{keepOnly}([cs_1]_<, ts_z) \quad ts_2 \equiv \mathsf{shiftTS}(\mathsf{keepOnly}([cs_2]_<, ts_z), -|cs_1|) \\ cs \equiv Map\ [(1, (Col\ \mathsf{item_1}), (2, (Col\ \mathsf{item_2})] \\ ts \equiv \{\mathsf{item_1} \rightarrow (q_1, cs_1, ts_1), \mathsf{item_2} \rightarrow (q_2, cs_2', ts_2)\}\end{array}}{\Gamma; loop \vdash \mathsf{unzip}(l) \Rightarrow (q, cs, ts)}\ (\textsc{Unzip})$$

and ts_2 respectively. The shifting of column names again establishes the invariant scheme for column names by ensuring that item column names begin with item_1.

Distinct

$$\frac{\begin{array}{c}\Gamma; loop \vdash l \Rightarrow (q_l, cs_l, \emptyset) \\ q_r \equiv \varrho_{\mathsf{group}:\langle \mathsf{iter}, [cs_l]_< \rangle}\ (q_l) \\ q_m \equiv \pi_{\mathsf{group'}:\mathsf{group},\mathsf{pos'}}\ \left(\mathrm{GRP}_{\mathsf{pos'}:\mathrm{MIN}(\mathsf{pos})/\mathsf{group}}(q_r)\right) \\ q \equiv \pi_{\mathsf{iter,pos},[cs_l]_<}\ (q_r \bowtie_{\mathsf{pos}=\mathsf{pos'} \wedge \mathsf{group}=\mathsf{group'}}\ q_m)\end{array}}{\Gamma; loop \vdash \mathsf{nubBase}(l) \Rightarrow (q, cs_l, \emptyset)}\ (\textsc{NubFlat})$$

Because of a type restriction (Section 4.2.2), nubFlat can only be applied to lists in which the element values are represented by a single row, i.e. do not contain surrogate values. This ensures that values can be compared efficiently by looking at a single table.

Row ranking over the item columns generates a column group which contains the same group value for all rows that have identical item fields. The inclusion of the iter column in the ranking criterion keeps rows which are equal but stem from different iterations apart. The MIN aggregate operator computes the minimal position at which every distinct combination of item values occurs.

The information about the first occurence of each group has to be combined with the actual payload data again, which is done with a join between q_r and q_m. As the join criteria include the pos and pos′ columns, rows which do not occur at the first position for their combination of item values are discarded by the join. Therefore, for every combination of item values, only the first occurence remains.

3.2. Variant Values and Pattern Matching

3.2.1. Variants

In this section, we extend the set of datatypes supported in query compilation by variant types, also known as *sum-of-product* types and *algebraic datatypes*. A variant type $[|\ \mathrm{C}_1(\tau_1)\ |\ \mathrm{C}_2(\tau_2)\ |\ r\ |]$ consists of multiple constructors or *variant tags* $C_1, \ldots, C_n$, each of which

$$
\begin{array}{rrcl}
(\textit{terms}) & T & ::= & \ldots \\
 & & | & \mathsf{C}(T) \\
 & & | & \mathsf{case}\ T\ \mathsf{of}\ \overrightarrow{(\mathsf{C}(x)) \rightarrow T} \\
 & & & \\
(\textit{constructors}) & C & & \\
 & & & \\
(\textit{primitive functions}) & P & ::= & \ldots \\
 & & | & \mathsf{max} \mid \mathsf{min} \mid \mathsf{avg}
\end{array}
$$

Figure 3.11.: Extension of the language SL with algebraic data types (AL).

tags values of different types $\tau_1, \ldots, \tau_n$. By wrapping values of different types into tags, a variant type allows to use values of different types in the same place or type. For instance, consider the list $[\mathsf{A}(15), \mathsf{B}(23.0, 16.0)]$ of type $[|\ \mathrm{A} : Int \mid \mathrm{B} : (Float, Float) \mid r\ |]$.

To support variant values in query compilation, the grammar of the input language SL is extended as shown in Figure 3.11, yielding the langugage AL. A variant value is constructed by applying a variant tag C to a value. The only way to deal with variant values, i.e. to destruct them, is a case expression. case dispatches over the variant tag and chooses the proper case branch. The tagged value is bound to the case variable x and the case branch is evaluated. Due to the availability of variant values, the primitive aggregate functions max, min and avg can be handled which use variant values to handle errors.

3.2.2. Data Model

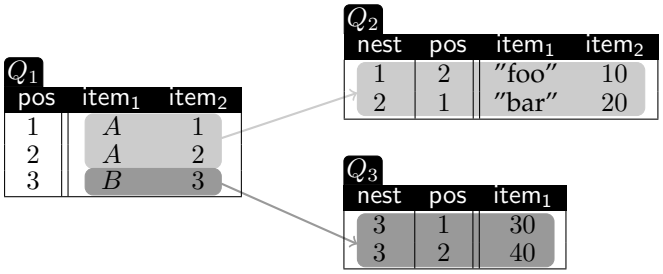


Figure 3.12.: Relational encoding of $[\mathsf{A}(\text{”}foo\text{”}, 10), \mathsf{A}(\text{”}bar\text{”}, 20), \mathsf{B}([30, 40])]$.

To handle polymorphic variants in query blocks and be able to return them as the result of a query, we need a relational encoding of variant values that fits into the loop-lifting framework. A variant type generally consists of multiple constructors. Each constructor tags values of a type that might be different from those of the other constructors. This means that the tagged values of the individual constructors have a different relational representation. The data model therefore has to consider two aspects: It must be possible to distinguish between the constructors and the representation has to combine values which have different relational representations.

An encoding using only a single table can not achieve this. Encoding a list of variant values into a single table might require different numbers of item columns for the tagged

$$\frac{\Gamma \vdash e = e' : \tau}{\Gamma \vdash C(e) = C(\boxdot\!\downarrow^{\tau}_{row}(e')) : row} \quad (21)$$

$$\frac{\begin{array}{c}\Gamma \vdash v = v' : \tau \\ {\scriptstyle i=1,\ldots,n} \,|\, \Gamma \cup \{x_i \to row\} \vdash e_i = e'_i : \tau_i \qquad \tau = \begin{cases} tbl & \tau_i = tbl \quad \forall i = 1,\ldots,n \\ row & \end{cases}\end{array}}{\begin{array}{ll}\Gamma \vdash & \mathsf{case}\ v\ \mathsf{of}\ (C_1(x_1) \to\ e_1), \ldots, (C_n(x_n) \to e_n) \\ = & \mathsf{case}\ v\ \mathsf{of}\ (C_1(x_1) \to\ \boxdot\!\downarrow^{\tau_1}_{\tau}(e'_1)), \ldots, (C_n(x_n) \to \boxdot\!\downarrow^{\tau_n}_{\tau}(e'_n)) : \tau\end{array}} \quad (22)$$

$$\frac{\Gamma \vdash l = l' : \tau \quad p \in \mathsf{max}, \mathsf{min}, \mathsf{avg}}{\Gamma \vdash p(l) = pl(\boxdot\!\downarrow^{\tau}_{tbl}(l)) : row} \quad (23)$$

Figure 3.13.: Additional rules for implementation type inference (Section 3.1.3) to handle variant values and case expressions.

types. Every item column could contain a combination of different atomic types and surrogate values (i.e. polymorphically typed columns). Additionally, the type of tagged values for one constructor might require inner tables, while those of another constructor might not. Therefore, an approach which stores complete variant values in one table would be very difficult to maintain and inefficient, if possible at all.

Instead, an approach was chosen that is quite similar to the encoding of nested lists via inner tables and foreign-key relationships. The encoding of variant values consists of two item columns. One column stores the tag of the value to be able to distinguish values with different tags. The other one contains surrogate values that refer to inner tables storing the tagged value. However, to avoid having to pack values of different types into one table, the encoding produces one inner table per tag. Every inner table stores the tagged values of one constructor. This means that the set of surrogate keys of the outer table of the encoding is split among multiple inner tables. Since the tagged values of one tag always have the same type, the types of the values in the inner tables are consistent. The plans encoding the inner tables are stored similarly to inner plans for nested lists in an auxiliary data structure (see Section 3.2.5 for details).

Figure 3.12 shows an example for the relational encoding of a list of variant values, in which the different tags contain values of different types and structure. Query $Q1$ contains the outer bla and queries $Q2$ and $Q3$ contain the representation of the tagged values for the tags A and B respectively.

3.2.3. Boxing of Variant Constructs

For the boxing phase (3.1.3), additional rules are needed that handle the new language constructs added in this section. The rules that handle variant values, case expressions and the added primitive functions are shown in Figure 3.13. Variant values always have implementation type row, as their tagged values are boxed into inner tables. But Rule 21 forces the implementation type of the tagged value to row, too. The reason is that the implementation type of the tagged values must be *uniform*. Consider the expression[2]

$$[\mathsf{C}([1,2]), \mathsf{C}(\mathsf{select}(2, [\mathsf{box}([3,4]), \mathsf{box}([5,6])]))]$$

[2]Frontend LINKS notation instead of SL notation is used for the list constructor for brevity.

The tagged list of the first constructor application is a literal list which is not boxed. The tagged list of the second constructor application is a selection from a nested list and therefore is boxed. When iterating over this list of variant values with a case expression, it would not be clear which implementation type the branch variable has. This would lead to the boxing phase failing on the body of case expressions. By forcing the tagged value to be of type row, tagged values have a uniform representation and the branch variables can always be set to row.

Rule 22 forces the implementation type of all branch bodies to row if not all branches are of type tbl. This happens for the same reason as for if expressions in Rule 10 (3.1.3).

3.2.4. Avalanche Safety

The augmentation of the input language with variant values does not violate the avalanche safety property (Section 3.1.4). However, the number of queries is no longer determined solely by the number of list type constructors in the return type of the query. Values that are tagged by a variant tag are boxed into inner tables, too. Therefore, every occurence of a variant type constructor in the return type leads to an additional query plan which – when evaluated – computes the tagged values of this constructor.

3.2.5. Auxiliary Data Structures and Functions

To support variant values, changes to the auxilliary data structures defined in Section 3.1.7 are necessary:

First, an additional Tag constructor is added to the cs datatype, so that the type now reads

$$\begin{aligned} \text{type } cs = \quad & | \quad Col \text{ of } column \\ & | \quad Unit \text{ of } column \\ & | \quad Tag \text{ of } (column * column) \\ & | \quad Map \text{ of } [(field * cs)] \end{aligned}$$

The Tag constructor is used to keep track of columns that represent variant values. If the plan contains one or multiple column pairs representing variant values, the cs component contains Tag $\mathsf{item_i}, \mathsf{item_j}$, entries, meaning that the column $\mathsf{item_i}$ contains the tag and column $\mathsf{item_j}$ contains the surrogate values referencing the tagged values in an inner table.

The compilation rules no longer produce rows (q, cs, ts), but add an additional component vs so that every rule produces a quadruple (q, cs, ts, vs). Basically, the vs component fullfils the same role as ts, storing inner plans which here contain the tagged values of variant values. However, columns can't serve as unique keys for the inner tables, because the surrogate values are split among multiple inner tables. Therefore, the keys for a variant inner plan consist of the tag and the corresponding surrogate column.

As an example, the vs component

$$\{(A, \mathsf{item_2}) \rightarrow (q_A, cs_A, ts_A, vs_A), (B, \mathsf{item_2}) \rightarrow (q_B, cs_B, ts_B, vs_B)\}$$

stores inner plan for the tags A and B. Surrogate values referencing these inner plans are stored in the same column $\mathsf{item_2}$.

3.2.6. Merging of Relations

When merging plans from multiple branches, not only the inner plans in ts components need to be merged, but also those in vs components. Therefore, a function $\overset{vs}{\Rightarrow}$ is needed that merges vs components similarly to $\overset{ts}{\Rightarrow}$. We omit the inference rule for the function $\overset{vs}{\Rightarrow}$ here, as it works in the same way as the function $\overset{ts}{\Rightarrow}$. The only difference is that for one surrogate column in the outer plan, multiple inner plans for the individual tags have to be appended.

3.2.7. Compilation Rules for Variant Values

Variant Construction

$$\frac{\begin{array}{c}\Gamma; loop \vdash v \Rightarrow (q_v, cs_v, ts_v, vs_v, fs_v) \\ q \equiv @_{\mathsf{item}_2:C}\,(@_{\mathsf{pos}:1}\,(\pi_{\mathsf{iter},\mathsf{item}_1:iter}\ loop)) \\ vs \equiv \{(1, C) \rightarrow (q_v, cs_v, ts_v, vs_v, fs_v)\}\end{array}}{\Gamma; loop \vdash C\ v \Rightarrow (q, Tag\ (,1,2), \emptyset, vs)} \quad \text{(VARIANT)}$$

Rule VARIANT generates the relational encoding of a variant value literal. The current iteration context *loop* is used to encode the surrogate keys for the outer table. Additionally, the tag of the constant is attached. The expression for the tagged value is compiled and the result is stored as an inner table in the vs column. Since the surrogate keys of the outer table and the iter values of the inner table both stem from the same *loop* plan, they are aligned.

3.2.8. Pattern Matching on Variant Values

A case expression

$$\mathsf{case}\ e\ \mathsf{of}\ \mathsf{C}_1(x_1) \Rightarrow\ c_1, \ldots, \mathsf{C}_n(x_n) \Rightarrow\ c_n,\ c_{default}$$

dispatches according to the tag C_i of the variant value e that serves as input to the case expression. The body c_i of the appropriate case is then evaluated in the current environment enriched with a binding of the tagged value of e to x_i.

In a loop-lifted manner, the case statement can not only be applied to a single variant value but it might be applied to all members of a loop-lifted sequence of variants at once. In order to achieve this, case expressions are handled in a similar way to the compilation of if expressions (Section 3.1.11): The iteration context is split into multiple sub-contexts based on the values in a column, which are then handled in the respective branches. The IFTHENELSE rule splits into two sub-contexts based on the boolean column resulting from the if condition.

But since we don't have only two branches in a case expression, we split into n sub-contexts containing the iterations for each of the n cases (if the optional *default* case is present, we obtain an additional sub-context containing all iterations which did not match one of the explicit cases). The sub-context corresponding to a specific case is created by

selecting all iterations in which the variant tag matches the tag of the case. Additionally, the tagged values of the variant values need to be unboxed from the inner tables for the respective tag. The body of each case is then compiled in the corresponding sub-context and in the current environment with the additional binding of the branch variable to the unboxed tagged value.

To sum up, the inference rule for case expressions has to perform the following steps:

- For each case
  1. select all rows for which the tag value in the tag column matches the tag of the case ①
  2. derive a new iteration sub-context *loop* from this selection ②
  3. unbox the tagged values from the inner plan for the current tag ③
  4. filter all iterations from the current environment which are not part of the current sub-context ④
  5. bind the unboxed tagged values to the lifted environment ⑤
  6. compile the case body in the iteration sub-context and the enhanced environment ⑥
- If a *default* case is present: use the remaining input rows which were not attributed to a specific case to derive an iteration sub-context for the default case; compile the body expression of the default case in this sub-context ⑦
- The results of the individual branches have to be brought together to form the overall result of the case expression. ⑧

Figure 3.14 shows the overall scheme of the inference rule CASE.

Selection of Tags

Figure 3.14 shows how the iteration context is split according to the tags. For each case, the case's tag is attached and compared to the tag column of the input. Those rows for which the tags match are selected for this case. The remaining rows which did not match are used as input for the next case. If there is a default case present, the rows which remain at the end constitute the input for the default case.

This stacked scheme is only necessary if there actually is a default case present. If there is none, the input to the individual branches can be selected solely from the complete original input.

One question that might arise is what happens if the input contains variant values with tags for which no case is included when there is no default case. The answer to this question is that this can not happen due to restrictions enforced by the type system. The type system enforces the type of the input to a switch statement to match the type of the patterns. This means that for every tag which occurs in the type of the input, a case must be present which handles values with this tag. Cases for tags which appear in the input type may only be left out if there is a default case. This default case handles all remaining tags that were not handled by specific cases. Thus, we have the guarantee that in a type-correct program, every tag that may occur in the input of a switch statement is handled by a specific case or by a default case.

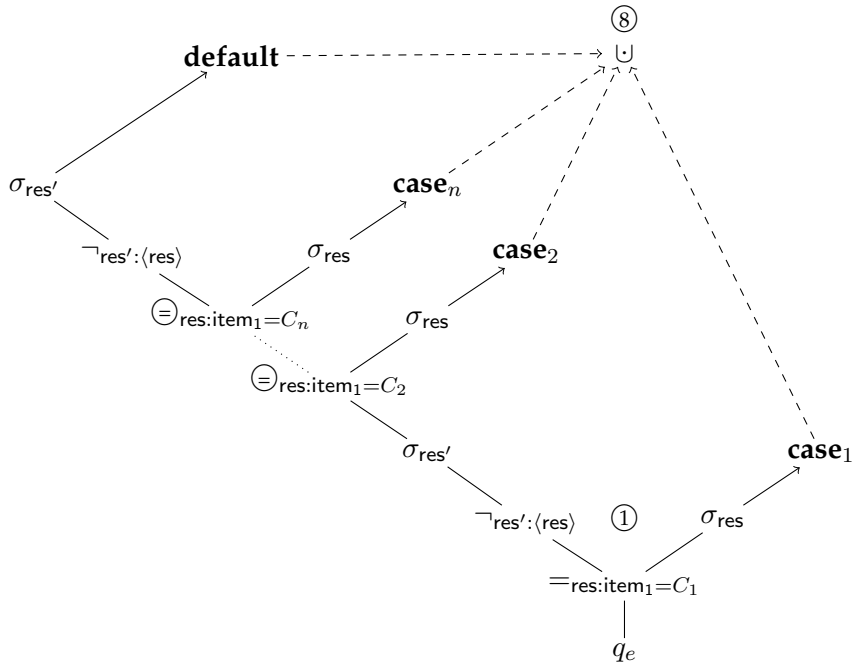


Figure 3.14.: Selection of rows for the individual case branches (attachment of C_i values and projections omitted).

Compilation of Individual Cases

Figure 3.15 shows the scheme for the compilation of a single case.

After the rows matching the current tag have been selected ①, a projection of the iter column generates a new iteration context $loop'$ ②.

