The Dodo Query Flattening System

Joeri van Ruth Maarten Fokkinga Maurice van Keulen

September 2004

Contents

1 Introduction **2**
1.1 Motivation 2
1.2 Overview 4
1.3 Differences with Moa 11

2 Data Types **13**
2.1 Type Formers 13
2.2 Algebras, monads and comprehensions 15
2.3 Application in Dodo 21

3 Language **25**
3.1 Column type system 25
3.2 Syntax 27
3.3 To point-free form 30

4 The prototype **33**
4.1 Column operators 33
4.2 Core frames and rewrite rules 37
4.3 Lists, bags, etc. 38
4.4 Rewriting example 43

5 Issues **44**
5.1 General Remarks 44
5.2 Future Work 48

Abstract

Analytical query processing over complex objects often suffers from disappointing performance due to excessive use of nested-loop (element at a time) evaluation. Storing the data in a flattened form enables collection based processing (set at a time), gaining performance at the cost of having to write more complicated queries. This report proposes Dodo, an approach to automatic translation of queries from the complex objects domain into set-at-a-time operations against data stored in a flattened form.

Chapter 1

Introduction

1 Goal. The basic question we try to answer in this report is: what is a natural way of translating expressions from a language based on the typed λ-calculus into a language based on binary relational algebras. The source language has a type system based on type formers such as "line segment", or "list of x", which can be arbitrarily nested, whereas the latter is based purely on binary relations between atomic types. The main purpose of the current report is to establish a systematic mapping from queries over the complex data model to queries based on bulk operations over binary relations. Optimization issues and notational conveniences are mostly postponed to another report.

2 Document structure. In the current chapter we give an informal introduction to what we try to achieve and how the Dodo approach works. We also summarize the differences with Moa, a predecessor system. Chapter 2 is more formal. There we first describe the type system of the high-level part of the language. Then we introduce several concepts from category theory in the context of that type system. In particular we show how the notion of a *catamorphism* allows us to unify the treatment of structured data types, aggregate functions and a convenient comprehension notation. In the final section of chapter 2 we look at the complications the two-layered Dodo system gives for the catamorphism-based approach. In chapter 3 we give the syntax of our initial Dodo language and a detailed account of how expressions are translated from their initial, value-at-a-time representation into hopefully more efficient set-at-a-time counterparts. This chapter is rather general. Chapter 4 pins down a couple of basic data structures and operations and gives the specific rewrite rules using which these can be translated into the language of binary relations which is the target of the Dodo rewriter. Finally, chapter 5 collects a number of issues that were skipped in earlier chapters or which constitute future work.

1.1 Motivation

3 Complex data models. Non-traditional application domains with need for complex data models are not well served by traditional relational database management systems. Examples of such application domains are geographical information systems and several kinds of multimedia applications. These applications involve large, often huge amounts of data with a high degree of nesting. Updates tend to be infrequent whereas the queries tend to be complex and usually aggregate or transform attributes of large numbers of entities. Current relational databases are typically optimized towards multitudes of simple queries that retrieve and update a small number of entities each (Transaction Processing).

The data models used by these applications are far away from the data model provided by the databases. Moreover, the monolithic nature of these databases makes it hard to pick and choose functionality: non-trivial extensions typically end up implementing their own storage management,

access methods and other facilities that ought to be the responsibility of the DBMS. Not only do these independent reinventions represent a lot of duplicate effort, they usually also lack data independence: there is no decoupling of the applications algorithms from the storage layout and the retrieval methods supported by it. Apart from the flexibility gained from using an intermediate abstraction layer, there are also performance reasons: the implementation of the domain specific algorithms tends to be very much tied to the implemented data access methods, which makes them almost impossible for a query optimizer to get a grip on. In contrast, the relational model, even its butchered implementation in existing SQL systems, both insulates the application from low-level issues of storage management and provides the optimizer with a small set of well-defined primitive operators for which many specialized implementations are known.

4 Nested loop processing. One particular pitfall in querying complex data structures is to get stuck in nested loop processing, which means a quick alternation between retrieving an object, doing a small amount of processing, retrieving a related object, etc. It is almost impossible to achieve any reasonable performance this way, because one ends up bound by the latency of the object retrieval phase. Especially on modern hardware, memory latency is extremely poor relative to the speed of the CPU. It is essential to organize the data in such a way that operations are performed on large numbers of items at the same time, preferably items that are stored consecutively to guarantee predictable access patterns.

The cost of branch mispredictions and cache misses has become so large that the only way to get modern hardware to approach its theoretical performance limits is by taking algorithms with extremely simple and predictable inner loops and tuning them to the cache hierarchy [9]. The cache hierarchy consists of the L1 cache at the top, with an access latency of only a couple of cycles, through other cache levels, main memory with a latency of hunderds to thousands of cycles, to disk storage with a latency that is measured in milliseconds, i.e., millions of cycles. It used to be sufficient to simply minimize the number of disk accesses, employing sometimes rather complicated algorithms to do so, but nowadays one must really take into account the properties of the L1 and L2 cache. This leads to simpler algorithms. In fact, in a surprisingly large number of cases, a sequential scan over an array outperforms more sophisticated algorithms for the simple reason that its memory access pattern is so predictable [4]. As Terje Mathisen of `comp.arch` fame puts it in his signature,

> *Almost all programming can be viewed as an exercise in caching.*

5 Data decomposition. With analytical processing in particular it is important to avoid nested loop processing. As indicated above, this implies expressing the query using relatively many bulk operations that each perform a simple action on a large number of items. The usual way to do this is to bring the data in 3NF or other relational normal form. One can take this a step further and split every so obtained n-ary relation R into $n - 1$ binary relations between the primary key of R and one of its attributes. This is called “vertical decomposition.” It is often possible to arrange for the key attributes to be a range of consecutive integers. In that case, only the value attributes need to be stored, possibly in an array-like structure with the keys as array indices. Clustering by attribute rather than by entity allows operations that access only one or two attributes of many entities to avoid wasting cache space and memory bandwidth on attributes they are not going to use. This kind of query is very common in analytical query processing.

6 Semantic gap. On the one hand, applications require complex data models with complex operations on this data; on the other hand, in order to get performance the data needs to be carefully decomposed and accessed using simple bulk operations. The idea behind Moa and now Dodo is to support rich data models with nested data structures on top of a decomposed storage abstraction, automatically translating queries from the nested domain to the flattened domain as needed. The automatic translation insulates the user from the details of the decomposition and the size and complexity of the queries, thus providing a degree of data independence. Another benefit of this approach is that new data structures are no longer byte sequences stored by, but

otherwise opaque to the underlying DBMS. Instead, new data structures are defined in terms of building blocks understood by the DBMS, in particular its optimizer. This can have massive benefits for both extensibility and performance [ref dexapaper].