An equijoin between the (restricted) outer table and the inner table for the current tag unboxes the tagged values of the variant values of the input and lifts them into the current (restricted) iteration context ③. Those tagged values are then bound to the variable of the case ⑤. For each binding in the environment, an equijoin between the restricted input plan and the environment entry removes those iterations from the environment which are not part of the current iteration sub-context ④. Those two steps generate the new environment Γ'.

Having acquired an iteration context $loop'$ as well as an environment Γ', these two serve as input to the normal compilation function $\Rightarrow$, which compiles the body expression of the case ⑥.

Sequence Construction

Bringing the results from the separate case branches together works exactly the same way as for if expressions. Since the iteration contexts from the branches are pairwise disjoint, a simple union would be sufficient. However, if there are any inner tables, new surrogate keys need to be computed and propagated to the appended inner tables. This is done by the helper inference rule SEQCONS ⑧.

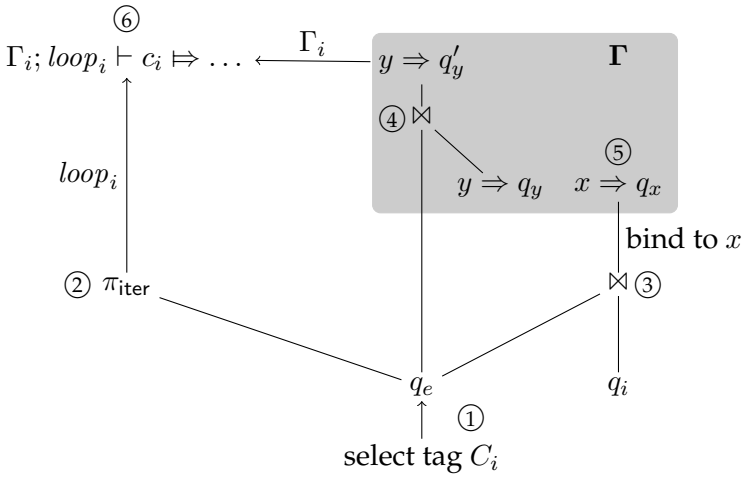


Figure 3.15.: Compilation scheme for individual case expressions. q_e is the query plan generating the outer table, from which all rows have been removed that do not match the tag C_i. q_i is the query plan for the inner table of the tag C_i.

$$
\begin{array}{l}
\Gamma; loop \vdash e \Mapsto (q_e, cs_e, ts_e, vs_e) \\
q_{0 \cdot not} \equiv q_e \\
\left.\begin{array}{l}
\left.\begin{array}{l}
q_{i \cdot m} \equiv =_{\mathsf{r}:\langle \mathsf{item}_1, \mathsf{item}' \rangle} \left(@_{\mathsf{item}':C_i} \left(q_{(i-1) \cdot not} \right) \right) \\
q_{i \cdot s} \equiv \pi_{\mathsf{iter},\mathsf{pos},\mathsf{item}_2} \left(\sigma_{\mathsf{r}} \left(q_{i \cdot m} \right) \right) \\
q_{i \cdot not} \equiv \pi_{\mathsf{iter},\mathsf{pos},\mathsf{item}_1,\mathsf{item}_2} \left(\sigma_{\mathsf{r}'} \left(\neg_{\mathsf{r}':\langle \mathsf{r} \rangle} (q_{i \cdot m}) \right) \right)
\end{array}\right\} ① \\
loop_i \equiv \pi_{\mathsf{iter}}(q_{i \cdot s}) \quad ② \\
vs_e \equiv \{\ldots, (C_i, \mathsf{item}_2) \Rightarrow (q_{C_i}, cs_{C_i}, ts_{C_i}, vs_{C_i})\} \\
q_{i \cdot unboxed} \equiv \pi_{\mathsf{iter}:\mathsf{iter}',[cs_{C_i}]_<} \left(\left(\pi_{\mathsf{iter}':\mathsf{iter},\mathsf{c}:\mathsf{item}_1} (q_{i \cdot s}) \right) \bar{\bar{\bowtie}}_{\mathsf{c}=\mathsf{iter}} \; q_{C_i} \right) \quad ③ \\
q_{z \cdot i} \equiv \pi_{\mathsf{iter},\mathsf{pos},[cs_z]_<} \left(q_z \; \bar{\bar{\bowtie}}_{\mathsf{iter},\mathsf{iter}'} \; \left(\pi_{\mathsf{iter}':\mathsf{iter}} (q_{q_{i \cdot unboxed}}) \right) \right) \quad ④ \\
\Gamma_i \equiv \{\ldots, z \Rightarrow (q_{z \cdot i}, cs_z, ts_z, vs_z), \ldots\} \cup \{x_i \Rightarrow (q_{i \cdot unboxed}, cs_{C_i}, ts_{C_i}, vs_{C_i})\} \quad ⑤ \\
\Gamma_i; loop_i \vdash c_i \Mapsto (q_{i \cdot r}, cs_{i \cdot r}, ts_{i \cdot r}, vs_{i \cdot r}) \quad ⑥
\end{array}\right\} i=1,\ldots,n \\
\left.\begin{array}{l}
loop_{default} \equiv \pi_{\mathsf{iter}} \left(q_{n \cdot not} \right) \\
q_{z \cdot default} \equiv \pi_{\mathsf{iter},\mathsf{pos},[cs_z]_<} \left(q_z \; \bar{\bar{\bowtie}}_{\mathsf{iter}:\mathsf{iter}'} \; \left(\pi_{\mathsf{iter}':\mathsf{iter}} (loop_{default}) \right) \right) \\
\Gamma_{default} \equiv \{\ldots, z \Rightarrow (q_{z \cdot default}, cs_z, ts_z, vs_z), \ldots\} \\
\Gamma_{default}; loop_{default} \vdash c_{default} \Mapsto (q_{default}, cs_{default}, ts_{default}, vs_{default})
\end{array}\right\} ⑦ \\
(q_{1 \cdot r}, cs_{1 \cdot r}, ts_{1 \cdot r}, vs_{1 \cdot r}), \ldots, (q_{1 \cdot r}, cs_{1 \cdot r}, ts_{1 \cdot r}, vs_{1 \cdot r}) \overset{seqcons}{\Mapsto} (q_r, cs_r, ts_r, vs_r) ⑧
\end{array}
$$

$$
\frac{\begin{array}{lll}
 & C_1(x_1) & \Rightarrow c_1, \\
 & \ldots, & \\
 & C_n(x_n) & \Rightarrow c_n \\
 & & \Rightarrow c_{default}
\end{array}}{\{\ldots, z \Rightarrow (q_z, cs_z, ts_z, vs_z), \ldots\}; loop \vdash \mathsf{case}\ e\ \mathsf{of} \Mapsto (q_r, cs_r, ts_r, vs_r)}
$$

(CASE)

Rule CASE shows the complete inference rule for case expressions.

3.2.9. Changes to Compilation Rules

To handle variant values, several changes to compilation rules from Section 3.1 are necessary. We describe these changes only briefly because they mostly consist of straight forward maintenance of auxiliary data structures.

- All rules that change the layout of plans by shifting names of item columns must ensure that the column names used as partial keys in the vs structure are shifted likewise. This affects rules RECORD, PROJECT, ZIP and UNZIP.
- Rules ZIP and RECORD, which combine multiple plans into a record structure must append the respective vs components.
- Rule APPEND must merge the vs components of the individual lists with the $\overset{vs}{\Mapsto}$ function.
- Rules ERASERECORD and PROJECT must remove all vs entries for which the corresponding columns have been removed.
- Rule UNZIP has to split the vs component in the same way as the ts component

3.2.10. Primitive Functions

$$\frac{\begin{array}{c} p \in \{\mathsf{max}, \mathsf{min}, \mathsf{avg}\} \quad \circ \in \{\text{MAX}, \text{MIN}, \text{AVG}\} \\ \Gamma; loop \vdash l \Mapsto (q_l, cs_l, ts_l, vs_l) \\ loop_e \equiv loop \setminus (\pi_{\mathsf{iter}}\,(q_l)) \quad q_{in} \equiv @_{\mathsf{pos}:1}\,(@_{\mathsf{item}_1:1}\,(loop_e)) \\ q_{ij} \equiv @_{\mathsf{pos}:1}\left(\text{GRP}_{\mathsf{item}_1:\circ\,(\mathsf{item}_1)/\mathsf{iter}}\,(q_l)\right) \quad q_{oj} \equiv @_{\mathsf{item}_1:Just}\,(\pi_{\mathsf{iter},\mathsf{item}_2:\mathsf{iter}}\,(q_{ij})) \\ q_{on} \equiv @_{\mathsf{item}_1:Nothing}\,(\pi_{\mathsf{iter},\mathsf{item}_2:\mathsf{iter}}\,(loop_e)) \\ cs_{ij} \equiv Col\ \mathsf{item}_1 \quad cs_o \equiv Tag\ \mathsf{item}_1, \mathsf{item}_2 \\ q_o \equiv @_{\mathsf{pos}:1}\,(q_{oj} \cup q_{on}) \\ vs_o \equiv \{(1, Just) \to (q_{ij}, cs_{ij}, \emptyset, \emptyset), (1, Nothing) \to (q_{in}, Unit1, \emptyset, \emptyset)\} \end{array}}{\Gamma; loop \vdash p(l) \Mapsto (q_o, cs_o, \emptyset, vs_o)}\ (\text{AGGRERROR})$$

Role AGGRERROR demonstrates the usefulness of variant values available in query contexts. The result of most aggregate functions available in usual database systems (e.g. MAX, MIN, AVG) is not defined when applied to empty tables or empty lists. When queried via an SQL interface, database systems just return an empty result table in this case. However, if an aggregate function's type is for example $[Int] \to Int$, there is no well-typed way to signal that an error occured, if there is no exception mechanism.

Instead, the functions max, min and avg use an approach that is common in functional programming: the occurence or absence of an error is signalled by returning a value of the $Maybe$ type:

$$\mathsf{avg} :: ([Int]) \to Maybe(Float)$$

The result is either Just(x), encapsulating the result x if possible, or a value Nothing which signals the error condition "empty list". The error can then be handled by dispatching on the return value via pattern matching.

$$
\begin{array}{rrcl}
(\textit{terms}) & T & ::= & \ldots \\
 & & \mid & \lambda\vec{x}.T \\
 & & \mid & T\ \vec{T} \\
(\textit{primitive functions}) & P & ::= & \ldots \\
 & & \mid & \mathsf{takeWhile} \mid \mathsf{dropWhile} \mid \mathsf{groupByFlat} \\
 & & \mid & \mathsf{concatMap} \mid \mathsf{map} \mid \mathsf{sortByFlat}
\end{array}
$$

Figure 3.16.: Extension of the input language AL with first-class functions (FL).

To handle errors in this way, rule AGGRERROR splits the iterations of the input into two sets: those which contain non-empty lists (i.e. those for which rows are present) and those which contain empty lists. For the non-empty iterations, the aggregate is computed and stored in an inner plan representing the tagged values of the tag $Just$. For the empty iterations, the tagged value is then of $unit$ type, which is stored in the inner plan for the $Nothing$ tag. The outer plans q_{oj} and q_{on} store the tags $Just$ and $Nothing$ for both cases and reference the tagged values with surrogate keys in column item_2. q_{oj} and q_{on} then need to be merged with a disjoint union to acquire the end result. The surrogate values in item_2 are derived from the disjoint iter columns and are therefore disjoint themselves.

3.3. First-Class Functions

As the last language feature, we add anonymous functions or λ-abstractions to the input language, resulting in the language FL (Figure 3.16). Anonymous functions can be treated in almost the same way as values of all other types (i.e. atomic values, lists, records and variant values). That is, they

- can be passed as arguments to functions
- can be returned by other functions
- can be stored in data structures (lists, records, variants)

The only restriction needed on the handling of functions is that functions must not occur in the return type of the query or FL expression. This restriction must be made because it is not possible to encode a function itself relationally, only its application.

As a consequence, not only primitive functions can be applied, but all arrow-typed expressions. Comprehensions are replaced by the semantically equivalent concatMap function. Where comprehensions have a fixed body expression, the higher order concatMap function takes an arbitrary functional expression as its functional argument. The ordering aspect that was previously handled by a for ... orderby expression is moved to a separate sortByFlat function, which also takes an arbitrary functional expression as its ordering function argument. Some additional primitive higher-order functions are supported, too.

3.3.1. Data Model

The approach used to handle functions as first-class values algebraically has been sketched by Rittinger [26]. We describe this approach here in more detail.

To be able to treat anonymous functions as first-class values, a relational representation of functions is needed. However, a function abstraction itself can not be represented relationally, as the relational data model and relational algebra contain no concept of abstraction and application. This means that a query can not return values containing functions and the return type of a query needs to be restricted accordingly. However, this in turn means that every function which occurs in a query block is either discarded or applied. We can, therefore, represent functions with an indirection: surrogate values which identify specific functions are used to represent functions relationally. These surrogate values are used to track the flow of functions through the expressions in which they occur. When the compilation process encounters a function application, the surrogate values are used to relate to the function that is actually applied.

Which function is actually applied might depend on data that is not known at compile time. Additionally, the compilation happens in a *loop-lifted* way, so that instead of single functions a loop-lifted sequence of functions might be applied at once. For those two reasons, at an application site the body of every function that might be applied at this site has to be included. The surrogate values are then used to map to the correct function body.

Another aspect has to be considered: a function might have free variables that are bound in the lexical environment of the function, forming a *closure*. We have to make sure that those free variables can still be accessed when the function is applied in another scope. The usual approach to implement closures is *closure conversion*, which converts a function into a closed term. This is achieved by representing a function by a data structure which contains the function itself (or a code pointer) and an environment containing the function's lexical environment, i.e. a binding for every free variable of the function. The compilation technique described here essentially performs closure conversion.

Putting these two aspects (representation and closures) together, we can now describe more specifically how closures are implemented. The relational representation of a closure consists just of a single item column which contains surrogate values. Surrogate values are created from iteration identifiers and updated with the rownum operator #, in the exact same way as surrogate values for nested lists or variants.

The compilation of a λ-abstraction adds yet another component fs to the quadruple (q, cs, ts, vs). fs maps surrogate columns to lists of *closure rows* (Γ_c, map, f).

- The closure enviroment Γ_c is the relational representation of the λ-abstraction's lexical environment. It just consists of a copy of the current environment Γ when the λ-abstraction is compiled. Note that in the loop-lifted compilation, if a lambda expression occurs in the body of a for comprehension, each environment entry maps a variable to the values of the variable for all iterations.

- map is a query plan which produces a table with exactly two columns surr and def. It provides a mapping between the surrogate values that represent closures and the iteration identifiers used in the closure environment Γ_c. During application, it is used to map between the iteration scope in which the λ-abstraction is compiled (*compilation scope*) and the iteration scope in which the corresponding closure is applied (*application scope*).

- f stores the λ-abstraction that is compiled for the application.

The plan map is the core ingredient of the approach. It fulfills two functions:

1. The loop-lifting compilation is fundamentally based on iteration scopes, expressed in iteration identifiers in the iter columns. A λ-abstraction might not be applied in the iteration scope in which it is compiled, but in a different scope. However, the iteration identifiers used in Γ_c entries still depend on the iteration scope in which the λ-abstraction was compiled, so the iteration identifiers used in Γ_c are not aligned with those of the current scope. With the help of map, environment entries can be lifted to the current iteration scope.

2. When multiple distinct λ-abstractions are combined in a list, there must be a way to connect the surrogate values to the λ-abstractions they represent. map is stored separately for each λ-abstraction and only contains those surrogate values which refer to this exact lambda. It can thus be used during application to filter those surrogate values which refer to a specific function.

Each time the surrogate values are updated because of list concatenation (append) or branch merging (if, case), the surr column of the mapping has to be updated (see Section 3.3.3).

To make these concepts clear, we consider the FL expression

$$\mathsf{concatMap}(\lambda i.[\lambda x.x + i, \lambda x.x - i], [10, 20, 30])$$

as an example for the relational representation of lambda abstractions. Semantically, it generates a list of 6 closures, each containing one free variable i.

q

iter	pos	item$_1$
1	1	1
1	2	2
1	3	3
1	4	4
1	5	5
1	6	6

fs

$$\Bigg[\Bigg(\begin{array}{|cc|}\hline \text{surr} & \text{def}\\ \hline 1 & 1\\ 3 & 2\\ 5 & 3\\ \hline\end{array}, \Bigg\{ i \rightarrow \begin{array}{|ccc|}\hline \text{iter} & \text{pos} & \text{item}_1\\ \hline 1 & 1 & 10\\ 2 & 1 & 20\\ 3 & 1 & 30\\ \hline\end{array} \Bigg\}, \lambda x.x + i \Bigg),$$

$$\Bigg(\begin{array}{|cc|}\hline \text{surr} & \text{def}\\ \hline 2 & 1\\ 4 & 2\\ 6 & 3\\ \hline\end{array}, \Bigg\{ i \rightarrow \begin{array}{|ccc|}\hline \text{iter} & \text{pos} & \text{item}_1\\ \hline 1 & 1 & 10\\ 2 & 1 & 20\\ 3 & 1 & 30\\ \hline\end{array} \Bigg\}, \lambda x.x - i \Bigg)\Bigg]$$

Figure 3.17.: Example for the intermediate representation of closures.

The compilation of this expression produces query plans which would evaluate [3] to the tables shown in Figure 3.17. As the compilation is loop-lifted, both λ-abstractions are *lifted* into the iteration scope generated by the input list. The outer plan q contains the surrogate values in column item$_1$. From this column and the surr column of the map tables, it is clear that the first abstraction $\lambda x.x + i$ occurs in the result list at positions 1, 3 and 5, while the second abstraction $\lambda x.x - i$ occurs at positions 2, 4 and 6. The map table relates the surrogate values to the corresponding environment iteration identifiers 1, 2, 3. Since the list of functions stems from an iteration, there is only one entry for each λ-abstraction, together with closure environments which contain the values of the free variables for all iterations.

[3] We emphasize that this expression alone would not be accepted as its return type contains a function type.

$$\frac{\Gamma \cup \{x_1 \to row, \ldots, x_n \to row\} \vdash e = e' : \tau}{\Gamma \vdash \lambda x_1 \ldots x_n.e = \lambda x.\downarrow^{\tau}_{row}(e') : row} \quad (23)$$

$$\frac{\Gamma \vdash f = f' : \tau \quad {}_{i=1,\ldots,n}\big|\Gamma \vdash a_i = a'_i : \tau_i}{\Gamma \vdash f\ (a_1, \ldots, a_n) = f'\ \downarrow^{\tau_1}_{row}(a'_1), \ldots, \downarrow^{\tau_n}_{row}(a'_n)) : row} \quad (24)$$

$$\frac{\Gamma \vdash f = f' : \tau_f \quad \Gamma \vdash l = l' : \tau_l \quad p \in \{\mathsf{concatMap}, \mathsf{map}, \mathsf{takeWhile}, \mathsf{dropWhile}, \mathsf{groupByFlat}, \mathsf{sortByFlat}\}}{\Gamma \vdash p(f, l) = pl(f', \downarrow^{\tau_l}_{tbl}(l')) : row} \quad (25)$$

Figure 3.18.: Additional rules for implementation type inference (Section 3.1.3) to handle first-class functions.

3.3.2. Boxing of Closures

Figure 3.18 lists the additional rules necessary for the boxing phase on FL expressions. Closures are represented by a single surrogate value in a single row. Therefore, their implementation type is row (rule 23). For similar reasons as in Rule 21, the body of the λ-abstraction must have type row. Consider an expression

$$\mathsf{concatMap}(\lambda f.(f\ y), [\lambda x.e_1, \lambda x.e_2])$$

which iterates over a list of functions and applies each of them to some value y. If the functions return lists, the body expressions e_1 and e_2 could be of different implementation types, returning boxed and unboxed lists. In this case, boxing of the expression surrounding the application would fail.

For applications (Rule 24), the arguments are forced to implementation type row. This happens to keep the representation uniform: A closure that is applied multiple times could be called with list-typed arguments that are boxed and unboxed. In this case, the boxing would fail on the body.

Rule 25 handles the boxing of primitive higher order functions. As their second argument is always a list, it is unboxed if necessary. For most of the functions handled by Rule 25, this is perfectly fine: the result of the functional arguments is of a type which is represented by a single row, e.g. $Bool$ (takeWhile, dropWhile) or $(\alpha) :: Flat$ (groupByFlat, sortByFlat). For the concatMap function, however, we encounter a small difficulty.

Recall that the implementation type of the body expression of a for comprehension must be tbl. Rules 14 and 15 (Section 3.1.3) enforces this by adding an unbox operator to the body expression if necessary. This is possible because in the case of for, the body expression is fixed and directly accessible. However, the concatMap function applies not a fixed expression but a closure (i.e. a lambda abstraction), which results from an arbitrary expression. Therefore, the boxing phase can't unbox the body of the λ-abstraction that is eventually applied. This would however be necessary, as the body of λ-abstractions is generally forced to row (Rule 23).

The solution to this problem is to unbox the body of the functional argument not during the explicit boxing phase, but during the compilation: The compilation rule that handles the concatMap function compiles the functional argument expression f. The result of this compilation contains an fs component, which in turn contains all λ-abstractions which

```
concatMap(
  λt.
    [(team = t.1,
      maxps =
        box(
          concatMap(
            λpos.
              [(1 = y.1,
                2 = max(concatMap(λp1.[p1.eff], unbox(pos.2))))],
            groupByFlat(λp2.p2.pos, unbox(t.2))))))]
  groupByFlat(λp3.p3.team, players))
```

Listing 3.1: Boxed representation of Query $Q3$.

might be applied at this point. At this point, the bodies of the abstractions can be unboxed, if necessary.

Boxing of the FL program that is equivalent to Query $Q3$ (Listing 2.6) results in the following program:

box and unbox calls have been inserted at three places: The result of the function groupByFlat is a list of records. The second element however is a list that is stored in an inner table. Therefore, accesses to the second element have to be unboxed. The value that is stored in the $maxps$ field is a list and must therefore be boxed so that it can be stored in a record.

3.3.3. Merging of Plans

When plans are merged, e.g. by rules APPEND or SEQCONS, the fs components need to be appended, too. Basically, for every column which represents closures, the matching lists of closure records from the fs components have to be appended. However, when new surrogate values are computed during merging, the surrogate values in the surr column of the map plans are no longer aligned with those in the item columns. Therefore, after the merging, the surr columns have to be updated with the new surrogate values.

$$\frac{\begin{array}{l} \left.\begin{array}{l} q_i \equiv \sigma_{\mathsf{res}}\left(=_{\mathsf{res}:\langle \mathsf{ord},\mathsf{ord}'\rangle}\ (@_{\mathsf{ord}':o}\,(q_o))\right) \\ fs_i \equiv [(\Gamma_1, map_1, \lambda_1), \ldots, (\Gamma_n, map_n, \lambda_n)] \\ {\scriptstyle i=1,\ldots,n}\ \middle|\ map'_i \equiv \pi_{\mathsf{surr}:\mathsf{item}',\mathsf{def}}\left(map_i \ \bar{\bar{\bowtie}}_{\mathsf{surr},\mathsf{item}_1}\right) \\ fs'_i \equiv [(\Gamma_1, map'_1, \lambda_1), \ldots, (\Gamma_n, map_n, \lambda_n)] \end{array}\right| {\scriptstyle o=1,\ldots,m} \\ fs \equiv \{1 \to fs'_1 + + \cdots + + fs'_m\} \end{array}}{q_o \vdash fs_1, \ldots, fs_m \overset{fs}{\Rightarrow} fs} \quad \text{(APPENDFS)}$$

Rule APPENDFS is responsible for the updating of map plans. We again show only a special case and assume that only the column item_1 represents closures. To refresh the surrogate values in map plans, a mapping between old and new surrogate values is needed. A join between map plans and the merged outer plan which contains the newly computed surrogate values in item' would not be correct: the old surrogate values are no longer unique after the merge. We can however use the plan q_{nk} from rules APPEND and SEQCONS, which includes the original surrogate columns, the newly computed surrogate

column item′ and the column ord, which stores the concatenation order of the individual plans. ord is used to select for each original outer plan only those rows from q_{nk} that originate from this plan. After this, the old surrogate values are unique again and an equijoin is used to update the map plans.

3.3.4. Construction

$$\frac{\begin{array}{c} map \equiv \pi_{\mathsf{surr:iter,def:iter}}(loop) \\ cs \equiv Col\ 1 \\ fs \equiv \{1 \to [(\Gamma, map, \lambda x.e)]\} \\ q \equiv @_{\mathsf{pos:1}}\left(\pi_{\mathsf{iter,item_1:iter}}(loop)\right) \end{array}}{\Gamma; loop \vdash \lambda x.e \Mapsto (q, cs, \emptyset, \emptyset, fs)} \quad \text{(DEFLAMBDA)}$$

Rule DEFLAMBDA compiles λ-abstractions and generates the relational representation and closure list in the way described in Section 3.3.1. As usual, surrogate values are initially derived from iteration identifers. This means that both the surr and def columns of the map plan are derived from the iter column. A copy of the current environment Γ is stored in the closure records.

3.3.5. Application

In the general case, a whole loop-lifted list of closures will be applied. Since at compile time it is not known which closure is applied in which iteration, the body of every closure is included in the plan. For every closure, the map plan is used to restrict the current iteration context to a restricted sub-context containing only those iterations in which this closure is to be applied. Additionally, the map plan is used to lift the closure environment from the definition scope to the current application scope.

$$\frac{\begin{array}{l} \Gamma; loop \vdash f \Mapsto \left(q_f, cs_f, \emptyset, \emptyset, fs_f\right) ① \\ \Gamma; loop \vdash a \Mapsto (q_a, cs_a, ts_a, vs_a, fs_a) ② \\ fs_f \equiv \{1 \Rightarrow [(\Gamma_1, map_1, \lambda x_1.body_1), \ldots, (\Gamma_n, map_n, \lambda x_n.body_n)]\} \\ \quad \Big| \begin{array}{l} \Gamma_i \equiv \{\ldots, z \to (q_z, cs_z, ts_z, vs_z, fs_z), \ldots\} \\ q_i \equiv q_f \ \bar{\bar{\bowtie}}_{\mathsf{item_1,surr}}\ map_i\ ③ \quad map_{i\cdot l} \equiv \pi_{\mathsf{current:iter,def}}\,(q_i)\ ④ \\ loop_i \equiv \pi_{\mathsf{iter}}(q_i)\ ⑤ \\ q_{i\cdot a} \equiv q_a \ \bar{\bar{\bowtie}}_{\mathsf{iter,iter'}}\ \left(\pi_{\mathsf{iter':iter}}\,(loop_i)\right) ⑥ \\ q_{z\cdot l} \equiv \pi_{\mathsf{iter:outer,pos}}\left(map_{i\cdot l} \ \bar{\bar{\bowtie}}_{\mathsf{inner,iter}}\ q_z\right) ⑦ \\ \Gamma_{i\cdot a} \equiv \begin{array}{l} \overset{[cs_z]_<}{\{\ldots, z \Rightarrow (q_{z\cdot l}, cs_z, ts_z, vs_z, fs_z), \ldots\}} \\ \quad \cup \{x_i \Rightarrow (q_{i\cdot a}, cs_a, ts_a, vs_a, fs_a)\} \end{array} ⑧ \\ \Gamma_{i\cdot a}; loop_i \vdash body_i \Mapsto (q_{i\cdot r}, cs_{i\cdot r}, ts_{i\cdot r}, vs_{i\cdot r}, fs_{i\cdot r}) ⑨ \end{array} \ i=1,\ldots,n \\ (q_{1\cdot r}, cs_{1\cdot r}, ts_{1\cdot r}, vs_{1\cdot r}, fs_{1\cdot r}), \ldots, (q_{1\cdot r}, cs_{1\cdot r}, ts_{1\cdot r}, vs_{1\cdot r}, fs_{1\cdot r}) \overset{sq}{\Mapsto} (q, cs, ts, vs, fs)\ ⑩ \end{array}}{\{\ldots, z \Rightarrow (q_z, cs_z, ts_z, vs_z, fs_z), \ldots\}; loop \vdash (f\ a) \Mapsto (q, cs, ts, vs, fs)}$$

(APPLYCLOSURE)

Rule APPLYCLOSURE handles the application of non-primitive functions. For brevity, only the special case is shown in which there is only one argument. It can be extended straight forwardly to the general case. In the following, we describe the individual steps of the rule in detail.

①, ② The function expression f and the argument expression a are compiled.

③ The plan q_f contains surrogate values for all closures. q_f is restricted by a join with the map_i plan to those surrogate values which relate to the current closure. As the surr column of map contains only those surrogate values which match the current closure, rows from q_f which relate to other closures do not find a join partner and are removed.

④ The mapping map_i between surrogate values and iteration identifiers from the definition scope is converted into a mapping $map_{i \cdot l}$ between the application scope and the definition scope.

⑤ The loop'_i plan represents an iteration sub-context (similar to the sub-contexts in the compilation of if and case expressions) which contains only those iterations in which the current closure i has to be applied.

⑥ The result of the compilation of the argument expression contains the argument values for all closures. A join with the iteration context $loop_i$ restricts the argument values to the current iteration sub-context. The result $q_{i \cdot a}$ contains only the arguments for the iterations in which the current closure is applied.

⑦ The mapping $map_{i \cdot l}$ is used to lift all entries in the closure environment Γ_i from the definition scope into the current application scope.

⑧ The lifted closure environment is expanded with a binding for the bound variable x_i, forming an environment $\Gamma_{i \cdot a}$ which contains bindings for all free and bound variables of the closure body.

⑨ In the restricted iteration sub-context $loop_i$ and the lifted closure environment $\Gamma_{i \cdot a}$, the body expression $body_i$ of the current closure is compiled. This results in a plan which contains the results of all applications of the current closure.

⑩ In the same way as in the compilation of if and case expressions, the plans stemming from the individual closures have to be merged into one plan. This is accomplished by the function $\overset{sq}{\Rightarrow}$.

3.3.6. An Example

We continue the example from Section 3.3.1 by iterating over the list of closures and applying each of them:

$$\mathsf{concatMap}(\lambda f.[f(42)], \mathsf{concatMap}(\lambda i.[\lambda x.x + i, \lambda x.x - i], [10, 20, 30]))$$

Figure 3.19 shows a part of the query plan that results from the compilation of this expression. As the complete plan is too large to be included here, we only show the sub-plan

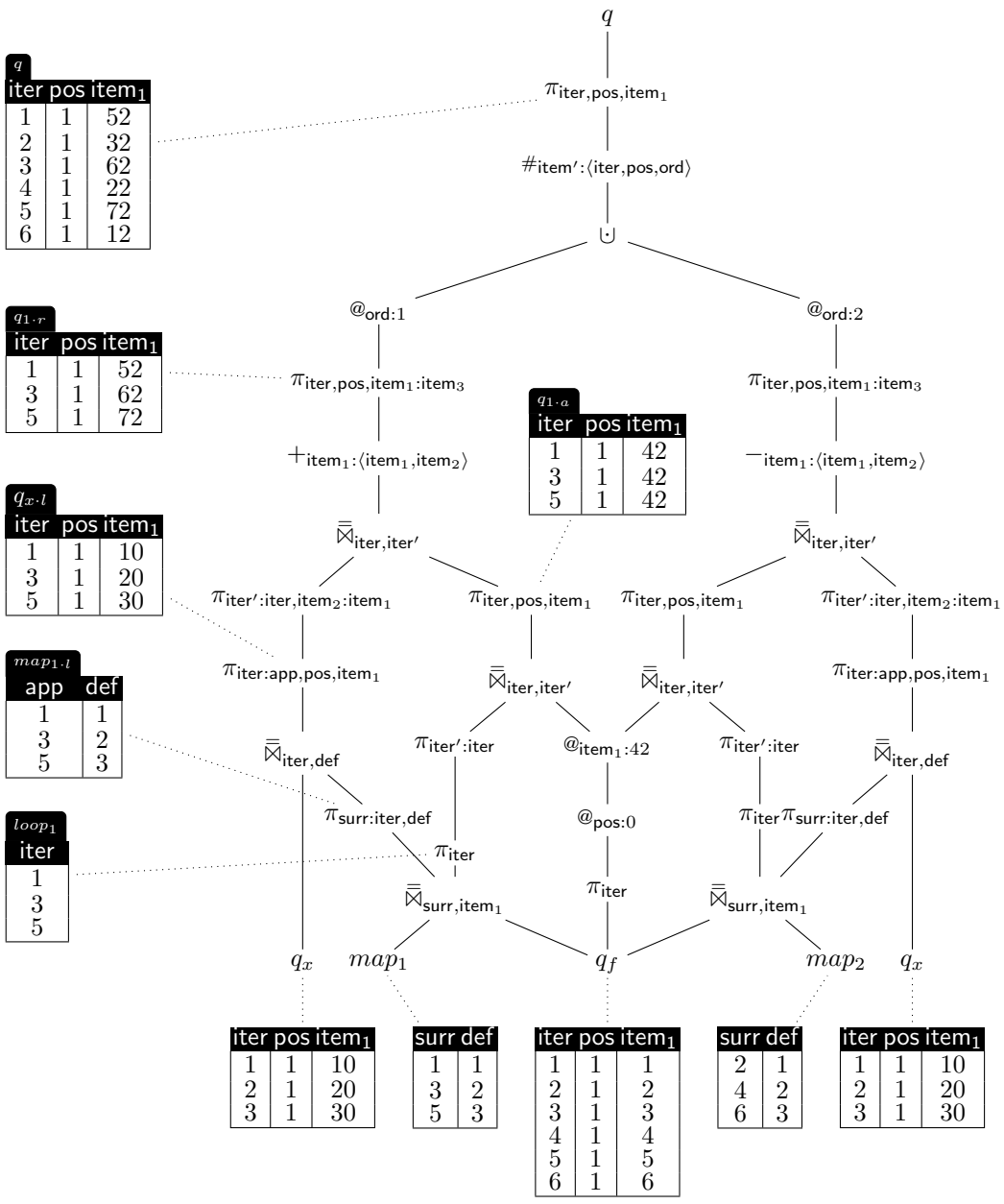


Figure 3.19.: Query plan resulting from function application (Example in Section 3.3.6).

that is generated by the rule APPLYCLOSURE. The plan q_f is the equivalent to plan q from Figure 3.17 that has been lifted into a new iteration scope according to the outer for iteration. Tables map_1, map_2 and q_x are the relational representations of the map and Γ_c components of the closure rows.

Tables on the lefthand side show the intermediate tables that result from the individual steps of rule APPLYCLOSURE (only for the first closure). Table q is the overall result of the body of the outer for iteration. Of course, this plan still needs to be mapped back into the original iteration context. The plan is perfectly symmetric because the bodies of the λ-abstractions from which the closures originate differ only in the operator that is applied $(+, -)$.

3.3.7. Primitive Higher-Order Functions

The availability of closures as first-class values allows compilation of multiple primitive functions which have functional arguments, e.g. groupByFlat, takeWhile, dropWhile, any and all, as well as general expressions used as the body of for comprehensions. Although equivalents to some of these functions have been available in the FERRY language described e.g. in [15], the functional arguments were not first-class values, but a syntactic convention for literal λ-abstractions. These literal abstractions were handled in the same way as the body expression of for iterations in Section 3.1.11.

If functional arguments of primitive functions are indeed literal λ-abstractions, converting them to closures and applying the closures would be highly inefficient because of the map overhead. Although the rules in this section only include the general case of closure application, the actual implementation cares for the special case and uses literal λ-abstractions in the direct way: parameter variables of the abstraction are bound to the compiled argument expressions and in the resulting environment, the body expression of the abstraction is compiled (essentially, this is *lifted* β-reduction).