7 MonetDB. MonetDB [4, 1, 3, 2] is an extensible database kernel based on binary relations. In MonetDB, binary relations are implemented as BATs, Binary Association Tables. For the reasons outlined in paragraph 4, a BAT is a region of memory containing consecutive $(value_1, value_2)$-pairs together with some metadata. In principle, BATs are operated upon in main memory, using cache-aware algorithms whenever possible. If not enough memory is available, MonetDB uses virtual memory. When available, MonetDB uses `madvice` to advice the virtual memory subsystem of its intentions.

MonetDB can readily be extended with new elementary types, new BAT operations and new search accelerator. A search accelerator is an auxiliary data structure that does not change the semantics of a BAT, but may speed up certain operations on it. We call MonetDB an extensible database kernel rather than a database management system because, even though it provides most additional features, MonetDB is more intended as a toolbox than a complete system. For instance, it does provide a lock manager but it can easily do without. The focus on vertically decomposed storage, combined with its general toolbox approach make MonetDB particularly well-suited for the approach outlined in this work.

8 Moa and Dodo. BATs are suitable as building blocks for larger data structures such as relations as used by the SQL front-end and document trees in the XQuery frontend [7]. The Moa system started as a general front-end to MonetDB in the MAGNUM GIS project [2, 12]. It allows the user (the extension writer, to be precise) to define new, nested data types together with a way of mapping them to BAT storage. Operations on the nested structures are translated to BAT operations. In later projects, Moa has been applied to multimedia retrieval [5, 10]. In the SUMMER project [13] Moa has served as an intermediate language for querying (distributed) SQL databases using a subset of XQuery.

Dodo can be regarded as a theoretical clean-up of Moa. It is an attempt to give Moa a theoretical foundation using a categorical theory of data types similar to [8]. Such a foundation is necessary for every attempt to reliably bridge the semantic gap referred to in paragraph 6. A bridge between complex data model and decomposed storage needs to be sufficiently abstract to provide data independence, it needs to be extensible at both the nested and the flattened level, and it needs express the nested query using bulk operations in order to achieve performance. For all these concerns, a clean mathematical foundation is a huge benefit.

1.2 Overview

This section gives a high level overview of the Dodo approach. We start with the way the nested data structures are mapped to binary relations, then explain what happens when a query is processed by Dodo.

9 Flattening data. The Dodo system evaluates queries over structured data which has been decomposed into binary relations. A "binary relation" is set of pairs. Note, however, paragraph 108. For historical reasons, we often use the word "column" for binary relations used in Dodo.

How the "flattening" of data structures into columns is achieved is best understood by means of an example. Consider the following bag of bags of strings

$$B = \{\!\!\{\{\!\!\{\text{fido}\}\!\!\}, \{\!\!\{\text{spot}, \text{rex}\}\!\!\}, \{\!\!\{\}\!\!\}\}\!\!\}.$$

Start by giving every part of the expression a unique identifier.

$$B = \{\!\!\{\{\!\!\{\text{fido}_1\}\!\!\}_{10}, \{\!\!\{\text{spot}_7, \text{rex}_8\}\!\!\}_{20}, \{\!\!\{\}\!\!\}_{30}\}\!\!\}_{100}.$$

There is one outer bag, three inner bags and three elements. We store their identifiers as three binary *identity* relations d_1, d_2 and d_3:

d_1	
100	100

d_2	
10	10
20	20
30	30

d_3	
1	1
7	7
8	8

By a binary identity relation we mean a binary relation consisting only of (x, x)-pairs. Reasons for using exactly this representation for sets are discussed in paragraph 12. The relation between outer bag and inner bags is given by r_1; the relation between inner bags and elements is given by r_2; the relation between element identifiers and dog names is given by f as follows:

r_1	
100	10
100	20
100	30

r_2	
10	1
20	7
20	8

f	
1	*fido*
7	*spot*
8	*rex*

Notice that 30, the identifier of the empty inner set, does occur in r_1 but not in r_2.

10 Frames. To express the way the columns model, in this case, a bag of bags of strings, the columns are arranged in *frames*:

$$F = bag\langle d_1, r_1, bag\langle d_2, r_2, atom\langle f\rangle\rangle\rangle. \tag{1.1}$$

A frame consists of a frame name, which identifies its type, together with zero or more columns and subframes. The bag frame corresponds to the "bag of" type. The $atom\langle\rangle$ frame corresponds in this case to the "string" type. It needs a single column to store its mapping from keys to dog names. It is very important to realise that a frame represents not one, but a collection of values, each value identified by an identifier that is unique within the frame. In other words, a frame represents a *function* from its domain to its corresponding type. Exactly how the value for a given key k is constructed depends on the frame type. With $atom\langle f\rangle$, the key is simply looked up in column f; for $bag\langle d, r, F\rangle$, first the relational image $\{k' \mid (k, k') \in r\}$ is constructed, and then the resulting elements keys are looked up in F. In paragraph 19, such a "lookup procedure" is called the *interpretation function* of the frame.

11 Explicit domains. The careful reader may have noticed that d_3 is not mentioned in equation (1.1). It is important to understand why d_3 does not occur there while d_1 and d_2 do. A frame represents a function. A function has a domain. As explained in section 73 the rewrite process needs a way of constructing the domain of a frame for each frame type. The domain of the $atom\langle f\rangle$ frame (d_3) can easily be obtained from f. However, the domain of $bag\langle d_2, r_2, atom\langle f\rangle\rangle$ cannot be obtained from r_2 and f alone: the empty bag with key 30 does not show in either r_2 or f. That is why bags carry along their domain explicitly in the form of a column d.