Every rule for a primitive higher-order function has to apply at least one functional argument. We denote with $\overset{app}{\Rightarrow}$ a function which essentially performs the same steps as rule APPLYCLOSURE. However, it does not take FL expressions for the closure and argument parts as arguments, but the results of the compilation of these expressions:

$$\Gamma; loop \vdash \left(q_f, fs_f\right), (q_a, cs_a, ts_a, vs_a, fs_a) \overset{app}{\Rightarrow} (q, cs, ts, vs, fs)$$

Aditionally, every rule which compiles a primitive higher order function has to lift the input list to apply the functional argument to each of its elements. To keep the presentation short, we denote with $\overset{lift}{\Rightarrow}$ a function which takes a query plan q as argument and produces the various components of the lifting from rule FOR: the original plan augmented with an inner column containing new iteration identifiers, the lifted list q_v, the plan map which relates the outer and inner iteration context, $loop$ which represents the inner iteration context and the environment Γ', in which all plans have been lifted to the inner iteration context:

$$\Gamma \vdash q \overset{lift}{\Rightarrow} (q_l', q_v, map, loop', \Gamma')$$

$$\frac{\begin{array}{c}\Gamma; loop \vdash l \Rightarrow (q_l, cs_l, ts_l, vs_l, vs_l) \\ \Gamma \vdash q_l \overset{lift}{\Rightarrow} (q_l', q_v, map, loop', \Gamma') \\ \Gamma'; loop' \vdash f \Rightarrow \left(q_f, cs_f, \emptyset, \emptyset, fs_f\right) \\ \Gamma'; loop' \vdash (q_f, fs_f), (q_l, cs_l, ts_l, vs_l, fs_l) \overset{app}{\Rightarrow} (q_e, cs_e, ts_e, vs_e, fs_e) \\ q \equiv \pi_{\mathsf{iter:outer,pos:pos'},[cs_e]_<} \left(\varrho_{\mathsf{pos'}:\langle \mathsf{iter,pos}\rangle} \left(q_e \,\bar{\bar{\bowtie}}_{\mathsf{iter,inner}}\, (\pi_{\mathsf{outer,inner}} map)\right)\right)\end{array}}{\Gamma; loop \vdash \mathsf{concatMap}(f, l) \Rightarrow (q, cs_e, ts_e, vs_e, fs_e)} \quad \text{(CONCATMAP)}$$

Concatenated Mapping

Rule CONCATMAP handles applications of the primitive function concatMap, which replaces for comprehensions. The mapping into a new iteration scope is done as usual, resulting in a new $loop'$ plan and a lifted environment Γ'. However, instead of binding the lifted list to the iteration variable and compiling the body directly, the functional expression f has to be compiled first. As f is applied to all elements of the list l, it has to be compiled in the new iteration context $loop'$ ①. The auxilliary function $\overset{app}{\Rightarrow}$ is then used to apply f to all list elements ②. The backmapping step is again performed as usual.

Note that the rule shown here does not include the (optional) unboxing of the functional argument's bodies described in Section 3.3.2. This has to happen on the fs_f component after step ①.

Ordering

$$\frac{\begin{array}{c}\Gamma; loop \vdash l \Rightarrow (q_l, cs_l, ts_l, vs_l, vs_l) \\ \Gamma \vdash q_l \overset{lift}{\Rightarrow} (q_l', q_v, map, loop', \Gamma') \\ \Gamma'; loop' \vdash p \Rightarrow \left(q_p, cs_p, \emptyset, \emptyset, fs_p\right) \\ \Gamma'; loop' \vdash (q_p, fs_p), (q_v, cs_l, ts_l, vs_l, fs_l) \overset{app}{\Rightarrow} (q_o, cs_o, ts_o, vs_o, fs_o) \\ q \equiv q_l' \,\bar{\bar{\bowtie}}_{\mathsf{inner,iter'}} \left(\pi_{\mathsf{iter'}:\mathsf{iter},\mathsf{item}_{|cs_l|+1}:\mathsf{item}_1,\mathsf{item}_{|cs_l|+n}:\mathsf{item}_m} (q_o)\right) ① \\ q' \equiv \pi_{\mathsf{iter,pos:pos'},[cs_l]_<} \left(\varrho_{\mathsf{pos'}:\langle \mathsf{item}_{|cs_l|+1},\ldots,\mathsf{item}_{|cs_l|+n},\mathsf{iter,pos}\rangle} (q)\right) ②\end{array}}{\Gamma; loop \vdash \mathsf{sortByFlat}(p, l) \Rightarrow (q', cs_e, ts_e, vs_e, fs_e)} \quad \text{(SORTBYFLAT)}$$

Rule SORTBYFLAT works principally in the same way as the ordering aspect of rule FOR (Section 3.1.11): the results of the ordering expression p are used as criteria by the ϱ operator for the computation of new positions. There are, however, two differences: First, the ordering expression p is an arbitrary functional expression, not a literal lambda abstraction. Second, the ordering is performed stand-alone, so that not the result of a for comprehension but the input list is ordered. The results of the ordering expression are aligned with the corresponding elements of the input list via an equijoin with q_l: q_l' contains the column inner which is aligned with the inner iteration context in column iter of q_o ②. For each element of the input list, the join thus adds the corresponding result of the ordering expression for this element. As before, these results are used as criteria for the ϱ operator and yield new positions ②.

Mapping

$$\frac{\begin{array}{c}\Gamma; loop \vdash l \Rightarrow (q_l, cs_l, ts_l, vs_l, vs_l) \\ q_l' \equiv \#_{\mathsf{inner}:\langle \mathsf{iter},\mathsf{pos}\rangle}(q_l) \quad q_v \equiv @_{\mathsf{pos}:1}\left(\pi_{\mathsf{iter}:\mathsf{inner},[cs_l]_<}\left(q_l'\right)\right) ② \\ map \equiv \pi_{\mathsf{outer}:\mathsf{iter},\mathsf{inner},\mathsf{pos}':\mathsf{pos}}\left(q_l'\right) \quad loop' \equiv \pi_{\mathsf{iter}}(q_v) \\ \Gamma' \equiv \left\{\ldots, x \Rightarrow \left(\pi_{\mathsf{iter}:\mathsf{inner},\mathsf{pos},[cs_l]_<}\left(q_x \mathbin{\bar{\bar{\bowtie}}}_{\mathsf{iter},\mathsf{outer}} map\right), cs_x, ts_x\right), \ldots\right\} \\ \Gamma'; loop' \vdash f \Rightarrow \left(q_f, cs_f, \emptyset, \emptyset, fs_f\right) ① \\ \Gamma'; loop' \vdash (q_f, fs_f), (q_l, cs_l, ts_l, vs_l, fs_l) \overset{app}{\Rightarrow} (q_e, cs_e, ts_e, vs_e, fs_e) ② \\ q \equiv \pi_{\mathsf{iter}:\mathsf{outer},\mathsf{pos}:\mathsf{pos}',[cs_e]_<}\left(q_e \mathbin{\bar{\bar{\bowtie}}}_{\mathsf{iter},\mathsf{inner}} map\right)\end{array}}{\{\ldots, x \to (q_c, cs_x, ts_x), \ldots\}; loop \vdash \mathsf{map}(f, l) \Rightarrow (q, cs_e, ts_e, vs_e, fs_e)} \quad (\textsc{Map})$$

In the concatMap function, every element of the input is mapped to a list. All result lists are flattened into one list. In rule CONCATMAP, new positions computed by the ϱ operator flatten the result lists from the individual iterations. In contrast, the map function maps each element of the input list to exactly one element of the output list. Because of this, we do not need to compute new positions during the backmapping step. Instead, the map plan stores the original positions in column pos′, which are then re-used in the result list.

Rule MAP can be extended straight forwardly to handle the mapi function. The plan q_i that computes the index argument passed to every invocation of the functional argument can be obtained from q_l' by recycling the pos column:

$$q_i \equiv \pi_{\mathsf{iter}:\mathsf{inner},\mathsf{pos},\mathsf{item}_1:\mathsf{pos}}\left(q_l'\right)$$

Grouping

$$\frac{\begin{array}{c}\Gamma; loop \vdash l \Rightarrow (q_l, cs_l, ts_l, vs_l) ① \\ \left.\begin{array}{c}\Gamma \vdash q_l \overset{lift}{\Rightarrow} (q_l', q_v, map, loop', \Gamma') \\ \Gamma'; loop' \vdash f \Rightarrow \left(q_f, cs_f, \emptyset, \emptyset, fs_f\right) \\ \Gamma'; loop' \vdash (q_f, fs_f), (q_l, cs_l, ts_l, vs_l, fs_l) \overset{app}{\Rightarrow} \left(q_{ge}, cs_{ge}, ts_{ge}, vs_{ge}, fs_{ge}\right)\end{array}\right\} ② \\ cs_{ge}' \equiv \mathsf{shift}(|cs_e|, cs_{ge}) ③ \\ q_1 \equiv \varrho_{\mathsf{key}:\langle [cs_{ge}']_<\rangle/\mathsf{iter}}\left(q_v \mathbin{\bar{\bar{\bowtie}}}_{\mathsf{inner},\mathsf{iter}'}\left(\pi_{\mathsf{iter}':\mathsf{iter},\ [cs_{ge}']_<:cs_{ge}}\right)\right) ④ \\ q_o \equiv \delta\left(\pi_{\mathsf{iter},\mathsf{pos}:key,\mathsf{key}':\mathsf{key},\ [cs_{ge}:[cs_{ge}']_<]_<}(q_1)\right) ⑤ \quad q_i \equiv \pi_{\mathsf{iter}:\mathsf{key},\mathsf{pos},\ [cs_e]_<}(q_1) ⑥ \\ ts_o \equiv \{\mathsf{key} \to (q_i, cs_e, ts_e, vs_e)\} ⑦ \quad cs_o \equiv Map\ [(1, cs_{ge}), (2, Col\ \mathsf{key})] ⑧\end{array}}{\Gamma; loop \vdash \mathsf{groupBy}(g, l) \Rightarrow (q_o, cs_o, ts_o, \emptyset, \emptyset)} \quad (\textsc{GroupBy})$$

Rule GROUPBY handles occurences of the groupByFlat function, providing access to a database system's grouping functionality. Note that in groupByFlat(p, l), the type of the result of the projection function p is restricted to values with a flat relational representation.

This ensures that the result of the grouping expression can be represented in a single table. Otherwise, the grouping could not be computed efficiently. As the values for each group are a nested list, GROUPBY always delivers a nested result.

Rule GROUPBY first applies the grouping expression to every list element and then computes the actual grouping. The result is split into an inner and an outer plan. In the following, we describe the individual steps in detail:

① First, the list that is to be grouped has to be compiled.

② Next, the functional argument f is applied to all elements of the input list. Basically this is handled in the same way as in rule CONCATMAP with one difference: no backmapping to the original scope context is performed.

③ As the result of the grouping expression and the original input are joined into one plan in the next step, column names are shifted to avoid overlaps.

④ An equijoin aligns elements of the input list with the corresponding result of the grouping expression. Note that this join uses the q_l' plan as input, which has both the original iteration identifiers (iter) as well as the new ones created for the grouping expression (inner). Because of this, the explicit backmapping via the map plan performed in the rule CONCATMAP is avoided. The actual grouping is performed with an application of the row ranking operator ϱ. Based on result of the grouping expression as the ranking criterion, it provides rank values which represent the individual groups in column key. The row ranking is performed grouped by the iter values, providing an independent grouping for each iteration. Because of the use of row ranking instead of numbering, rows which belong in the same group are assigned the same rank.

⑤ Both the outer and the inner plan are derived from the same plan q_i. The table computed by the outer plan contains for every group the result of the grouping expression key and a column with surrogate values referencing the inner table which stores the content of the individual groups. An application of the distinct operator δ removes all duplicate rows, reducing the table to one row per group. The surrogate columns don't need to be computed explicitly. Instead, they can be recycled from the unique grouping keys.

⑥ For the inner plan q_i, the grouping key is removed with a projection. In the iter column, the grouping keys are used, which are aligned with the surrogate values in the outer plan q_o.

⑦ The ts_o component stores the inner plan which contains the values for each group.

⑧ As the overall result is a record, an appropriate cs component is created that stores the record layout of the result.

Additional Primitive Higher-Order Functions: takeWhile, dropWhile

The rules for the remaining functions takeWhile and dropWhile, share the same pattern: A functional argument which serves as a boolean predicate is applied to (i.e. mapped over) all elements of the input list. From the boolean results of the predicate, the last position at

which the predicate was true is obtained. Based on whether the function is takeWhile or dropWhile, the list elements up to this position are either taken or dropped.

$$\frac{\begin{array}{l}\Gamma; loop \vdash l \Rightarrow (q_l, cs_l, ts_l, vs_l, vs_l)\ ① \\ \left.\begin{array}{l}\Gamma \vdash q_l \overset{lift}{\Rightarrow} (q_l', q_v, map, loop', \Gamma') \\ \Gamma'; loop' \vdash f \Rightarrow \left(q_f, cs_f, \emptyset, \emptyset, fs_f\right) \\ \Gamma'; loop' \vdash (q_f, fs_f), (q_l, cs_l, ts_l, vs_l, fs_l) \overset{app}{\Rightarrow} \left(q_{pr}, cs_{pr}, ts_{pr}, vs_{pr}, fs_{pr}\right)\end{array}\right\}\ ② \\ q' \equiv \pi_{\mathsf{iter,pos,res},[cs_l]_<}\left(q_l' \bar{\bar{\bowtie}}_{\mathsf{inner,iter'}}\ \left(\pi_{\mathsf{iter'}:\mathsf{iter,res}:\mathsf{item}_1}\ (q_{pr})\right)\right)\ ③ \\ q_m \equiv \mathrm{GRP}_{\mathsf{pos'}:\mathrm{MIN}(\mathsf{pos})/\mathsf{iter}}\ \left(\sigma_{pos'}\ \left(\neg_{\mathsf{pos'}:\mathsf{res}}\ (q')\right)\right)\ ④ \\ q_e \equiv \pi_{\mathsf{iter,pos},[cs_l]_<}\left(q_l \bar{\bar{\bowtie}}_{\mathsf{iter,iter'}}\ \left(\pi_{\mathsf{iter'}:\mathsf{iter}}\ (loop \setminus (\pi_{iter}\ (q_m)))\right)\right)\ ⑤ \\ q'' \equiv q_e \uplus \left(\pi_{\mathsf{iter,pos},[cs_l]_<}\left(\sigma_{\mathsf{res'}}\left(>_{\mathsf{res'}:\langle \mathsf{pos'},\mathsf{pos}\rangle}\ \left(q' \bar{\bar{\bowtie}}_{\mathsf{iter,iter'}}\ \left(\pi_{\mathsf{iter'}:\mathsf{iter,pos'}}\ (q_m)\right)\right)\right)\right)\right)\ ⑥\end{array}}{\begin{array}{c}\Gamma; loop \vdash \mathsf{takeWhile}(p, l) \Rightarrow (q'', cs_l, ts_l, vs_l, fs_l) \\ (\textsc{TakeWhile})\end{array}}$$

① First, the list argument is compiled.

② The predicate p is applied to all members of the list for each iteration. Again, this happens in the same way as in CONCATMAP, except that the backmapping step is omitted.

③ An equijoin aligns the elements of the original lists with the corresponding predicate results. Note that the join implicitly performs the backmapping of the predicate results into the original iteration scope: The join is performed not with q_l, but with q_l', which is itself in the original iteration scope but includes the iteration identifiers of the inner iteration scope in column inner.

④ A pos′ column is added which contains for all iterations the minimal position, at which the predicate is not true.

⑤ The rule has to pay attention to a special case: The predicate might never be false so that there is no minimal position. Set difference with the $loop$ plan delivers those iterations in which there is no minimal position. A join between the result and q' keeps only those iterations in which the predicate is never false. These have to be included completely in the overall result.

⑥ By comparing list positions with the minimal positions from step ④, this step obtains all list elements for which the predicate is true. Disjoint union brings together the taken list elements on the right side and rows from lists for which the predicate is not false at all. The sets of iteration identifiers from the tables encoded by the plans on which the $\uplus$ operator is applied are always disjoint: The plan on the right side only contains iterations which were already in q_m because of the equijoin. As q_e is created exactly from those iterations which are not in q_m, q_e and q_m are already disjoint.

We omit the rule DROPWHILE here, as it works in exactly the same way as TAKEWHILE.

3.4. Additional Constructs

In this section we explain briefly how some minor aspects of LINKS can be handled in the loop-lifting framework: comparison of non-atomic values (3.4.1) and full support for LINKS' regular expressions (3.4.2).

3.4.1. Comparison Operators

Having polymorphic types $(\alpha, \alpha) \to Bool$, LINKS' comparison operators can be applied to operands of any type as long as the types of both operands are equal. Rule BINOP handles atomic values but clearly can't compare non-atomic values like records, lists and variants. This would lead to a loophole in the compilation scheme: type-correct programs fail at runtime (i.e. query compile time). However, with some extra effort, comparisons on non-atomic values can be handled algebraically as well with the same semantics as the native LINKS implementation.

From the relational representation of values (Section 3.1.2) follows that the comparison has to work on multiple levels: Individual values that are represented by one (atomic values and boxed lists) or two (variant values) *columns*, multiple such values that are packed horizontally in a record represented by a *row*, and finally unboxed lists represented by *tables*, packing values vertically. In the following, we outline how comparison on these levels works. For brevity, we describe only the equality operator $==$. Although things are a bit more complicated for the larger-than operator $>$, it can be implemented for composed values as well [4].

In LINKS, two lists are equal iff they are of the same length and elements at the same positions are equal. This semantic can be expressed conveniently in terms of functions that have been already described as

$$l_1 == l_2 \quad = \quad (\mathsf{length}(l_1) == \mathsf{length}(l_2)) \;\&\&\; \mathsf{all}(\mathsf{map}((==), \mathsf{zip}(l_1, l_2)))$$

This means that the algebraic code implementing these functions can be reused for the implementation of list equality, the only exception being that the inputs to these functions are not FL expressions but query plans. Note that the map function applies the equality comparison to the zipped lists iteratively, i.e. to a loop-lifted sequence of pairs. This is important when handling comparison on boxed lists and variants (see below).

Records represented by *rows* are equal iff all fields of the records are equal. The type system ensures that the structure of records used as operands of $==$ is the same. Since nested records are mapped to a flat sequence of columns containing only the leaf fields (see Section 3.1.2), only these columns need to be compared. More precisely, what needs to be compared are the corresponding Tag (variants) and $Column$ (atomic values, boxed lists) entries from the cs components of both operands. The boolean results of the individual comparisons are then aggregated pairwise by the algebraic $\&\&$ operator, resulting in a single boolean result for each row.

Atomic values are simply handled by rule BINOP. The remaining constructs are handled as follows:

[4] With the exception of variant values, for which LINKS defines no order.

Boxed lists To compare nested lists, the boxed lists must be unboxed first. This happens in the exact same way as performed by rule UNBOX. After unboxing, the comparison scheme is applied to the individual unboxed lists. For unboxing, it is essential that the input list is loop-lifted, so that each iteration contains only one row. Otherwise, unboxing would violate the first normal form.

Variants Variant values are equal iff their tags are equal and the tagged values are equal. Relationally, this is implemented as follows: First, rows from the same iterations are aligned with an equijoin. Next, the set of iterations is divided into those in which the tags are not equal and those in which the tags are equal. For the former, the result is $false$. For the latter, the tagged values need to be unboxed and compared. Since the tagged values can be different for each tag, tagged values values must be unboxed and compared for each tag individually. The results of the comparisons for all tags are then unionend.

While lists, records and variants can be compared, there is no (meaningful) way to compare e.g. functions. Such comparisons would still fail at runtime [5]. This problem can however be avoided by restricting the polymorphism of the comparison operators (see Section 4.2.2).

3.4.2. Regular Expressions

LINKS supports Perl-like syntax for string matching with regular expressions: the result of the expression `s ~ /p/` is true if the string-typed expression `s` matches the pattern `p`. Patterns can contain the usual pattern operators: "." which matches any single character, "(...)" for grouping of characters, "[...]" which matches any of a set of characters and the usual quantifiers "+", "*" and "?". The Murrayfield query translator maps the matching operator to the SQL **`LIKE`** operator. Unfortunately, **`LIKE`** can't handle most of the pattern operators available in LINKS regular expressions, but basically only combinations of "." and "*". This leads to yet another case where the compilation of type-correct queries fails at runtime [4].

The new query backend avoids this problem by using the SQL:1999 **`SIMILAR TO`** operator instead of **`LIKE`**. **`SIMILAR TO`** supports all required pattern operators and is available in all standard-conforming RDBMS's.

Static patterns alone are not very interesting. LINKS regular expression strings can contain expressions `{e}`, where `e` is an arbitrary string-typed expression which is used as part of the pattern. While Murrayfield fails on expressions `e` that are not literal strings, the new backend can handle any databaseable string-typed expression in this place, because on the SQL side the pattern of the `SIMILAR TO` operator does not have to be a literal string but can be an arbitrary column containing pattern strings. A string matching expression `s ~ /.*{s}.*/` is converted to an internal SL expression `s ~ (".*" ^^ (s ^^ ".*"))`, combining the binary operators `^^` (string concatenation) and `~` (string matching). The algebraic compilation of this expression then consists just of multiple applications of rule BINOP which handles these binary operators. This means that patterns can be constructed from arbitrary string expressions, which for example include string values from database tables.

[5] In fact, they also fail in the normal LINKS interpreter.

4. Embedding FERRY into LINKS

Having started with the actual query compilation, we now describe how this compilation technique is embedded in the LINKS compiler to obtain a FERRY-based query backend. First, we outline the architecture of the backend (Section 4.1). Then, the missing steps are described: minor changes to type system and syntax (Section 4.2), the translation of the LINKS intermediate representation into a simpler query representation (Section 4.3.1) and an optimization phase on the query representation (Section 4.3.2). Algebraic optimization and SQL code generation are provided by the external component Pathfinder, which we briefly describe in Section 4.4. As an alternative to algebraic closure conversion, defunctionalization was considered. Section 4.5 describes defunctionalization in the context of query compilation and justifies why this approach was abandoned.

4.1. Architecture

This section summarizes the steps that queries as part of a LINKS program take and refers to the relevant sections in the remainder of this work. Figure 4.1 provides an overview over the architecture.

1. The compiler frontend parses, typechecks and desugars the program. The frontend was only in the scope of this thesis because two minor changes are necessary: new subkinds are needed to lessen the restriction on the type of queries and declarations of table handles need to include information about the *key columns* of a table (Section 4.2).

2. The interpreter evaluates the desugared intermediate representation of the program. When a query is encountered during evaluation, it is passed to the query compiler.

3. The query compiler is responsible for compiling the query into a bundle of query plans. The query compiler's input consists of two components: the intermediate representation of the query and the environment which contains the result of the evaluation of expressions outside of the query.
   a) Intermediate representation and value environment are combined into a simpler query representation which unifies both forms and only includes those constructs that can appear in queries (Section 4.3.1).
   b) The query representation is optimized by an application of rewrite rules that perform partial-evaluation-style rewrites, including value and variable propagation, reduction of record operations, `case` expressions and `if` conditionals over known values and aggressive function inlining. Additionally, this phase rewrites some primitive functions into combinations of other primitive functions and changes the list representation (Section 4.3.2).

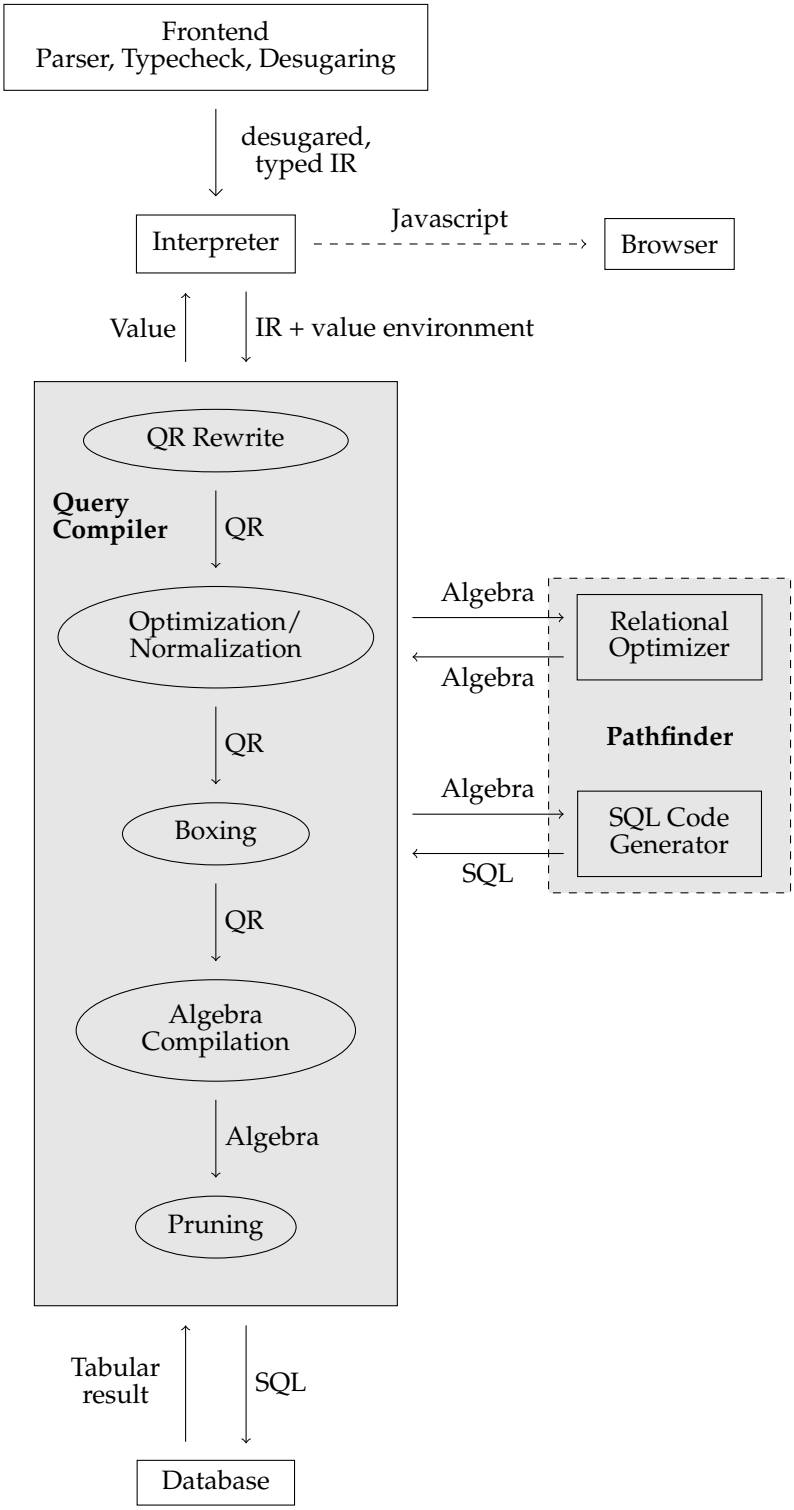


Figure 4.1.: Architecture of the FERRY-based query backend.

c) According to the method outlined in Section 3.1.3, the QR expression is annotated with box/unbox calls to obtain the correct relational list representation.

d) The boxed query representation is compiled to logical algebra, following the FERRY compilation scheme described in Section 3.1.

e) Finally, the pruning phase removes literal empty tables as described in Section 3.1.10.

4. The query plans as generated by the compilation scheme are generally large and feature an exotic shape that often overwhelms the optimizers of database systems. An external algebra optimizer – part of the Pathfinder compiler – employs a set of algebraic rewrites which in most cases dramatically shrink the size of query plans (Section 4.4).

5. An external code generator – also part of Pathfinder – is used to generate a bundle of SQL:1999 queries from the logical algebra plans (Section 4.4).

6. The bundle of SQL queries is executed on the database, which returns tabular results for each SQL query. The bundle of tabular results must be assembled into a LINKS value according to its type. We do not describe this step as it is very straight-forward. The assembled value is finally returned as the result of the query to the regular interpreter, which continues the evaluation of the program.

4.2. Frontend

This section describes changes made to the frontend of the LINKS compiler.

4.2.1. Syntax

In LINKS, references to database-resident tables are declared using the **table** keyword. For example, a reference to a table "players" with the schema shown in Figure 2.4 would be declared as follows:

```
table players with (id:Int,
                    name:String,
                    team:String,
                    pos:String,
                    eff:Int)
              from db
```

The information which columns of a table form (composite) keys is an essential ingredient for the Pathfinder algebraic optimization phase (see Section 4.4). Key information must be manually declared as part of the table reference declaration. To this end, the **table** syntax is extended with a keyword **tablekeys** which must be followed by a literal list of lists of string-valued column names. Every inner list describes one – possibly composite – key, which is formed by the contained column names. The reference to the table "players" is then declared as shown in Listing 4.1.

Since the information about table keys play no role in the algebraic compilation itself and is only fed to Pathfinder, we ignore this aspect in the description of the compilation.

```
table players with (id:Int,
                    name:String,
                    team:String,
                    pos:String,
                    eff:Int)
              tablekeys [["id"]]
              from db
```

Listing 4.1: Declaration of table reference with key information.

4.2.2. Type System

The only subkind supported so far in the LINKS type system is $Base$, which restricts type variables to be instantiated only to atomic types. The type system uses $Base$ to restrict the return type of queries, the type of table comprehensions (both to lists of records of base types: $[(|\ r :: Base)]$) and the type of certain functions (e.g. sortByBase) (Section 2.1).

The FERRY-based query backend lifts the restrictions on query return types considerably by allowing nesting and variant types. However, a query still must not return values of arbitrary types. It can, for instance, not return functions. Therefore, we need a new subkind which imposes a lighter restriction on the type of queries. We introduce two new subkinds:

- Type variables with the subkind $Query$ can be instantiated to all type combinations which might be returned by a query. That is: atomic types, record types, list types, variant types and arbitrary combinations of those.
- Type variables with the subkind $Flat$ can be instantiated to all type combinations which are represented relationally via a single tuple. $Flat$ type variables are therefore limited to atomic values and records. As the relational encoding of records only stores the leaf fields, record fields can again contain records, as long as their leaf fields only contain atomic values.

The $Flat$ subkind is used to restrict type variables in the types of the primitive functions sortByFlat, groupByFlat and nubFlat. These functions use the row ranking operator to compute rank values for tuples according to the values of their item columns, which are then used to obtain the distinct values (nubFlat, groupByFlat) or to order list elements (sortByFlat). The ranking operator can efficiently compute rank values using multiple columns as ranking criteria. Those columns must not contain surrogate values, as otherwise the contents of the corresponding inner tables would have to be taken into consideration. As a consequence, the projection functions of the sortByFlat and groupByFlat functions can return any type which has a flat relational representation, and nubFlat computes the distinct elements of a list of flat elements. In contrast, the projection function of the sortByBase function is limited to records of atomic values because the Murrayfield translation maps sortByBase directly to a SQL **`ORDER BY`** expression.

With the new subkinds, $Query$ is used to restrict the return type of queries. The restriction on the type of table comprehensions is removed altogether. This means that, for instance, table comprehensions can now return lists of functions, which was not possi-

ble before. As every table comprehension is implicitly contained in a query block, the type restriction on queries ensures that every function is applied or discarded, but not returned.

Additionally, the types ranged over by $Query$ type variables are exactly those types which can be handled by the comparison operators $==$ and $>$, both in the normal LINKS interpreter and algebraically in queries (Section 3.4.1). By restricting the types of these operators to $(\alpha :: Query, \alpha :: Query) \to Bool$, the comparison operators can be made type-safe.

The definition of subkinds is currently hard-coded in the LINKS type checker. This is very unflexible and thus a general mechanism would be beneficial that allows the definition of new subkinds. Ultimately, this could result in a mechanism along the lines of Haskell's type classes [30].

4.3. Internal Representation and Optimization

The input to the query compiler consists of two parts. The query itself is passed in the form of the internal representation of LINKS programs: an intermediate representation (in the following referred to as *IR*) based on A-normal forms (ANF). ANF is an intermediate representation commonly used in functional compilers [8]. The distinctive trait of ANF is that it requires intermediate computations that occur for instance as arguments in function applications to be let-bound. This property makes it easier to perform optimizing rewrites. For example, β-reduction is sound on ANF [1].

Figure 4.2 shows a simplified version of the LINKS IR. All constructs that can not appear in queries are omitted. The actual IR includes amongst others additional constructs for recursive function bindings, XML values and continuation handling. Since the type-and-effect system forbids their use in queries, the query compiler does not need to handle them. Also, the real LINKS IR includes type information in the form of type annotations on value and function bindings to allow the reconstruction of the types of IR terms. We omit these type annotations because the algebraic compilation does not make use of type information anyway and so the translation to the untyped QR discards all type information.

The grammar distinguishes between values and tail computations. This forces all function arguments that are not neither constants, variables nor data constructors (records, lists and variants) but intermediate computations to be bound to a variable.

A convenient property of the LINKS IR is the uniqueness of variable names, so that a variable name is bound exactly once in a program. This makes the compilation simpler because it does not have to keep track of which bindings are visible.

The representation of the query itself is not sufficient to compile the query: a query might not be closed but might (and typically will) have free variables that are bound outside of the query. Such free variables occur when a query is parameterized, for example, over values used in comparisons, predicates or whole sub-queries. Variables that occur free in a query are either let-bound or bound by a function outside of the query. At query compile time, the expressions to which these variables are bound have been evaluated to a value by the LINKS interpreter. Values are the result of the evaluation of every LINKS expression. To make the values for the query's free variables available, the interpreter passes the current environment to the query compiler when it encounters a query. The environment binds all free variables of the query.

$$
\begin{array}{rrcl}
(\textit{value}) & V & ::= & c \mid x \\
 & & \mid & V.l \mid (\overrightarrow{l = V}) \mid (\overrightarrow{l = V} \mid V) \\
 & & \mid & (V \setminus \vec{l}) \\
 & & \mid & C(V) \\
 & & & \\
(\textit{special}) & S & ::= & \mathsf{table} : (\mathsf{col}_1, \ldots, \mathsf{col}_n) \\
 & & \mid & \mathsf{query}\ \{M\} \mid \mathsf{query}\ [V, V]\ \{M\} \\
 & & & \\
(\textit{tail computation}) & T & ::= & \mathsf{return}\ V \\
 & & \mid & V(\vec{V}) \\
 & & \mid & \mathsf{case}\ V\ \mathsf{of}\ \overrightarrow{(C(x)) \rightarrow M} \\
 & & \mid & \mathsf{if}\ V\ \mathsf{then}\ M\ \mathsf{else}\ M \\
 & & \mid & \mathsf{special}\ S \\
 & & & \\
(\textit{binding}) & B & ::= & \mathsf{val}\ x = T \\
 & & \mid & \mathsf{fun}\ x(\vec{x}) = M \\
 & & & \\
(\textit{computation}) & M & ::= & \mathsf{let}\ \vec{B}\ \mathsf{in}\ T
\end{array}
$$

Figure 4.2.: Grammar of the ANF-based intermediate representation (IR).

$$
\begin{array}{rrcl}
(\textit{values}) & E & ::= & c \mid p \\
 & & \mid & \mathsf{table} : (\mathsf{col}_1, \ldots, \mathsf{col}_n) \\
 & & \mid & [\vec{E}] \\
 & & \mid & (\overrightarrow{l = E}) \\
 & & \mid & \mathsf{C}(E) \\
 & & \mid & Clos(\left\{\overrightarrow{x \rightarrow E}\right\}, \lambda\vec{x}.C) \\
 & & & \\
(variables) & x & & \\
(primitive functions) & p & &
\end{array}
$$

Figure 4.3.: Grammar of environment values.

Figure 4.3 shows a simplified [1] grammar of LINKS values. Note that besides data like records and lists it includes closures as the result of the evaluation of a function binding.

4.3.1. Query Representation

The combination of an IR AST and the value environment is transformed into a simpler, less structured language that combines all constructs from the IR and environment values: the query representation QR, which is identical to the language FL from Section 3.3. QR serves as input for the algebraic compilation and contains exactly those language constructs and functions which can be handled by the algebraic compilation. Figure 4.4 shows the complete QR grammar for reference.

$$
\begin{array}{rrcl}
(\textit{terms}) & T & ::= & \mathsf{let}\ x = T\ \mathsf{in}\ T \\
 & & | & \mathsf{if}\ T\ \mathsf{then}\ T\ \mathsf{else}\ T \\
 & & | & \mathsf{table}\ t : (\mathsf{col}_1, \ldots, \mathsf{col}_n) \\
 & & | & [T] \mid [] \mid \mathsf{append}(\vec{T}) \\
 & & | & x \mid c \\
 & & | & T \circ T \\
 & & | & P(\vec{T}) \\
 & & | & T.l \mid (\overrightarrow{l = T}) \mid (\overrightarrow{l = T} \mid T) \mid (T \setminus \vec{l}) \\
 & & | & () \\
 & & | & \mathsf{C}(T) \mid \mathsf{case}\ T\ \mathsf{of}\ \overrightarrow{(\mathsf{C}(x)) \rightarrow T} \\
 & & | & \lambda\vec{x}.T \mid (T\vec{T}) \\
(\textit{atomic constants}) & c & & \\
(\textit{variant constructors}) & \mathsf{C} & & \\
(\textit{variables}) & x & & \\
(\textit{operators}) & \circ & ::= & +, -, *, / \\
(\textit{primitive functions}) & P & ::= & \mathsf{take} \mid \mathsf{drop} \mid \mathsf{zip} \mid \mathsf{unzip} \mid \mathsf{length} \\
 & & | & \mathsf{sum} \mid \mathsf{select} \mid \mathsf{reverse} \mid \mathsf{or} \\
 & & | & \mathsf{and} \mid \mathsf{nubFlat} \mid \mathsf{concat} \\
 & & | & \mathsf{max} \mid \mathsf{min} \mid \mathsf{avg} \\
 & & | & \mathsf{takeWhile} \mid \mathsf{dropWhile} \mid \mathsf{groupByBase} \\
 & & | & \mathsf{concatMap} \mid \mathsf{map} \mid \mathsf{sortByFlat}
\end{array}
$$

Figure 4.4.: Grammar of the query representation (QR).