12 Everything a binary relation. The choice to represent "index sets" such as d_1, d_2 and d_3 as binary index relations rather than unary relations was made because it allows many operations to be expressed using relational composition. The relational composition operator $*$ is defined by

$$r * r' = \{(x, z) \mid (x, y) \in r \land (y, z) \in r'\}.$$

If we additionally define $twin(r) = \{(x, x) \mid (x, y) \in r\}$ and $rtwin(r) = \{(y, y) \mid (x, y) \in r\}$, intersection, relational restriction and relational image can all be expressed using these three operations. With $d_1' = \{100\}$, $d_2' = \{10, 20, 30\}$ and $d_3' = \{1, 7, 8\}$ as the unary counterparts of d_1,

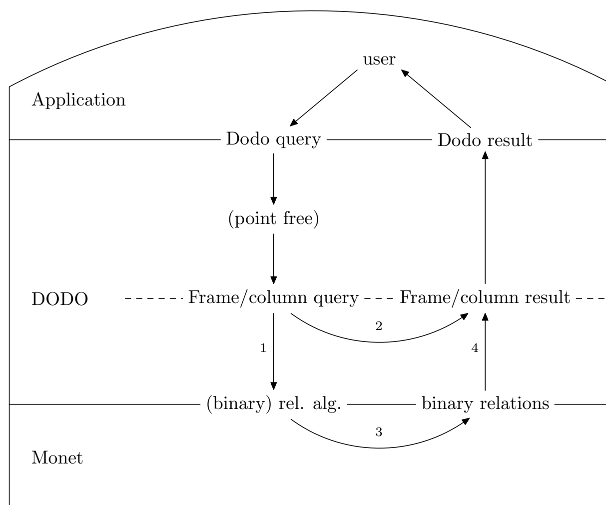

Figure 1.1: Flow of information in the DODO system. Horizontal lines denote boundaries between components. The Dodo query and result languages form the interface between Dodo and the application. The target database, is addressed using its own query language and returns data in the form of binary relations. The dashed line within the DODO component separates the target-independent and the target-dependent parts of DODO.

d_2 and d_3, the following are equivalent:

"intersection"	$d'_1 \cap d'_2$	and	$d_1 * d_2$
"domain restriction"	$d'_1 \lhd r$	and	$d_1 * r$
"range restriction"	$r \rhd d'_2$	and	$r * d_2$
"relational image"	$r[d'_1]$	and	$rtwin(d_1 * r)$
"range"	$\mathrm{ran}(r)$	and	$rtwin(r)$
"domain"	$\mathrm{dom}(r)$	and	$twin(r)$

In section 3.2 we generalize the composition operator, allowing it to under some circumstances take frames rather than columns as a right hand argument. Every frame type provides a rewrite rule to implement this. For instance, the *atom* frame implements $*$ as

$$r * atom\langle f \rangle = atom\langle r * f \rangle.$$

If we implemented the above operations directly rather than using column operations and $*$, new frame types would need to define more rewrite rules.

13 Flow of processing. The general flow of information in Dodo is depicted in figure 1.1. In the next several paragraphs we follow a query as it flows through the system.

Consider a system which stores information about dogs. The underlying database defines d_1, d_2, d_3, r_1, r_2 and f as in (1.1), and also $g = \{1 \mapsto 3, 7 \mapsto 1, 8 \mapsto 9\}$, a column of dog ages. However, instead of 100, the key for the outermost bag in equation (1.1) is the special singleton key †. The key † is the element of the unit type $\mathbf{1} = \{\dagger\}$. This type is used throughout Dodo to emphasize that there is exactly one instance of a given entity. The revised columns can be found in figure 1.2.

The cycle starts when an application poses a query to the Dodo system on behalve of the user. The interface between Dodo and the application consists of the *Dodo query language* for posing the query and the *Dodo result language* for retrieving the result. The degree to which any of these are visible to the user depends on the application.

d_1	
†	†

d_2	
10	10
20	20
30	30

d_3	
1	1
7	7
8	8

r_1	
†	10
†	20
†	30

r_2	
10	1
20	7
20	8

f	
1	*fido*
7	*spot*
8	*rex*

g	
1	3
7	1
8	9

Figure 1.2: Example database. The unit key † identifies the database as a whole. The keys 10, 20 and 30 can be considered to identify owners, although the example does not depend on this interpretation. The keys 1, 7 and 8 identify dogs. This database is redundant: d_1 is a synonym for *idunit*, see section 82. Likewise, d_2 is an abbreviation for $rtwin(r_1)$. Alternatively, $r_1 = sethead(d_2, \dagger)$.

Suppose the user asks the application for a bag containing a (*name*, *age*) pair for every dog in the system. Given the data in figure 1.2, Dodo should return the bag

$$\{\!|(fido, 3), (spot, 1), (rex, 9)|\!\}. \tag{1.2}$$

The result frame for this query re-uses keys for multiple parts of the expression, which are disambiguated by their syntactical position: one is the left hand of a bag member, one is the right hand of a bag member, and one is the bag member itself:

$$\{\!|(\mathrm{fido}_1, 3_1)_1, (\mathrm{spot}_7, 1_7)_7, (\mathrm{rex}_8, 9_8)_8|\!\}_\dagger. \tag{1.3}$$

Note that the wording "a collection of values, each value identified by a key *unique within the frame*" in paragraph 9 allows this re-use of keys. Assuming the existence of $idunit := \{\dagger \mapsto \dagger\}$ and $sethead(r, v) := \{(v, y) \mid (x, y) \in r\}$, the following is the corresponding frame for bag (1.3):

$$bag\langle idunit, sethead(d_3, \dagger), pair\langle atom\langle f\rangle, atom\langle g\rangle\rangle\rangle. \tag{1.4}$$

This frame forms a bag by taking d_3, which represents the dog keys in the system, and turning it into a relation $sethead(d_3, \dagger) = \{(\dagger, 1), (\dagger, 7), (\dagger, 8)\}$. The dog keys are looked up in the pair frame, which in turn looks them up in the two atom frames.

14 Schema. To pose a query in the Dodo language, we first need a schema that defines the entities the query can refer to. A Dodo schema can be regarded as an "external view" on the database. Many such schemas can co-exist at the same time, combining columns into frames in different ways. We assume that the database administrator has provided the following schema:

$$\begin{array}{rcl} nd = nesteddogs : \mathsf{Bag\ Bag}\ Dog & := & (bag\langle d_1, r_1, bag\langle d_2, r_2, atom\langle d_3\rangle\rangle\rangle)\ \dagger \\ & = & \{\dagger \mapsto \{\!|\{\!|1_1|\!\}_{10}, \{\!|7_7, 8_8|\!\}_{20}, \{\!||\!\}_{30}|\!\}_\dagger\}; \\ dn = dogname : Dog \to String & := & atom\langle f\rangle \\ & = & \{1 \mapsto \mathrm{fido}, 7 \mapsto \mathrm{spot}, 8 \mapsto \mathrm{rex}\}; \\ da = dogage : Dog \to Int & := & atom\langle g\rangle \\ & = & \{1 \mapsto 3, 7 \mapsto 1, 8 \mapsto 9\}. \end{array} \tag{1.5}$$

Note that the $atom\langle\rangle$ frame in the first definiton depends on the fact that d_3 is a binary identity relation. We assume that the schema also specifies that all known dogs do occur somewhere in *nesteddogs*. This schema is intentionally clumsy in that it stores the dog identifiers in a bag of bags, rather than just in a single bag. Later on we shall see how Dodo eliminates this nesting.