Although it would in principle be possible to compile the combination of an IR term and the value environment directly to algebra, there are multiple reasons why it is beneficial to transform to QR first: Firstly, due to the restrictions of ANF, transformations on the IR are inconvenient. For example, the standard β-reduction transformation is not closed on ANF forms and requires a renormalization step to keep the ANF property [21]. Such transformations are, however, a crucial step in the query compilation and should be possible without much extra effort. Additionally, the IR and environment value forms include semantically equivalent constructs like record creation and extension. For brevity, they are

[1] Actual values include amongst others XML values, and continuations.

here presented using the same syntax, though their actual internal representation is quite different. Merging IR and Value into a single form makes the compilation simpler and avoids having two redundant compilation rules for semantically equivalent constructs. Transforming environment entries into IR terms is not an option, because the IR does not include equivalents for some forms that occur in the environment (e.g. primitive functions and literal lists).

The transformation consists of syntax-driven functions $\mathcal{V}$ (values), $\mathcal{S}$ (special), $\mathcal{T}$ (tail computations), $\mathcal{B}$ (bindings) and $\mathcal{M}$ (computations) which transform the respective parts of the LINKS IR into equivalent QR terms. Another function $\mathcal{X}$ transforms environment values into QR terms. We omit the complete definition of the mostly straight forward transformation rules and mention only some aspects:

1. Before values from the environment are converted into the query representation, the environment is restricted to variables which actually occur free in the query. The environment might contain variables which are never used in the query. While this step reduces the amount of values that need to be converted and thus reduces the amount of work, it is crucial for another reason: The environment might contain values which can not be used in queries, e.g. recursive functions. The restriction of the environment guarantees that it contains only such values that can occur in queries and can be represented in QR and converted to it.

2. The restricted value environment is converted to a sequence of let-bindings, with a binding for every entry in the environment. A pair of a value environment and an IR term c is transformed as

$$(\{x_1 \to v_1, \ldots, x_n \to v_n\}, c) \quad \Rightarrow \quad \begin{array}{l} \mathsf{let}\ x_1 = \mathcal{E}[\![v_1]\!]\ \mathsf{in} \\ \vdots \\ \mathsf{let}\ x_n = \mathcal{E}[\![v_n]\!]\ \mathsf{in}\ \mathcal{C}[\![c]\!] \end{array}$$

   This transformation has the benefit that subsequent compilation stages do not need to handle the environment, just a single QR term.

   In the IR representation, primitive functions only occur in the form of variables that refer to the environment. The QR language only supports the application of primitive functions. To be able to handle primitive functions as first-class values (e.g. store them in data structures etc.), η-expansion is used to convert bindings to primitive functions in the environment into let-bindings to equivalent lambda abstractions.

$$\{\ldots, x \to p, \ldots\} \Rightarrow \begin{array}{l} \vdots \\ \mathsf{let}\ x = \lambda x_1 \ldots x_n.p(x_1, \ldots, x_n)\ \mathsf{in} \\ \vdots \end{array}$$

   where p is a primitive function and n is the arity of p. This conversion does not lead to overhead: If a primitive function is applied in the query, the binding to the corresponding η-expanded lambda abstraction is known and can be inlined by the optimization phase.

3. Function bindings that have already been evaluated outside of the closure are stored in the value environment in the form of closures $Clos(\Gamma_c, f)$: the actual function f is combined with a closure environment Γ_c which contains bindings for all free variables of the function. To match the QR representation of functions, closures are converted back into closed lambda abstractions. One possibility is to simply replace every occurence of a free variable in the body of the closure with the corresponding value from the closure environment. However, this would duplicate work if a free variable occurs multiple times. Instead, the closure environment is converted into a sequence of let-bindings in the body of the function which bind the individual environment entries. To keep the property that all variable names are unique, values from the environment must be renamed, because a variable name might occur in multiple closure environments. A closure record $Clos\left(\{x_1 \to v_1, \ldots, x_n \to v_n\}, \lambda y_1 \ldots y_n.b\right)$ is converted into a closed lambda abstration

$$\lambda y_1 \ldots y_n.\mathsf{let}\ x'_1 = \mathcal{V}[\![v_1]\!]\ \mathsf{in}\ \ldots\ \mathsf{let}\ x'_n = \mathcal{V}[\![v_n]\!]\ \mathsf{in}\ replace(\mathcal{C}[\![b]\!])$$

where $x'_1, \ldots, x'_n$ are unique, freshly chosen variable names and $replace$ is a function which replaces every variable x with its replacement x'. If a variable actually occurs only once in the body, it is inlined by the separate optimization phase (Section 4.3.2).

4. The LINKS IR does not include anonymous functions, just function bindings. Function bindings are transformed into let-bindings to anonymous functions:

$$\mathsf{let\ fun}\ f(x_1, \ldots, x_n) = b\ \mathsf{in}\ t \quad \Rightarrow \quad \mathsf{let}\ f = \lambda x_1 \ldots x_n.b\ \mathsf{in}\ t$$

5. The IR includes query annotations, which occur because of explicit annotations or table comprehensions (see Section 2.1). They signal to the normal LINKS interpreter that the annotated expression is to be handled by the query backend. The query compiler can just skip these annotations. Additionally, an annotation can include a *limiting* expression $[l, o]$. This means that the list-typed result of the query is limited to l elements starting at offset o. The Murrayfield query backend allowed these annotations only on the top-level query annotation and mapped them to the SQL construct **`LIMIT ... OFFSET ...`**. Instead, we also allow it to occur inside of queries and implement this feature using the take and drop functions: an annotation query $[l, o]$ $\{e\}$ is translated to take(l, drop(o, e)). As the type of queries is no longer restricted to lists of base records, a minor change to the typechecking of queries is necessary: it ensures that limiting is only used on list-typed queries.

4.3.2. Optimization

Query expressions are in general very suitable for optimizing rewrites: A type-correct query does not include explicit recursion and side-effects so that there are no non-terminating expressions and the order of evaluation of subexpressions does not matter. This means that rewrites like β-reduction are sound. QR expressions are optimized by applying partial-evaluation-style rewrite rules. Partial evaluation refers to the specialization of programs for which a part of the input is known at compile time [7]. Computations which only

depend on known values can be carried out during compilation. LINKS query compilation nicely matches this scheme: values that have been computed outside of the query are known, while values that depend on database tables are unknown at query compile time. Known values especially occur because LINKS allows the reuse of queries by abstraction: queries can be wrapped in functions that abstract over certain parts of the query. For instance, a predicate that is used to filter rows from a database table can be abstracted over (Listing 2.1 in Section 2.1).

At runtime, an invocation of the query compiler for a query that has been abstracted over means that the enclosing function has been applied. At that point, the arguments of the abstraction have been evaluated by the LINKS interpreter and the resulting values are available in the value environment. This means that these values are known in the query and computations which depend solely on these known values can be performed at (query) compile time. For instance, applications of abstracted predicates inside the query can be β-reduced.

The optimization pass implements a modest set of partial-evaluation reductions, the choice of which has been mainly inspired by [28]. It consists of three syntax-driven functions: the top-level function $\mathcal{Q}$ (Figure 4.5) recursively traverses a QR term. All let-bound values are added to environment Γ to be available for inlining. If a term is potentially reducible (e.g. a function application), the evaluation function $\mathcal{E}$ (Figure 4.6) is applied to the term after applying $\mathcal{Q}$ to all of its sub-terms. When $\mathcal{Q}$ encounters an application of a primitive function, function $\mathcal{P}$ (Figure 4.7) is applied to normalize certain primitive functions. With a bit more detail, the transformations performed by $\mathcal{Q}$, $\mathcal{E}$ and $\mathcal{P}$ are:

Inlining of small values $\mathcal{Q}$ inlines let-bound values. Inlining of values avoids the overhead of environment lookups. Recall from rule VARIABLE that the overhead for a variable lookup consists of an equijoin that aligns the iterations. However, care must be taken to avoid the duplication of computations. Therefore, let-bound values are only inlined (a) if they are simple in the sense that they do not involve a computation (i.e. constants and variables) or (b) if they are used only once. Additionally, lambda abstractions are always inlined to expose β-reductions (see below).

Partial evaluation The evaluation function $\mathcal{E}$ performs reduction of certain expressions over known values. if conditionals are reduced if the condition expression is a boolean constant. Similarly, case expressions are reduced if the value that is matched is a known value and projection and extension of known records are performed.

Aggressive function inlining The function $\mathcal{E}$ aggressively performs β-reduction or *function inlining* on every application of a known function. This step is crucial to get rid of the overhead of function application.

In usual compilers, the decision whether to inline a function is not an easy one. The benefit of the avoided overhead for the function call has to be weighed against the increase in code size if the function is applied more than once. This problem has been investigated extensively in compiler construction theory. In the algebraic compilation scheme used in this work, however, the decision is easy: due to specifics of the algebraic compilation, inlining is always worthwile.

The "calling" overhead in algebraic function application basically consists of the equijoin that constructs the iteration sub-context for a specific function/closure by

$$
\begin{aligned}
\mathcal{Q}\,[\![\mathsf{let}\ x = v\ \mathsf{in}\ c]\!]^{\Gamma} &= \mathcal{Q}\,[\![c]\!]^{\Gamma \cup \{x \to \mathcal{Q}[\![v]\!]^{\Gamma}\}} \\
\mathcal{Q}\,[\![\mathsf{if}\ c\ \mathsf{then}\ t\ \mathsf{else}\ e]\!]^{\Gamma} &= \mathcal{E}\left[\!\!\left[\mathsf{if}\ \mathcal{Q}\,[\![c]\!]^{\Gamma}\ \mathsf{then}\ \mathcal{Q}\,[\![t]\!]^{\Gamma}\ \mathsf{else}\ \mathcal{Q}\,[\![e]\!]^{\Gamma}\right]\!\!\right]^{\Gamma} \\
\mathcal{Q}\,[\![\mathsf{table}\ t : (\mathsf{col}_1, \ldots, \mathsf{col}_n)]\!]^{\Gamma} &= \mathsf{table}\ t : (\mathsf{col}_1, \ldots, \mathsf{col}_n) \\
\mathcal{Q}\,[\![[v]]\!]^{\Gamma} &= [\mathcal{Q}\,[\![v]\!]^{\Gamma}] \\
\mathcal{Q}\,[\![[]]\!]^{\Gamma} &= [] \\
\mathcal{Q}\,[\![\mathsf{append}(l_1, \ldots, l_n)]\!]^{\Gamma} &= \mathsf{append}(\mathcal{Q}\,[\![l_1]\!]^{\Gamma}, \ldots, \mathcal{Q}\,[\![l_n]\!]^{\Gamma}) \\
\mathcal{Q}\,[\![x]\!]^{\Gamma} &= \begin{cases} \Gamma[x] & x \in dom(\Gamma) \wedge inlineable(\Gamma[x]) \\ x \end{cases} \\
\mathcal{Q}\,[\![c]\!]^{\Gamma} &= c \\
\mathcal{Q}\,[\![()]\!]^{\Gamma} &= () \\
\mathcal{Q}\,[\![a \circledast b]\!]^{\Gamma} &= \mathcal{Q}\,[\![a]\!]^{\Gamma} \circledast \mathcal{Q}\,[\![b]\!]^{\Gamma} \\
\mathcal{Q}\,[\![P(v_1, \ldots, v_n)]\!]^{\Gamma} &= \mathcal{P}[\![P(\mathcal{Q}\,[\![v_1]\!]^{\Gamma}, \ldots, \mathcal{Q}\,[\![v_n]\!]^{\Gamma})]\!] \\
\mathcal{Q}\,[\![r.l]\!]^{\Gamma} &= \mathcal{E}\left[\!\!\left[\mathcal{Q}\,[\![r]\!]^{\Gamma}.l\right]\!\!\right]^{\Gamma} \\
\mathcal{Q}\,[\![(l_1 = v_1, \ldots, l_n = v_n)]\!]^{\Gamma} &= (l_1 = \mathcal{Q}\,[\![v_1]\!]^{\Gamma}, \ldots, l_n = \mathcal{Q}\,[\![v_n]\!]^{\Gamma}) \\
\mathcal{Q}\,[\![(l_1 = v_1, \ldots, l_n = v_n \mid r)]\!]^{\Gamma} &= \mathcal{E}\left[\!\!\left[(l_1 = \mathcal{Q}\,[\![v_1]\!]^{\Gamma}, \ldots, l_n = \mathcal{Q}\,[\![v_n]\!]^{\Gamma} \mid \mathcal{Q}\,[\![r]\!]^{\Gamma})\right]\!\!\right]^{\Gamma} \\
\mathcal{Q}\,[\![(r \backslash l_1, \ldots, l_n)]\!]^{\Gamma} &= \mathcal{E}\left[\!\!\left[(\mathcal{Q}\,[\![r]\!]^{\Gamma} \backslash l_1, \ldots, l_n)\right]\!\!\right]^{\Gamma} \\
\mathcal{Q}\,[\![\mathsf{C}(v)]\!]^{\Gamma} &= \mathsf{C}(\mathcal{Q}\,[\![v]\!]^{\Gamma}) \\
\mathcal{Q}\left[\!\!\left[\begin{array}{l}\mathsf{case}\ c\ \mathsf{of} \\ \\ \end{array}\begin{array}{l}\mathsf{C}_1(x_1) \to b_1, \\ \ldots, \\ \mathsf{C}_m(x_m) \to b_m\end{array}\right]\!\!\right]^{\Gamma} &= \mathcal{E}\left[\!\!\left[\begin{array}{l}\mathsf{case}\ \mathcal{Q}\,[\![c]\!]^{\Gamma}\ \mathsf{of} \\ \\ \end{array}\begin{array}{l}\mathsf{C}_1(x_1) \to \mathcal{Q}\,[\![b_1]\!]^{\Gamma}, \\ \ldots, \\ \mathsf{C}_m(x_m) \to \mathcal{Q}\,[\![b_m]\!]^{\Gamma}\end{array}\right]\!\!\right]^{\Gamma} \\
\mathcal{Q}\,[\![\lambda x_1 \ldots x_n.b]\!]^{\Gamma} &= \lambda x_1 \ldots x_n.\mathcal{Q}\,[\![b]\!]^{\Gamma} \\
\mathcal{Q}\,[\![(f\ (a_1, \ldots, a_n))]\!]^{\Gamma} &= \mathcal{E}\left[\!\!\left[\left(\mathcal{Q}\,[\![f]\!]^{\Gamma}\ \left(\mathcal{Q}\,[\![a_1]\!]^{\Gamma}, \ldots, \mathcal{Q}\,[\![a_n]\!]^{\Gamma}\right)\right)\right]\!\!\right]^{\Gamma}
\end{aligned}
$$

Figure 4.5.: Complete definition of the optimization function $\mathcal{Q}$. Function $dom(\Gamma)$ returns the set of variables bound in Γ and predicate $inlineable(v)$ is true iff either x occurs only once in the query or v is small or a lambda abstraction. $\Gamma[x]$ returns the value that is bound to variable x in Γ.

$$
\begin{aligned}
\mathcal{E}\,[\![(\lambda x_1 \ldots x_n.b\ (a_1,\ldots,a_n))]\!]^{\Gamma} &= \mathcal{Q}\,[\![b]\!]^{\Gamma\cup\{x_1\to a_1,\ldots,x_n\to a_n\}} \\
\mathcal{E}\,[\![(\mathsf{if}\ c\ \mathsf{then}\ t\ \mathsf{else}\ e)\ (a_1,\ldots,a_n)]\!]^{\Gamma} &= \begin{array}{ll} \mathsf{if}\ c & \mathsf{then}\ \mathcal{E}\,[\![(t\ (a_1,\ldots,a_n))]\!]^{\Gamma} \\ & \mathsf{else}\ \mathcal{E}\,[\![(e\ (a_1,\ldots,a_n))]\!]^{\Gamma} \end{array} \\
\mathcal{E}\left[\!\!\left[\left(\mathsf{case}\ c\ \mathsf{of}\ \begin{array}{l} \mathsf{C}_1(x_1)\to b_1, \\ \ldots, \\ \mathsf{C}_m(x_m)\to b_m \end{array} \quad (a_1,\ldots,a_n)\right)\right]\!\!\right]^{\Gamma} &= \mathsf{case}\ c\ \mathsf{of}\ \begin{array}{l} \mathsf{C}_1(x_1)\to \mathcal{E}\,[\![b_1(a_1,\ldots,a_n)]\!]^{\Gamma}, \\ \ldots, \\ \mathsf{C}_m(x_m)\to \mathcal{E}\,[\![b_m(a_1,\ldots,a_n)]\!]^{\Gamma} \end{array} \\
\mathcal{E}\,[\![(f\ (a_1,\ldots,a_n))]\!]^{\Gamma} &= (f\ (a_1,\ldots,a_n)) \\
\mathcal{E}\,[\![\mathsf{if}\ true\ \mathsf{then}\ t\ \mathsf{else}\ e]\!]^{\Gamma} &= t \\
\mathcal{E}\,[\![\mathsf{if}\ false\ \mathsf{then}\ t\ \mathsf{else}\ e]\!]^{\Gamma} &= e \\
\mathcal{E}\,[\![\mathsf{if}\ c\ \mathsf{then}\ t\ \mathsf{else}\ e]\!]^{\Gamma} &= \mathsf{if}\ c\ \mathsf{then}\ t\ \mathsf{else}\ e \\
\mathcal{E}\,[\![\mathsf{case}\ \mathsf{C}(v)\ \mathsf{of}\ \ldots,\mathsf{C}(x)\to b\ldots]\!]^{\Gamma} &= \mathcal{Q}\,[\![b]\!]^{\Gamma\cup\{x\to v\}} \\
\mathcal{E}\,[\![\mathsf{case}\ v\ \mathsf{of}\ \ldots,\mathsf{C}(x)\to b\ldots]\!]^{\Gamma} &= \mathsf{case}\ v\ \mathsf{of}\ \ldots,\mathsf{C}(x)\to b,\ldots \\
\mathcal{E}\,[\![(\ldots,l=v,\ldots).l]\!]^{\Gamma} &= v \\
\mathcal{E}\,[\![(l_1=v_1,\ldots,l_m=v_m\mid r)]\!]^{\Gamma} &= (l_1=v_1,\ldots,l_m=v_m\mid r) \\
\mathcal{E}\left[\!\!\left[\ \begin{array}{l} (l_1=v_1,\ldots,l_m=v_m \\ \mid (l_{m+1}=v_{m+1},\ldots,l_{m+n}=v_{m+n})) \end{array}\ \right]\!\!\right]^{\Gamma} &= \begin{array}{l} (l_1=v_1,\ldots,l_m=v_m, \\ l_{m+1}=v_{m+1},\ldots,l_{m+n}=v_{m+n}) \end{array}
\end{aligned}
$$

Figure 4.6.: Complete definition of the partial evaluation function $\mathcal{E}$.

an equijoin with the map plan. Additionally, if the function forms a closure, i.e. has free variables bound in the closure environment, an additional overhead is incured by the equijoins that lift the plans in the closure environment to the current iteration scope. Inlining of functions during the optimization phase avoids these overheads.