15 Example Query. In the Dodo query language, the simplest query to retrieve the (*name*, *age*)-pairs from the given schema is

$$Q = Bag[(dn\ \ d, da\ \ d) \mid b \leftarrow nd, d \leftarrow b].$$

This query calls for the construction of a bag Q by iterating over the bags in *nesteddogs* (i.e., $b \leftarrow nd$) and then over every dog d in those bags ($d \leftarrow b$). For every dog, a pair is constructed $((\cdot, \cdot))$.

The left hand of the pair is the name of the dog, which we get by applying the function *dogname* to the dog identifier ($dn\ d$). The right hand is constructed using *dogage*.

Translating Dodo query language expressions into the frame and column language corresponds to the upper two downward arrows in the DODO block in figure 1.1. Before constructing frames, Dodo first brings the expression in point-free form. Point-free form is explained in paragraph 16. The point-free form of Q is

$$\begin{aligned} Q' &= \mathsf{Bag}(onl\ dn \circ onr\ da \circ \Delta) \circ unnest_{\mathsf{Bag}} \circ const\ nesteddogs \\ &= \mathsf{Bag}\ onl\ dn\ \circ\ \mathsf{Bag}\ onr\ da\ \circ\ \mathsf{Bag}\ \Delta\ \circ\ unnest_{\mathsf{Bag}}\ \circ\ const\ nesteddogs. \end{aligned}$$

These two variations are equivalent. The first thing to note here is that unlike Q, query Q' is of function type, as signified by the occurrences of the functional composition symbol $\circ$. The reason we need Q' to be a function is that frames are functions, and we want to translate our query into a frame. Therefore, at some point Q needs to be turned into a function. The first step in our rewrite process is to wrap Q into a lambda expression, yielding

$$\lambda w \bullet Q : \{\dagger\} \to \mathsf{Bag}(\mathbb{S} \times \mathbb{Z}).$$

This is the expression that is subsequently rewritten to point-free form.

16 Point-free form. Reading Q' from right to left, the constant function *const nesteddogs* takes the initial † and produces a nested bag of dog identifiers: $\{\!|\{\!|1_1|\!\}_{10}, \{\!|7_7, 8_8|\!\}_{20}, \{\!||\!\}_{30}|\!\}_\dagger$. Then $unnest_{\mathsf{Bag}}$ removes one level of nesting, yielding $\{\!|1_1, 7_7, 8_8|\!\}_\dagger$. The Bag operator takes a function and applies it to all elements in a bag. The "split" operator $\Delta := (\lambda x \bullet (x,x))$ constructs identical pairs, so $\mathsf{Bag}\ \Delta$ transforms $\{\!|1_1, 7_7, 8_8|\!\}_\dagger$ into $\{\!|(1_1,1_1)_1, (7_7,7_7)_7, (8_8,8_8)_8|\!\}_\dagger$. The operators *onl* and *onr* apply an operator to the left- and right hand side of a pair respectively, giving the end result

$$\{\!|(\text{fido}, 3_1)_1, (\text{spot}, 1_7)_7, (\text{rex}, 9_8)_8|\!\}_\dagger.$$

Informally, an expression is in point-free form when it is expressed completely as a composition of functions, without any reference to individual elements at all. However, as demonstrated by *const nesteddogs* in Q', individual elements come back in through the back door in the form of constant functions. For every value, a constant function can be defined that always returns that particular value.

Function applications to individual elements such as $dn\ d$ in Q have in Q' been replaced by bulk function applications such as $\mathsf{Bag}\ onl\ dn$, hopefully eliminating some nested loop processing from the eventual query plan. However, the bulk operations still operate on potentially complex collection types. In the example we only used bags, but any other data structure for which a frame has been defined can be used. The next task in our translation process is to break down these complex bulk operations and replace them by "simple" ones, that is, operations that work on columns (MonetDB BATs).

17 To columns and frames. The key to understanding the translation from point-free form to the column and frame language is the fact that frames represent functions. Therefore, they can be substituted for functions. The translation proceeds by repeatedly taking the composition $(f \circ F)$ of a function f with a frame F and replacing it by a new frame F' that incorporates the action of f into F. As a simple example, consider the pair formation function Δ. For this function, Dodo has the following rewrite rule:

$$\Delta \circ F = pair\langle F, F\rangle.$$

Similarly, the *onl* operator, defined as $onl\ f\ (x,y) = (f\ x, y)$, has the rule

$$onl\ f \circ pair\langle F, G\rangle = pair\langle f \circ F, G\rangle.$$

In this way, every function defined in a Dodo schema comes with one or more rewrite rules which prescribe how to compose that function with a frame of suitable type. Often, the rewrite rules need column operators. For instance, the $unnest_{\mathsf{Bag}}$ function is rewritten like this:

$$unnest_{\mathsf{Bag}} \circ bag\langle d_1, r_1, bag\langle d_2, r_2, F\rangle\rangle = bag\langle d_1, r_1 * r_2, F\rangle$$

where $*$ is the relational composition (semijoin) operator.

To bootstrap the translation, a composition with $atom\langle idunit\rangle$ is appended to the query. Using rewrite rules similar to the above, the query

$$Q' \circ atom\langle idunit\rangle$$

is eventually rewritten to

$$bag\langle d_1, r_1 * r_2, pair\langle atom\langle d_3 * f\rangle, atom\langle d_3 * g\rangle\rangle\rangle.$$

The above frame is more complicated than the hand-crafted frame (1.4). If, however, metadata is attached to the columns, for instance a hint declaring that d_3 comprises the full space of dog keys, i.e., that every dog key in the system occurs exactly once in relation d_3, it is not hard for the system to simplify $d_3 * f$ into f. Similarly, if r_1 is an alias for $sethead(d_2, \dagger)$, and d_2 is known to comprise the space of owner keys, $r_1 * r_2$ can be simplified to

$$sethead(d_2, \dagger) * r_2 = sethead(d_2 * r_2, \dagger) = sethead(r_2, \dagger).$$

That Dodo finds $sethead(r_2, \dagger)$ here rather than $sethead(d_3, \dagger)$ is caused by the clumsiness of the example schema, which does not mention d_3 at all.

This concludes our informal description of the arrows from "Dodo query" to "column and frame query" in figure 1.1. A more detailed description is given in chapters 3 and 3.3.

18 Execution. After the query has been translated into the frame/column language, the column expressions are extracted from the frames, translated into the target language and executed by the target platform. The results are then put back into the frame structure. Currently, the target is MonetDB with its query language MIL. In diagram 1.1, these steps are indicated by the curved four-arrow rectangle in the lower part of the diagram. Arrow 1 is the translation from column expressions into MIL. Arrow 2 shows that the frame skeleton surrounding the column expressions is passed into the answer unmodified. Arrow 3 signifies execution by MonetDB, and arrow 4 represents the trivial transition from BATs (MonetDB domain) to columns (DODO domain).

In our example, the query

$$Q'' = bag\langle d_1, r_1 * r_2, pair\langle atom\langle d_3 * f\rangle, atom\langle d_3 * g\rangle\rangle\rangle$$

is split into

$$\begin{aligned} Q'' &= bag\langle m_1, m_2, pair\langle atom\langle m_3\rangle, atom\langle m_4\rangle\rangle\rangle, \\ m_1 &= generate_mil(d_1) = \texttt{d1} \\ m_2 &= generate_mil(r_1 * r_2) = \texttt{r1.join(r2)} \\ m_3 &= generate_mil(d_3 * f) = \texttt{d3.join(f)} \\ m_4 &= generate_mil(d_3 * g) = \texttt{d3.join(g)} \end{aligned}$$

After execution, this becomes

$$Q'' = bag\langle m_1, m_2, pair\langle atom\langle m_3\rangle, atom\langle m_4\rangle\rangle\rangle$$

with

(1.6)

m_1	
†	†

m_2	
†	1
†	7
†	8

m_3	
1	fido
7	spot
8	rex

m_4	
1	3
7	1
8	9

This is our result set, but still in in flattened form. The final step is to turn this flattened form back into a nested form for delivery to the user.

19 Dodo Result language. We have followed the example query from the application through Dodo and MonetDB until the point where a frame is constructed which represents the result value. The final step is to bring this flattened representation of the result in a more intuitive form:

$$\{\!\!\{(\mathrm{fido}_1, 3_1)_1, (\mathrm{spot}_7, 1_7)_7, (\mathrm{rex}_9, 9_8)_8\}\!\!\}_\dagger$$

rather than

$$\mathit{bag}\langle \begin{array}{c|c} \multicolumn{2}{c}{m_1} \\ \hline \dagger & \dagger \end{array}, \begin{array}{c|l} \multicolumn{2}{c}{m_2} \\ \hline \dagger & 1 \\ \dagger & 7 \\ \dagger & 8 \end{array}, \mathit{pair}\langle \mathit{atom}\langle \begin{array}{r|l} \multicolumn{2}{c}{m_3} \\ \hline 1 & \text{fido} \\ 7 & \text{spot} \\ 8 & \text{rex} \end{array}\rangle, \mathit{atom}\langle \begin{array}{r|l} \multicolumn{2}{c}{m_4} \\ \hline 1 & 3 \\ 7 & 1 \\ 8 & 9 \end{array}\rangle\rangle\rangle.$$

For every frame type there is an "interpretation function." Informally, this is a function of type $\mathit{Frame} \to \mathit{Key} \to X$ taking a frame and a key for which to retrieve the corresponding value. The return type is given here as X and the question is: what is X?

The simplest solution is to let X be the type of strings. For a $\mathit{bag}\langle\rangle$ frame, the interpretation function can build a string representation by interspersing the string representations of its elements with commas and enclosing the result in suitable brackets. This approach has the advantage that it is easy to understand and implement. However, string representations like this tend to be more suitable for human consumption than for further processing by machines.

In the Moa system, X takes the form of a series of method invocations on a "driver object." Basically, such an object defines `open(t)`, `close(t)` and `atom(x)` methods. In the $\mathit{bag}\langle\rangle$ example above, `open` corresponds to the opening bracket, `close` to the closing bracket and `atom` to the individual strings and numbers. Applying the interpretation functions corresponds to the second last arrow of diagram 1.1, the driver object corresponds to the final arrow back to the user. The interesting feature of using a driver object is that it allows the output to be displayed and processed in various ways. For instance, one can have a driver function which generates human-readable string representations and another one which generates XML.

From a theoretical point of view, the exact form of the "Dodo Result Language" X is mostly an engineering issue. Results have to be handed to the requesting application in a way that is suitable for the application in question. What matters however, is that whatever result language has been chosen, it has been specified with sufficient precision for defining unambiguous interpretation functions. As mentioned before, the interpretation function defines the semantics of a frame. Extending Dodo with new frame types means defining interpretation functions for them, and several other operations. The rewrite system depends on these operations to obey certain laws. If the result language and interpretation functions are not defined precisely enough, it becomes impossible to prove that the laws are satisfied, and chaos ensues.

20 Roles. It is often useful to distinguish between different kinds of roles a person can play from the point of view of Dodo.

An *extension writer* is someone with a more or less intimate knowledge of the inner working of the Dodo system. At the nested-structure layer, an extension can introduce new types by declaring a new frame type and giving an interpretation function for the new frame. New operators are added by declaring a name and a type for the new function, and giving rewrite rules that express them in terms of frames and columns. At the column level, an extension can declare new column operators by giving their name, type and a translation into the language of the underlying database system, for instance MIL or SQL.

A *schema writer* knows about the data in a specific database and is aware of the frame types and interpretation functions defined by various extensions. It is the task of the schema writer to map the data in the underlying system to columns and to organize the columns in frames, allowing users to pose queries over it in terms of complex data structures. In paragraph 14 we called the schema writer a database administrator. Schemas extend Dodo with data definitions just like extensions extend Dodo with operations. The distinction is mostly a convention, the underlying mechanisms are exactly the same.

A *user* primarily accesses the data in the database using the nested data structures defined by the schema writer and the operations defined by the extension writer.

21 Summary. In this section we have seen how nested data can be stored in a flattened way, and how queries expressed in terms of the nested representation can be transformed into queries over the flattened representation.

In the flattened representation, the data is organized in frames, which each represent a collection of values of a given type, identified by a locally unique key. Frames are constructed out of other frames and binary relations between atomic values. The nested form of a value is reconstructed using an "interpretation function" that takes a frame and a key and returns the corresponding value in the Dodo Result Language, the details of which do not influence the rest of the system very much.

The transformation of queries is done in two steps. First the query is turned into a composition of functions. A query consists of two things: references to data in the database and operations performed on this data. The operations are already in functional form, the data is represented as a constant function. A Dodo schema defines frames for these constant functions and frame transformations for operations. By applying these, we arrive at a frame representation of the (constant function of) the result value of the query.

Computations on complex data types have been replaced by operations on binary relations within the result frame. The second step is taking these "column expressions" out of the frames and executing them on the target database. To do so, the column expressions are first translated to the target database language, in our case MIL, the language of MonetDB. Semantically, the column algebra and MIL are very similar, so the translation is rather straight-forward. After execution, the results are put back in the frame and interpretation functions are used to construct a nested return value.