Generally, inlining leads to an increase in code size if a function is called multiple times. In algebraic compilation however, this increase can fundamentally not be avoided. Consider a term let $f = \lambda x.b$ in c, in which the function f is applied more than once in the term c. Due to the let-binding, the function is *known* in c. Assume that f was not inlined. During algebraic compilation, rule DEFLAMBDA would compile the lambda abstraction into the relational representation described in Section 3.3: a plan containing surrogate values and a closure record $\Gamma_c, map, \lambda x.b$, consisting of the (potentially empty) closure environment, the map plan and the abstraction itself. This representation would be bound to the environment. At the call sites of f, this representation would be retrieved from the environment and the application would be handled by rule APPLYCLOSURE. Recall that in rule APPLYCLOSURE, a function is essentially applied by binding the bound variables of the abstraction to the arguments in the environment and compiling the body, i.e. β-reduction. Thus, at every call site, the function body is inlined. The fundamental reason for this is that the table algebra – considered as a target *language* – has no notion of *calling* a function.

For a functional expression at a call site $f\ (a_1, \ldots, a_n)$, f might not be an expression which can be reduced to a literal lambda by evaluation. If f is a non-reducible case or if expression, function $\mathcal{E}$ pushes the application into the branches of the expression to expose potential β-reductions, e.g.

$$(\text{if } c \text{ then } t \text{ else } e)\ a \quad \Rightarrow \quad \text{if } c \text{ then } (t\ a) \text{ else } (e\ a)$$

This is, however, a dangerous transformation, as it duplicates the code of the arguments in every branch. It is therefore only performed when all arguments are "inlineable", according to the same criteria used when inlining variables.

Normalization of primitive applications Function $\mathcal{P}$ (Figure 4.7) rewrites applications of primitive functions. The primitive functions any, all and filter can be trivially expressed using the concatMap function. In the IR, list construction is done as usual in functional programming languages: the cons function constructs lists by attaching a value of type α to a list of type $[\alpha]$. Longer lists are constructed by nested sequences of cons applications. Single occurences and whole sequences of cons applications are rewritten into one application of the append operator, which appends and flattens a sequence of lists [2]. The reason for this rewrite lies in the algebraic construction of lists. Recall from rule APPEND (Section 3.1.10), that lists are appended with a disjoint union. After the union, new positions must be computed with the ϱ operator and new surrogate values are computed with the $\#$ operator. Rule APPEND performs the recomputation of positions and surrogate values exactly once, after all argument lists have been unionend. However, a compositional handling of the cons function

[2] Note that append is not a function, but an operator which takes an arbitrary number of arguments.

$$
\begin{aligned}
\mathcal{P}[\![\mathsf{any}(p,l)]\!] &= \mathsf{or}(\mathsf{concatMap}(p,l)) \\
\mathcal{P}[\![\mathsf{all}(p,l)]\!] &= \mathsf{and}(\mathsf{concatMap}(p,l)) \\
\mathcal{P}[\![\mathsf{filter}(p,l)]\!] &= \mathsf{concatMap}(\lambda x.\mathsf{if}\ p(x)\ \mathsf{then}\ [x]\ \mathsf{else}\ [],l) \\
\mathcal{P}[\![\mathsf{cons}(x,[])]\!] &= [x] \\
\mathcal{P}[\![\mathsf{cons}(x,l)]\!] &= \mathsf{append}([x],l) \\
\mathcal{P}[\![\mathsf{cons}(x_1,\mathsf{cons}(x_2,\ldots\mathsf{cons}(x_n,l)))]\!] &= \mathsf{append}([x_1],[x_n],\ldots,[x_n],l) \\
&\qquad \text{where } 1,\ldots,n \text{ is the longest prefix s.t. } l \neq cons(\ldots)
\end{aligned}
$$

Figure 4.7.: Definition of the normalization function $\mathcal{P}$.

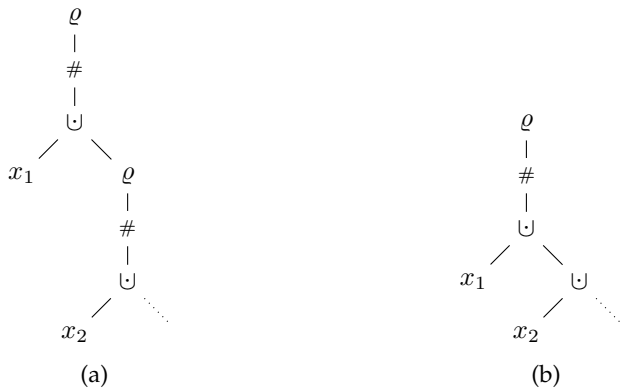


Figure 4.8.: Different schemes for algebraic list construction: $\mathsf{cons}(x_1,\mathsf{cons}(x_2,\ldots))$ (a) and $\mathsf{append}([x_1],[x_2],\ldots)$ (b).

would require to perform this recomputation after every cons in a sequence. Figure 4.8 illustrates the difference in the algebraic compilation of cons and append.

The rewrite transforms the first arguments of cons into singleton list arguments of the append constructor. Since the relational representation of a non-list value v and a singleton list $[v]$ are exactly the same, there is no overhead for those singleton lists.

The optimization phase is followed by a phase of dead-code elimination. It removes all bindings to variables which are not referenced. Especially, it removes bindings of variables which have been inlined by the preceding optimization phase. Dead code elimination is not strictly necessary: In the query plan generated by the algebraic compilation, expressions that are bound to a variable only show up if they are actually used (see rule VARIABLE, Section 3.1.10). However, the removal of unused bindings reduces the work necessary on subsequent compilation stages and makes the QR term more compact and thus more readable.

4.3.3. An Example

As an example for the optimized QR representation of queries, we come back to Query $Q3$ from Section 2.2. We do not show the IR and unoptimized QR representations of the query because of space restrictions. The optimization phase described in Section 4.3.2 results in the following rather compact QR representation:

```
concatMap(
  λt.
    [(team = t.1,
      maxps =
        concatMap(
          λpos.
            [(1 = y.1,
              2 = max(concatMap(λp1.[p1.eff], pos.2)))],
          groupByFlat(λp2.p2.pos, t.2)))]
  groupByFlat(λp3.p3.team, players))
```

Listing 4.2: Optimized representation of Query $Q3$.

The A-normalization of the source program leads to a large number of let-bindings. The optimizer removes all these bindings because the let-bound variables are only used once. Note that non-primitive functions (`eff`, `rosters`) have been eliminated by β-reduction, except for those which serve as arguments to primitive higher-order functions (`projTeam`, `projPos`). This representation serves as input for the boxing and algebraic compilation phase (Section 3).

4.4. Pathfinder

For the optimization of the generated table algebra plans and for SQL code generation, the query backend relies on the external Pathfinder component. Pathfinder has originally been developed to support the execution of XQuery queries on relational database systems. It uses the loop-lifting technique to compile XQuery to table algebra plans, optimizes them and generates database specific code (e.g. SQL:1999 or MonetDB's MIL), obtaining a highly efficient XQUERY execution platform on off-the-shelf RDBMS's.

The optimizations described in Section 4.3.2 operate on a language level to specialize QR expressions, reduce their size and eliminate overhead. The optimizer embedded in Pathfinder takes a rather different viewpoint: it employs rewrites on the level of algebra operators, taking information about key properties into account. Amongst others, a common subexpression elimination phase aggregates sub-plans that occur multiple times. Although we do not elaborate on the optimizations further here, the algebraic optimizer is actually a crucial component in the query compilation. Due to the compositionality of the loop-lifting compilation, the generated plans feature an exotic shape that the optimizers of current RDBMS's often can't cope with for more complex queries. Pathfinder's property inference and algebraic rewrites typically reduce the size of plans considerably and ideally change their shape into join graphs, a form RDBMS optimizers better cope with [13].

Pathfinder's code generator [14] is used to generate SQL queries that conform to the SQL:1999 standard from table algebra plans. It uses Common Table Expressions to implement the sharing of sub-plans that is common in the generated plans (see Section 3.1.7).

Some advanced SQL constructs are relied upon to provide implementations of table algebra operators: the SQL:1999 OLAP primitive DENSE_RANK() implements the rowrank operator ϱ and ROWNUM() implements the rownum operator $\#$.

As a downside, this means that MySQL and SQLite are not supported by the LINKS query backend anymore, because they do not implement the required SQL functionality. However, in principle, every SQL:1999-compliant database can be supported. The open-source RDBMS PostgreSQL supports all necessary SQL:1999 constructs and is currently supported in LINKS. Other examples are IBM's DB2 and the column store MonetDB.

4.5. Defunctionalization

As described in Section 3.3, first-class functions and higher-order functions have been implemented via algebraic closure conversion. The availability of variants (Section 3.2) makes another well-known approach to this problem feasible: *defunctionalization*. Defunctionalization allows to eliminate the higher-order aspect of programs at an earlier stage, so that the algebraic compilation does not have to deal with it. In this section, we describe defunctionalization in general, sketch how to use it in the context of the FERRY query backend, point to similarities between the algebraic closure and defunctionalization approaches and justify why the closure conversion approach was followed instead in this thesis.

Defunctionalization has first been described by Reynolds [25]. It is a whole-program transformation that turns higher-order programs into programs using only first-order values. The basic idea is to represent first-class functions with first-order data types, namely a variant type. Each function is mapped to a specific variant tag, which holds the values for the free variables of the function. Note that as different tags can tag values of different type, it is no problem for first-class functions to have different numbers of free variables with different types.

Applications of a first-class function are replaced with the applications of an apply function, which takes as arguments the variant representation of the function and the actual arguments. apply dispatches over the variant's tag and applies the code of the first-class function.

To illustrate the concept, we return to the example query in Section 3.3.6. A defunctionalized version of this query is shown in Listing 4.3. Both lambda abstractions in the body of the first comprehension have been replaced by applications of the variant tags A and B, both of which hold the free variable i. This first-order program could be handled by the FERRY query backend with variants and pattern matching alone, without the algebraic closure handling described in Section 3.3.

Reynolds described defunctionalization informally in an untyped setting. Bell et al. have noticed that the apply functions are in general not typable in a Hindley-Milner type system [2]. Although the query compiler does not need types, typability of the apply functions is nonetheless essential in the context of query compilation. Consider one unique apply-function which handles the application of abstractions of types $(Int) \to Int$ as well as $(Int) \to (Float)$. The case term in this apply-function would have multiple branches, of which one returns an Int value while another one returns a $Float$ value. When compiled algebraically, this would result in incompatible column layouts for the disjoint union

```
let fs = concatMap(λi.[A(i), B(i)], [10, 20, 30]) in
let apply = λfx.
              case f
                A(i) → x + i
                B(i) → x − i
in
  concatMap(λf.(apply (f, 42)), fs)
```

Listing 4.3: Defunctionalized version of the example in Section 3.3.6.

operator ($\uplus$) that merges the results of the indiviual case branches (see Section 3.2.8). As a consequence, it is necessary that all branches of a apply function return the same type (and have the same argument type). Informally, this is equivalent to typability of the apply functions.

A number of authors have explored defunctionalization in a typed setting [2, 23, 28]. The usual solution is to use multiple, type-specific apply functions, each of which handles first-class functions of one specific arrow type. This way, all branches of the case expression have the same type in an apply function. In the presence of polymorphism however, this leads to another problem. A polymorphic higher-order function that applies a functional argument can not decide with which type-specific apply function to replace the callsite. This makes it necessary to turn the program into an equivalent monomorphic version. Bell et al. describe an algorithm which monomorphizes the input program on the fly as part of defunctionalization [2]. Instead, in the following, we outline defunctionalization for the query compiler along the lines of a simpler algorithm by Tolmach et al. [28], which uses a separate monomorphization step.

1. Since the defunctionalization essentially relies on type information, the untyped QR representation can no longer be used. In principle, the typed IR representation could be used directly. However, for the same reasons given in Section 4.3.1, a typed version of the QR representation is used.

2. Just like closure application, the apply function introduces overhead, this time in the form of the case dispatch and the unboxing of the values for the free variables. For this reason, basically the same optimization phase as described in Section 4.3.2 is employed to eliminate as many function applications – especially applications of higher-order functions – by inlining.

3. The query is monomorphized by cloning every polymorphic function for every type it is applied at.

4. The monomorphic query is defunctionalized. To this end, a variant tag is assigned to every function definition and the corresponding type-specific apply function is equipped with an appropriate branch. A clone of the function is created and the function definition itself is replaced by the variant tag. The algorithm then leaves calls to known functions in place (although these should have been inlined anyway) and replaces applications of unknown functions with applications of the proper apply function. apply functions must be handled explicitly to primitive higher-order functions so that they can be applied in the primitive's implementation.

5. Defunctionalization is followed by another application of the optimization phase. This time, the main objective is to (partially) inline calls to the apply functions.

Defunctionalization and closure conversion come into play at various stages of the query compiler. On closer examination, however, the two approaches are structurally quite similar in the algebraic setting. Both approaches use surrogate values to track the flow of functions through the query. For the defunctionalization approach, the surrogate values are those refering to the inner plans which encode the tagged values. One difference is that in the case of the variant representation, the surrogates in the item_2 column are further qualified with the variant tag stored in the item_1 column, whereas for the closure approach, the equivalent for this qualification is the specific set of surrogates in the map plan. In both approaches, values for the free variables of abstractions (the closure environment) are stored in the form of query plans. For the defunctionalized variant representation, these plans are stored in the form of vs plans for tagged values and for the closure approach, the plans are stored in the closure environment. For both approaches, these plans compute the free variables for all iterations, when abstractions occur in the body of comprehensions.

At a callsite, the code for all abstractions which might be applied at this site is inlined (either in the form of the apply dispatch function or the function bodies from the closure records). The surrogate values with the additional qualification of tags and map plans respectively are used to reference the correct function to be applied in each iteration. Recall from Section 3.2.8 that in the presence of a default case, the selection of tags is performed in a stacked way to obtain the remaining iterations for the default case. However, for defunctionalization there is no default case and therefore all tags are selected from the same table. This roughly results in the same structure as for closure application, the only difference is that the dispatch happens with a tag *selection* instead of a map *join*.

We have argued that both approaches are structurally similar and differ more in implementation details. Reasoning abouth detail differences between the algebraic implementation of both approaches has proven to be rather pointless, because the structure of the algebraic code is heavily modified by the Pathfinder optimizer before the execution on a database and is further modified by the databases' query planner. At first sight, manually defunctionalized queries seem to compare favorably to ones using the algebraic closure approach with respect to running time. However, on closer inspection, this seems to be due to the Pathfinder optimizer which misses some optimization oportunities and can better cope with the defunctionalized plans. A further optimization of the closure plans by hand leaves no substantial difference between the plans for the two approaches. Given that an implementation of the defunctionalization approach needs considerably more effort than the closure approach and does not seem to bring substantial benefits, and that we assume that first-class functions are not used too often in queries, it seems reasonable to stay with the closure approach and further improve the Pathfinder optimizer.

5. Evaluation

The Murrayfield query translator seems to be able to compile the queryizable LINKS subset into SQL queries that are usually efficient and quite idiomatic. The examples of resulting SQL queries given in [5] roughly correspond to SQL queries that a programmer would write by hand. In contrast, the rather complicated algebraic compilation scheme described in Chapter 3 might raise suspicions as to whether the generated SQL code is efficient. A full-fledged assessment of the SQL code produced for different queries and a performance-wise comparison with the Murrayfield query translator is not in the scope of this thesis. However, in this chapter we give anecdotal evidence to support two claims:

1. The FERRY query translator typically generates idiomatic and efficient SQL queries.
2. The availability of nested results and access to database functionality like aggregate functions makes it possible to formulate queries which are considerably more efficient than equivalent Murrayfield programs that assemble nested results in the heap.

Figure 5.1a shows a simple query that is also supported by Murrayfield. The query performs a join between two tables, applies an additional predicate and sorts the result. With the help of the Pathfinder optimization and SQL code generation facilities, the FERRY query translator translates this query into the equivalent SQL query in Figure 5.1b. Apart from a column and table naming scheme that is specific to the algebraic compilation, the SQL query again is what a programmer would write manually.

For a more complex example that is not supported by Murrayfield, we turn again to Query $Q3$ (Section 2.2), whose algebraic compilation results in a plan bundle of four algebraic query plans. The left side of Figure 5.2 shows the outermost query plan, which computes the shape of the outermost list of the result. Because of space restrictions we omit the other unoptimized plans. They are however of similar shape and size. Because of the

```
for (p <-- players) where (p.eff > 10)
  orderby (p.name)
    for (t <-- teams) where (p.team == t.id)
      [(name = p.name, team = t.name)]
```

(a) Query $Q4$

```
SELECT 1 AS iter20_nat,
       a0001.name AS item19_str,
       a0000.name AS item16_str
  FROM players AS a0000,
       teams AS a0001
 WHERE 10 < a0000.eff
   AND a0000.team = a0001.id
ORDER BY a0000.name ASC,
       a0000.id ASC;
```

(b) SQL code for Query $Q4$

Figure 5.1.: A simple LINKS query and the SQL query generated from it by the FERRY query backend.