1.3 Differences with Moa

22 Same principle. Moa and Dodo are based on the same principle. Data structures are stored in a flattened form. The flattening roughly follows the nested structure of the data type: a bag of bags of strings is stored using two $bag\langle\rangle$ frames and one $atom\langle\rangle$ frame. Basically, for every type former a frame. The frame for a type former expresses the contribution of the type former to the structure of data in terms of columns (binary relations), delegating the structure of its argument types to their respective frames. There is a difference in naming: in this report we speak about *frames* where the Moa literature would talk about *structures*. This change was made to free the word "structure" for use in a more general way.

23 Value and IVS. Moa distinguishes between two kinds of structures: Value structures and IVS structures. A value structure represents a single value, an IVS structure represents a collection of values identified by keys. The acronym IVS stands for "indexed value set." In Dodo, IVS structures correspond to frames, whereas value structures correspond to the special case of a frame with domain $\{\dagger\}$. Moa distinguishes these two forms because an IVS is a partial function whereas a query asks for a single (but structured) result value, i.e., "the set of dogs that..." or "a list of names of...". As a consequence, value structures usually occur only at the outermost level of the query. They also occur in *modifiers*, which can be regarded as the bodies of lambda terms with a variable implicitly named $THIS$. A typical use of modifiers is turning a set $S = Set\langle d, r, F\rangle$ into a set of pairs:

$$map[Tuple\langle THIS, THIS\rangle](S) = Set\langle d, r, Tuple\langle F, F\rangle\rangle. \qquad (*)$$

We see here that the Moa *map* has two kinds of arguments: actual arguments between parentheses, and modifiers between square brackets. The arguments between the parentheses are structures, or things that are eventually rewritten to structures. The modifiers are functions, in this case the pair-forming function $\big(\lambda THIS \bullet (THIS, THIS)\big)$.

There are two disadvantages to the value/IVS distinction. One is simple: every operation in Moa is implemented twice. It is implemented once for value arguments and once for IVS arguments. The other disadvantage is more subtle and more fundamental. Looking closely at the

two occurences of *Tuple* in equation ($*$), it turns out that the one on the left is a *Tuple* value, where the one on the right is an *Tuple* IVS. The one on the left is a value structure because *THIS* refers to a single element of S. Therefore, the resulting tuple is also a single value. The *Tuple* on the right is part of a *Set* structure, and therefore an IVS. This phenomenon requires the implementation of *map* to be able to somehow convert value structures into IVS structures. For *Tuple*, this is easy, because *Tuple* values and IVSes have exactly the same structure. But for *Set* structures, which have a different structure in the value and IVS cases, it is a very hard problem. Moa deals with this simply by requiring value structures occuring in modifiers to be trivially translatable to IVS. Note however, that it is still possible to put *Set*-valued expressions in the modifier, only the actual $Set\langle\rangle$-structures are forbidden.

It is interesting to take a closer look at the difference between *Set* value and *Set* IVS in Moa. In the value case $Set\langle e, F\rangle$, it has two components. The first component contains identifiers of the elements of the set, and the second is an IVS in which these identifiers can be looked up. In the IVS case $Set\langle d, r, F\rangle$q, there are three components: a domain d, a relation r between set identifiers and element identifiers, and an IVS to look up the element keys.

24 Unifying value and IVS. Dodo unifies value and IVS structures into a single frame concept, with the value structure recognizable through the type system. Value structures correspond to frames with domain type $\{\dagger\} = 1$. Due to the categorical influence on its design, almost everything in Dodo is a function. The query result problem is solved by allowing the user to enter a query Q of type X, but explicitly wrapping it in a lambda-term $\lambda z \bullet Q : 1 \to X$ if X is not a function type. Operators with modifiers in Moa are in Dodo replaced by higher-order functions which take lambda terms as arguments. The class of lambda terms allowed corresponds precisely to the modifier they replace: first-order lambda terms, i.e., functions of type $A \to B$ with A and B non-function types. The advantage of lambda terms is that names other than *THIS* can be used, and that they are a convenient and well-known tool both in query formulation and in query processing. They also make it easier to implement things like the comprehension syntax in paragraph 48.

25 Labels. There is one particular aspect of the Moa language for which there is no true equivalent in Dodo. In Moa, to every subexpression a label can be attached, which is usually preserved by rewrite rules. These labels have many uses. Their original purpose was to make records out of tuples by labelling the components. Later, in XML-related applications, they have been used to indicate the xml-tag that should be wrapped around the data produced by particular subexpressions.

Convenient as they are, these labels are hard to fit into the more rigid type system used by Dodo. The closest equivalent would be a family of *label_foo* functors that basically include label-information in the type, lifting a type A to a type "A labeled foo." This is sufficient for XML-tags, but somewhat inconvenient for records, because when one needs to access one of the components of a record, one either needs to lift the accessing function to the label, or explicitly "unlabel" the value first. Time will tell whether all uses of labels can be expressed by existing Dodo mechanisms or that Dodo will have to be extended in one or more ways.

Chapter 2

Data Types

Before we describe the Dodo Query language and frame/column language in detail, we describe the data types it works on. In the Dodo layer (see figure 1.1) two type systems are used. One based on binary relations between atomic types and one based on complex data structures with nesting. The details of the relation-based type system are not very well-flushed out, more about this in section 3.1.

The type system for nested data developed in this chapter is based on a categorical notion of data types, similar to [8] and [6]. The main ideas borrowed from category theory are: data types are formed by applying type formers to other types; those type formers (functors) simultaneously map functions over argument types to functions over the new type; in general, the emphasis is more on the structure of functions between data types than the data types themselves, with special attention to functions that can be defined by induction to the construction of their arguments (catamorphisms); the *monad* concept gives a nice mathematical foundation for comprehension syntax, e.g., $\{x \in \mathbb{R} \mid x^2 < 2\}$.

In the first part of this chapter, the categorical theory of data types is described. When possible, categorical concepts are introduced under the guise of their Set-theoretic specialisations. For instance, we speak of functions rather than arrows and of data types rather than nodes or objects. Skipping as much as possible, we try to get to the notion of *catamorphisms* and introduce the monad concept. Then in section 2.3 we step back and discuss how all this applies to Dodo.

26 Notation. In this document, we write the application of a function f to a value e as $f\ e$ rather than $f(e)$. Lambda terms are written with a $\bullet$, that is, $\lambda x \bullet x$. Constant functions $\lambda x \bullet c$ are abbreviated to $\underline{c}$.

2.1 Type Formers

Every Dodo value has a type. The type is constructed out of the following elements:

27 Unit type. The unit type 1 contains precisely one value, †. This data type has the interesting property that it requires zero bytes of storage.

28 Primitive types. Primitive types are defined by the underlying system. In this report we assume *Int*, *Str* and *Bool*, sometimes written $\mathbb{Z}$, $\mathbb{S}$ and $\mathbb{B}$. The **if then else** construct requires the boolean type.

29 Function types. Function types $A \to B$ denote maps from one type to another. A function of type $A \to B$ assigns to *every* element of A an element of B. To ensure efficient evaluation on database back-ends there are limits on the complexity of function types that may actually occur in queries, see section 3.3 for details.

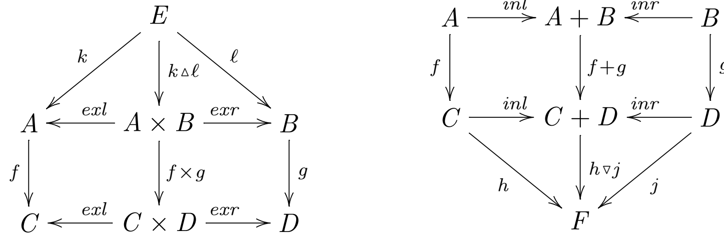

Figure 2.1: A summary of functions related to sum- and product types. This is a *commutative diagram.* That means that it is a graph with types as nodes and maps as edges. A path through the graph is a concatenation of edges and corresponds to a composition of maps. The existence of multiple paths between a given pair of nodes implies equality of the maps corresponding to those paths. For instance, among the equalities implied this way by the above diagrams are $exl \circ (k \vartriangle \ell) = k$ and $exl \circ (f \times g) = f \circ exl$.

30 Product types. An element of a product type $A \times B$ is a pair (a, b) with $a \in A$ and $b \in B$. The functions $exl : A \times B \to A$ and $exr : A \times B \to B$ are used to retrieve the left- and righthand parts. For every $k : E \to A$ and $\ell : E \to B$ we define $k \vartriangle \ell : E \to A \times B$ by

$$(k \vartriangle \ell)\ e = (k\ e,\ \ell\ e)$$

and for $f : A \to C$ and $g : B \to D$ we define $f \times g : A \times B \to C \times D$ by

$$(f \times g)\ (a, b) = (f\ a,\ g\ b).$$

The relationship between these functions is summarized in the left part of figure 2.1. The formula $k \vartriangle \ell$ is generally pronounced "k split ℓ".

31 Sum types. An element of a sum type $A + B$ is either a left-handed or a right-handed value. Left-handed values are drawn from type A, right-handed are drawn from type B. Left- and right-handed values are created using $inl : A \to A + B$ and $inr : B \to A + B$, respectively. Given $f : A \to C$ and $g : B \to D$, the function $f + g : A + B \to C + D$ applies f if it encounters a left-handed argument and g if encounters a right-handed argument. For $h : C \to F$ and $j : D \to F$, the function $h \triangledown j : C + D \to F$ applies h or j as appropriate, but returns the resulting F value without a left- or right-handed orientation. Again see figure 2.1 for a pictorial presentation of the relationship between these functions. The formula $h \triangledown j$ is generally pronounced "h junc j."

32 Functors. Types can be *lifted* to other types using *functors*. For example, the List functor transforms a type A into the type $\mathsf{List}A$ of lists over that type. At the same time, List transforms any function $f : A \to B$ into a new function $\mathsf{List}f : \mathsf{List}A \to \mathsf{List}B$ that applies f to every item in the list:

$$\mathsf{List}\ f\ [1, 2, 3] = [f\ 1, f\ 2, f\ 3].$$

A functor has two defining characteristics: it lifts the identity function of a type to the identity function of the new type

$$\mathsf{List}\ id\ [1, 2, 3] = [id\ 1, id\ 2, id\ 3] = [1, 2, 3] \tag{2.1}$$

and it distributes over composition

$$\begin{aligned} (\mathsf{List}\ f \circ \mathsf{List}\ g)\ [1, 2, 3] &= \mathsf{List}\ f\ (\mathsf{List}\ g\ [1, 2, 3]) = \mathsf{List}\ f\ [g\ 1, g\ 2, g\ 3] \\ &= [(f \circ g)\ 1, (f \circ g)\ 2, (f \circ g)\ 3] \\ &= \mathsf{List}\ (f \circ g)\ [1, 2, 3]. \end{aligned} \tag{2.2}$$

Defining new functors is a common way of extending Dodo. In the case of container types like $\mathsf{F} = \mathsf{List}$, the function $\mathsf{F}\ f$ is defined as applying f to the items of the list.

33 Bifunctors. It is possible to have functors that take more than one type as an argument. Such functors are called *bifunctors*. See, however, paragraph 59. The type formers $+$ and $\times$ for product- and sum types are examples of such bifunctors. Figure 2.1 illustrates their operation as function combinators. Proving the generalized functor properties $id_A \times id_B = id_{A\times B}$ and $(f \times f') \circ (g \times g') = (f \circ g) \times (f' \circ g')$ using the equations from that diagram is left as a useful exercise for the reader.

2.2 Algebras, monads and comprehensions

34 Constructors and destructors. The general pattern of data structures in Dodo is this: some data types, such as integer, string and boolean, are defined by the underlying database system. Dodo simply inherits them, together with functions to operate on them. Other data types are created by applying type formers to existing types. Examples of such type formers are $\times$, $+$ and functors. They are used in two ways: to construct new types $A \times B$ and to construct new functions $f \times g$. Type formers themselves do not provide ways to actually create or destroy instances of the types: $f \times g$ works on an existing pair and creates a new pair. Likewise, $\mathsf{List}\, f$ works on an existing list and creates a new list.

There are roughly three ways for a user to obtain values of a structured data type: explicit construction, transformation of other values and having it predefined. Explicit construction of a data type is done using its constructors. What those constructors look like depends on the type. Sum types are constructed using *inl* and *inr*, product types are constructed using $\vartriangle$ and lists are constructed using *nil* and *cons* operators, the first of which return the empty list while the second prepends an element to an existing list.

Constructing values explicitly is common for "small" data types like sum types and product types, but not for types like List. Values of such types are more often created by transforming existing values, i.e., sorting a bag or squaring the elements of a list of numbers. Of course, in the end such transformations are defined in terms of explicit construction, but from a user point of view, this is not visible. The question remains however, where the values being transformed come from. Sometimes, they may be explicitly constructed, but usually database queries over complex data structures start with data stored in the database. The example schema (1.5) illustrates how the schema defines a nested Bag structure directly in terms of data stored in the database, without reference to constructor functions.