Figure 5.2.: Query plans resulting from the compilation of Query $Q3$: Unoptimized outermost plan P_1 (a), optimized plans $P_{1\cdot opt}$ - $P_{4\cdot opt}$ ((b)-(e)).

Data set size	Murrayfield backend		FERRY-based backend	
# teams	# queries	⏱ (sec)	# queries	⏱ (sec)
100	392	0.4	4	0.15
1000	3792	15.6	4	0.2
10000	37952	1199.3	4	0.8
100000	≈ 370000	DNF	4	7.2

Table 5.1.: Number of emitted queries and observed wall-clock execution times for Query $Q3$, average of 10 runs on pre-heated caches (DNF: did not finish within hours).

compositional nature of the compilation, the plan contains a considerable number of operators and has a irregular shape that bears no resemblance with the typical join-graph shape that results from SQL queries. The relational optimizer component of Pathfinder however manages to transform these plans into very compact plans, which are displayed on the right side of Figure 5.2. We omit the SQL queries that result from these optimized plans. It is however obvious that they can be transformed into equally compact SQL queries. Note that `max` is never applied to an empty list in this example, so that the error case never occurs. This is reflected in plan $P_{4 \cdot opt}$, which computes the tagged values for the Nothing tag and just consists of a literal empty table.

These two examples show that the new query backend *can* – and typically *will* – produce efficient and idiomatic SQL code. However, they also indicate that the efficiency of the produced SQL queries fundamentally relies on the performance of the Pathfinder optimization phase.

To support the second point, we compared the runtime of Query $Q3$ using the FERRY query translator on datasets of various sizes with the runtime of an equivalent LINKS program using the Murrayfield query translator. The Murrayfield version is shown in Appendix B. The benchmark was carried out on a MacBook Pro 6.2 (2.4 GHz Intel Core i5, 8 GB RAM) running Mac OS X 10.6 and using PostgreSQL 9. Data was generated according to the schema shown in Figure 2.4, using various numbers of teams $(10^2, 10^3, 10^4, 10^5)$. For each team, the number of players was drawn from a normal distribution with a mean of ten and a standard deviation of three and rounded. The position for each player was drawn uniformly distributed from a set of three positions.

Table 5.1 shows the running time for the different data sets as well as the number of SQL queries generated. As can be seen, the running time and number of queries of the Murrayfield version explode with increasing dataset size. The number of queries seems to raise linearly while the running time shows more of a quadratic behaviour. Meanwhile, the number of queries for the FERRY version is static and the runtime increases only modestly.

Part of the tremendous runtime of the Murrayfield version is due to a grossly inefficient implementation: The function `elem` uses a linear list scan, resulting in an asymptotic runtime of $\mathcal{O}(n^2)$ for the function `nub` that computes the distinct list of grouping keys. This implementation was chosen because the LINKS standard library does not contain a more efficient data structure. But even if a data structure with an efficient membership test were

used, the amount of SQL queries generated would still be prohibitive. Only small parts of the Murrayfield version are queryizable. The nested structure of the result has to be assembled by hand: Driven by an iteration over the distinct teams, the data for every single inner list is fetched with separate queries. This means that the number of queries raises linearly with the number of teams (and positions), an example of the *query avalanche* phenomenon mentionend before (Section 2.2).

In contrast, the FERRY version of the program computes the complete result in one query block, resulting in a static number of four SQL queries. It makes use of the database's implementation of `nub` (**`DISTINCT`**) and `max`. As the entire computation (apart from the assembly of the end result from the tabular results of the four queries) is performed on the database, the amount of data transferred from the database to the LINKS heap is substantially smaller.

6. Wrap Up

This section discusses related work (Section 6.1), concludes the thesis with a summary (Section 6.2) and points to possible future work (Section 6.2).

6.1. Related Work

The loop-lifting compilation technique has originally been developed to handle XQUERY'S FLOWR construct in the purely relational XQUERY compiler Pathfinder [11, 17]. Loop-lifting has since been generalized and extended to the FERRY compilation framework, with a richer data model and more language constructs to handle queries in more general programming languages. The FERRY framework has first been used in a stand-alone experimental language that centers around comprehensions and is designed to be executed completely on a relational database [15]. Besides this stand-alone language, FERRY forms the foundation of language-integrated query facilities for multiple host programming languages: Database Supported Haskell (DSH) is a library that executes parts of a HASKELL program on a relational database [10]. FERRY has also been used to construct an avalanche-safe provider for Microsoft's .NET LINQ framework [16], as well as a LINQ-based query facility for SCALA [9]. Finally, SWITCH, a seamless integration of relational queries into the RUBY language, is based on FERRY [12].

All these embeddings essentially operate with the same data model, which consists of atomic base values and arbitrary nested combinations of ordered lists and records or tuples. An exception is the FERRY LINQ provider, which also permits the querying of relationally encoded XML documents. Variant types are not supported and the queryizable subsets of the host languages are first-order. Compared to LINKS, the embeddings differ on grounds of safety: In DSH, explicit control over side-effects is not necessary, as Haskell is pure by default. Data types that are used in a query are restricted using Haskell's type class mechanism. The embedding technique of DSH, however, does not restrict the use of recursive or otherwise non-queryizable functions in queries. SWITCH – in line with RUBY'S untyped nature – does not provide static checks of queryizability. Also, the LINQ embeddings for .NET and SCALA only provide limited static checks for queryizability.

Giorgidze et al. have noticed that loop-lifting bears strong resemblance to Blellochs nested data parallelism in general and – in the context of Haskell – to Data Parallel Haskell (DPH) [19] in particular. They conjecture that loop-lifting can be formulated equivalently in terms of Blelloch's flattening transformation [10]. This similarity also arises in the additions to the loop-lifting framework that have been described in this thesis: variant types and first-class functions. To represent parallel arrays of binary variant types (or *sum types*), DPH uses two separate inner arrays for array elements from the left and right side of the sum type respectively, and a selector array which describes at which positions elements from the left and right type appear [3]. Generalizing this to n-ary variant types leads to a

representation that is very similar to the encoding of lists of variants described in Section 3.2: an outer array or table describes which tag is used at which position of the array or list and n inner arrays or tables contain the actual data for each tag. DPH's encoding of first-class functions is also quite similar to the one described in Section 3.3: DPH represents arrays of functions by a closure record. The closure environment contains whole arrays of assignments for the free variables of the function.

6.2. Conclusions

This thesis described the implementation of a FERRY-based query backend for the LINKS language. Concerning language-integrated queries, LINKS with its emphasis on safe queries through typing and seamless integration is certainly a step forward. However, we argued that the subset of the LINKS language that can be translated to SQL queries leaves much to be desired. The FERRY framework provides a compilation technique that makes it possible to extend the queryizable LINKS subset considerably. We applied this compilation technique in the context of LINKS. The new FERRY-based query backend more faithfully adheres to LINKS' list semantics, allows a query to have a larger set of types including nested lists and makes a substantially larger subset of the LINKS standard library available in queries. At the same time, the FERRY framework has proven to be flexible and powerful enough to incorporate more language features (variants and first-class functions). While making it possible to use tools in queries that functional programmers are used to, the new backend does not bring a loss in safety or in efficiency of generated SQL code. We demonstrated the considerable performance benefits obtained from the ability to evaluate complex queries completely on the database. Overall, we believe that the increased capabilities of the query backend move the query facility of LINKS a step further to true language integration, in which databases seamlessly participate in the evaluation of programs.

6.3. Future Work

Although the optimizations described in Section 4.3.2 are rather primitive, they succeed reasonably in eliminating obviously reducible expressions. For typical queries, these efforts seem to be sufficient, because simple queries offer only limited starting points for optimization. For more complex analytical queries however, it might be worthwile to investigate the suitability of more advanced source-level optimizations like Deforestation [29], Supercompilation [27] and more advanced partial evaluation techniques [7]. It might also be interesting to investigate if and how such optimization techniques are related to the algebra-level optimizations performed by Pathfinder.

Query compilation is performed at runtime because a query might have free variables, values for which are only available at runtime. For a closed query without free variables, this results in superfluous overhead. It should be straight forward to compile closed queries once at compile time, store the resulting SQL query bundle and at runtime only execute these SQL queries.

LINKS' syntax for list and table comprehensions provides syntactic sugar (`orderby`) to sort the results of the comprehension. It does however not provide syntactic sugar for

grouping as proposed by Wadler [20]. Since the Murrayfield query facility does not permit grouping, this was not a disadvantage in queries. The availability of grouping in the new query backend however might make it worthwile to implement a grouping syntax extension.

Appendix

A. Additional Compilation Rules and Supported Functions

$$\frac{\begin{array}{c}\Gamma; loop \vdash l \Rightarrow (q_l, cs_l, ts_l, vs_l, fs_l) \\ \Gamma; loop \vdash n \Rightarrow (q_n, cs_n, ts_n, vs_n, fs_n) \\ q \equiv \pi_{\mathsf{iter},\mathsf{pos},[cs_l]_<} \left(\sigma_{\mathsf{res}} \left(<_{\mathsf{res}:\langle \mathsf{pos},\mathsf{pos}' \rangle} \left(q_l \;\bar{\bar{\bowtie}}_{\mathsf{iter},\mathsf{iter}'} \left(\pi_{\mathsf{iter}':\mathsf{iter},\mathsf{pos}',\mathsf{item}_1} (q_n)\right)\right)\right)\right)\end{array}}{\Gamma; loop \vdash \mathsf{take}(n, l) \Rightarrow (q, cs_l, ts_l, vs_l, fs_l)} \quad (\textsc{Take})$$

Rule DROP works in exactly the same way as TAKE. The only difference is that the comparison between pos and pos′ is reversed.

$$\frac{\begin{array}{c}\Gamma; loop \vdash l \Rightarrow (q_l, cs_l, \emptyset, \emptyset, \emptyset) \\ q \equiv @_{\mathsf{pos}:1} \left(\left(\mathrm{GRP}_{\mathsf{item}_1:\mathrm{AND}(\mathsf{item}_1)/\mathsf{iter}} (q_l)\right) \cup \left(@_{\mathsf{item}_1:\mathsf{true}} (loop \setminus \pi_{\mathsf{iter}} (q_l))\right)\right)\end{array}}{\Gamma; loop \vdash \mathsf{and}(l) \Rightarrow (q, Col\ 1, \emptyset, \emptyset, \emptyset)} \quad (\textsc{And})$$

Rule OR works in extactly the same way as AND. The only difference is that the aggregate OR is used and that the default value for empty lists is false instead of true.

$$\frac{\begin{array}{c}\Gamma; loop \vdash l \Rightarrow (q_l, cs_l, ts_l, vs_l, fs_l) \\ q_e \equiv @_{\mathsf{pos}:1} \left(@_{\mathsf{item}_1:true} (loop \setminus q_l)\right) \\ q_{ne} \equiv @_{\mathsf{pos}:1} \left(@_{\mathsf{item}_1:true} (\delta (\pi_{\mathsf{iter}} (q_l)))\right)\end{array}}{\Gamma; loop \vdash \mathsf{empty}(l) \Rightarrow (q_e \cup q_{ne}, Col\ 1, \emptyset, \emptyset, \emptyset)} \quad (\textsc{Empty})$$

$$\frac{\begin{array}{c}\Gamma; loop \vdash l \Rightarrow (q_l, cs_l, \{\mathsf{item}_1 \to (q_1, cs_1, ts_1, vs_1, fs_1)\}, \emptyset, \emptyset) \\ q \equiv \pi_{\mathsf{iter},\mathsf{pos}:\mathsf{pos}',[cs_1]_<} \left(\varrho_{\mathsf{pos}'':\langle \mathsf{pos},\mathsf{pos}' \rangle} \left((\pi_{\mathsf{surr}:\mathsf{item}_1,\mathsf{iter},\mathsf{pos}} (q_l))\right) \bar{\bar{\bowtie}}_{\mathsf{surr},\mathsf{iter}'} \left(\pi_{\mathsf{iter}':\mathsf{iter},\mathsf{pos}':\mathsf{pos},[cs_1]_<} (q_1)\right)\right)\end{array}}{\Gamma; loop \vdash \mathsf{concat}(l) \Rightarrow (q, cs_1, ts_1, vs_1, fs_1)} \quad (\textsc{Concat})$$

Category	Functions/Operators
Arithmetic	`+ - * / ^ mod +. -. *. /. ^.`
Logical	`&& || not`
Comparison	`== < > <= >= !=`
Math	`floor ceiling cos sin tan log sqrt`
String	`~ ^^ strsub strlen`
Conversion	`intToString stringToInt intToFloat floatToInt FloatToString stringToFloat`
List	`concat length take drop max min zip unzip reverse`
Aggregate	`max min and or sum avg`
Higher-order	`concatMap map mapi filter sortByFlat groupByFlat all any takeWhile dropWhile`

Table A.1.: Primitive functions and recursive functions from the LINKS prelude that can be used in queries.

B. Query $Q3$ (Murrayfield)

```
fun nub(l) {
  fun aux(d, x) {
    if (elem(x, d)) d else x :: d
  }
  reverse(fold_left(aux, [], l))
}

fun groupBy(project, t) {
  var keys = nub(for (p <-- t) [(k = project(p))]);
  for (key <- keys)
    var foo = for (p <-- t) where (project(p) == key.k) [p];
    [(key.k, foo)]
}

fun projTeam(p) { p.team }
fun positions(ps) { for (p <- ps) [p.team
fun eff(ps) { for (p <- ps) [p.eff] }

fun fetchByTeamPos(team, pos) {
  for (p <-- players_table) where (p.team == team && p.pos == pos)
    [p]
}

for (r <- groupBy(projTeam, players_table)) {
  var posn = for (pos <- positions)
               [(pos, fetchByTeamPos(r.1, pos)];
  var maxps = for (pos <- posn) [(pos.1, max(eff(pos.2)))];
  [(team = r.1, maxps = maxps)]
}
```

Listing B.1: Murrayfield version of Query $Q4$.

List of Tables

List of Figures

Listings

Bibliography

[1] Andrew W. Appel and Trevor Jim. Shrinking lambda Expressions in Linear Time. *Journal of Functional Programming*, 7(5):515–540, September 1997.

[2] Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-Driven Defunctionalization. In *Proc. ICFP*, pages 25–37. ACM, September 1997.

[3] Manuel M. T. Chakravarty and Gabriele Keller. More Types for Nested Data Parallel Programming. In *Proc. ICFP*, pages 94–105. ACM, September 2000.

[4] Ezra Cooper. *Programming Language Features for Web Application Development*. PhD thesis, University of Edinburgh, 2009.

[5] Ezra Cooper. The Script-Writer's Dream: How to Write Great SQL in Your Own Language, and Be Sure It Will Succeed. In *Proc. DBPL*, pages 36–51. Springer, August 2009.

[6] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web Programming Without Tiers. In *Proc. FMCO*, pages 266–296. Springer, November 2006.

[7] Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Partial Evaluation*. Springer, 1996.

[8] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The Essence of Compiling with Continuations. In *Proc. PLDI*, pages 237–247, New York, NY, USA, June 1993. ACM.

[9] Miguel Garcia, Anastasia Izmaylova, and Sibylle Schupp. Extending Scala with Database Query Capability. *Journal of Object Technology*, 9(4):45–68, July 2010.

[10] George Giorgidze, Torsten Grust, Tom Schreiber, and Jeroen Weijers. A Database Coprocessor for Haskell. In *Proc. IFL*. Springer, June 2010.

[11] Torsten Grust. Purely Relational FLWORs. In *Proc. XIME-P*, June 2005.

[12] Torsten Grust and Manuel Mayr. A Deep Embedding of Queries into Ruby, 2011. Submitted for publication.

[13] Torsten Grust, Manuel Mayr, and Jan Rittinger. Let SQL Drive the XQuery Workhorse (XQuery Join Graph Isolation). In *Proc. EDBT*, pages 147–158. ACM, March 2010.

[14] Torsten Grust, Manuel Mayr, Jan Rittinger, Sherif Sakr, and Jens Teubner. A SQL: 1999 Code Generator for the Pathfinder XQuery Compiler. In *Proc. SIGMOD*, pages 1162–1164. ACM, June 2007.

[15] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. FERRY: Database-Supported Program Execution. In *Proc. SIGMOD*, pages 1063–1066. ACM, June 2009.

[16] Torsten Grust, Jan Rittinger, and Tom Schreiber. Avalanche-Safe LINQ Compilation. *PVLDB*, 3(1):162–172, 2010.

[17] Torsten Grust, Sherif Sakr, and Jens Teubner. XQuery on SQL Hosts. In *Proc. VLDB*, pages 252–263. Morgan Kaufmann, September 2004.

[18] Torsten Grust and Jens Teubner. Relational Algebra: Mother Tongue - XQuery: Fluent. In *Proc. Twente Data Management Workshop (TDM)*, pages 9–16. University of Twente, 2004.

[19] Simon L. Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *Proc. FSTTCS*, pages 383–414, 2008.

[20] Simon L. Peyton Jones and Philip Wadler. Comprehensive Comprehensions. In *Proc. Haskell Workshop*, pages 61–72. ACM, September 2007.

[21] Andrew Kennedy. Compiling with Continuations, Continued. In *Proc. ICFP*, pages 177–190. ACM, October 2007.

[22] Sam Lindley and Philip Wadler. The Audacity of Hope: Thoughts on Reclaiming the Database Dream. In *Proc. ESOP*. Springer, March 2010.

[23] François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization. In *Proc. POPL*, pages 89–98. ACM, January 2004.

[24] François Pottier and Didier Remy. The Essence of ML Type Inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.

[25] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proc. ACM*, pages 717–740, New York, NY, USA, 1972. ACM.

[26] Jan Rittinger. *Constructing a Relational Query Optimizer for Non-Relational Languages*. PhD thesis, Universität Tübingen, 2010.

[27] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[28] Andrew P. Tolmach and Dino Oliva. From ML to Ada: Strongly-Typed Language Interoperability via Source Translation. *Journal of Functional Programming*, 8(4):367–412, 1998.

[29] Philip Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, 73(2):231–248, 1990.

[30] Philip Wadler and Simon L. Peyton Jones. How to Make ad-hoc Polymorphism Less ad-hoc. In *Proc. POPL*, pages 60–76. ACM, 1989.