35 Abstract data types. Recall the way lists of integers are defined, denoted L: there are two operations

1. $nil_L : 1 \to L$, returns an empty list;
2. $cons_L : \mathbb{Z} \times L \to L$, prepends a number to a list.

Different combinations of *nil*s and *cons*es yield different lists, and every list can be constructed using nil_L and $cons_L$. Functions on lists can be defined by induction to the $nil/cons$ construction of an argument, e.g.

$$\begin{aligned} f\ (nil\ \dagger) = f\ [\,] &= e, \\ f\ cons(x, \ell) &= x \oplus f\ \ell \end{aligned}$$

for suitable e and $\oplus$. To calculate the sum of the numbers in the list, one could take $e = 0$ and $\oplus = +$. The reader is encouraged to verify this by hand.

As another example of a data structure, bags of integers, denoted B, are defined using the same two kinds of operation, with in addition the equation

$$cons(x, cons(y, \ell)) = cons(y, cons(x, \ell)), \tag{2.3}$$

which expresses the indifference of bags towards the order in which elements are inserted. With lists, there was a one-to-one relation between the elements of L and the $nil/cons$-trees. With bags,

there is a one-to-one relation between elements of B and the *equivalence classes* induced on the trees by equation (2.3).

In general, abstract data types are modelled as an *algebra*: a base type together with a finite collection of operators. An *algebra with laws* is an algebra that additionally carries equations that govern the behaviour of its operators.

36 Combining operators into one. Using sum types, it is possible to combine *nil* and *cons* into a single operator

$$\tau = nil \triangledown cons : 1 + \mathbb{Z} \times L \to L.$$

The usefulness of this will soon become apparent. Combining is a reversible process. The original operators can be recovered using the *inl* and *inr* operators defined for sum types:

$$\begin{aligned} nil_L &= \tau \circ inl, \\ cons_L &= \tau \circ inr, \end{aligned}$$

as follows from the law $h = (h \triangledown j) \circ inl$ implied by figure 2.1. The reader is encouraged to verify this on a piece of paper.

37 Definition (Algebra). Given a functor F, an F-*algebra* is a function $\tau : \mathsf{F}A \to A$. In this definition, F represents the “signature” of the algebra. The example algebra has two operators: *nil* a constant one and *cons* taking a number and a list. Accordingly, $\tau = nil \triangledown cons$ has type $1+\mathbb{Z}\times L \to L$, which can be written $\mathsf{INS}\ L \to L$ if we define the functor INS by $\mathsf{INS}\ X = 1+\mathbb{Z}\times X$ and $\mathsf{INS}\ f = id_1 + id_{\mathbb{Z}} \times f$. Another example of an INS-algebra is the function $\underline{0} \triangledown (+)$ of type $1+\mathbb{Z}\times\mathbb{Z} \to \mathbb{Z} = \mathsf{INS}\ \mathbb{Z}$. In this example, $(+) : \mathbb{Z}\times\mathbb{Z} \to \mathbb{Z}$ is addition, and $\underline{0} : 1 \to \mathbb{Z}$ is the constant function $(\lambda z \bullet 0)$. Again, the reader is encouraged to verify that $\underline{0} \triangledown (+)$ is indeed an INS-algebra.

The name INS is derived from *insert algebra,* referring to the property that list values are constructed by starting with the empty list and inserting elements. An alternative representation for lists and bags would be using $nil : 1 \to L$, $tip : \mathbb{Z} \to L$ and $concat : L \times L \to L$. This is called a *union algebra* because now the fundamental operation is concatenation (union). Union algebras have signature functor $\mathsf{UN}\ X = 1 + \mathbb{Z} + X \times X$.

The class of F-algebras for a functor F is written $\mathcal{A}lg(\mathsf{F})$. The class of F-algebras that satisfy a set of equations E is written $\mathcal{A}lg(\mathsf{F}, E)$. Informally, an algebra $a \triangledown op$ satisfies, say, equation (2.3) if substituting $nil = a$ and $cons = op$ yields a valid equation.

38 Algebras with equations. In this paragraph we give a more formal definition of what it means for an algebra to satisfy an equation. Readers satisfied with the informal definition given above may skip this paragraph. For a full treatment of this subject, see [6].

We will write equation (2.3) in the form $T\ \tau = T'\ \tau$ with $\tau = nil \triangledown cons$. Here, $T\ \tau$ and $T'\ \tau$ are functions that take a value $(x, (y, \ell))$ and turn it into $cons(x, cons(y, \ell))$ and $cons(y, cons(x, \ell))$, respectively. Examples of *Transformers* T and T' that do this are

$$\begin{aligned} T\ \phi &= \phi \circ inr \circ \big(exl \triangledown (\phi \circ inr \circ exr)\big), \\ T'\ \phi &= \phi \circ inr \circ \big((exl \circ exr) \triangledown \big(\phi \circ inr \circ (exl \triangledown (exr \circ exr))\big)\big). \end{aligned}$$

In these intimidating looking definitions, the *inr* serves to select the proper sub-operation of ϕ, i.e., *cons* if $\phi = nil \triangledown cons$. The *exl*s and *exr*s serve to shuffle the x, y and ℓ of the argument into the proper positions in the equation. Any equation can be captured as a pair (T, T') of suitable transformers. We say that an algebra α satisfies (T, T') when the equation $T\ \alpha = T'\ \alpha$ holds.

39 Definition (Homomorphism). Many interesting operations can be regarded as a homomorphism between algebras. Let $\sigma : \mathsf{F}A \to A$ and $\tau : \mathsf{F}B \to B$ be F-algebras. A function $h : A \to B$

is an F-*homomorphism* from σ to τ, denoted $h : \sigma \to_{\mathsf{F}} \tau$, if $h \circ \sigma = \tau \circ \mathsf{F}\ h$. Pictorially, this equation reads

$$\begin{array}{ccc} \mathsf{F}A & \xrightarrow{\sigma} & A \\ \downarrow{\scriptstyle \mathsf{F}h} & & \downarrow{\scriptstyle h} \\ \mathsf{F}B & \xrightarrow{\tau} & B \end{array}$$

Homomorphisms have composition and identity: if $f : \beta \to_{\mathsf{F}} \gamma$ and $g : \alpha \to_{\mathsf{F}} \beta$ are homomorphisms, then $f \circ g$ is a homomorphism $\alpha \to_{\mathsf{F}} \gamma$.

40 Examples. A homomorphism well known from calculus is the exponentiation function $exp : \mathbb{R} \to \mathbb{R}$. To see this, take $\mathsf{F}\ X = X \times X$ with $\mathsf{F}\ f = f \times f$ and notice that the operations $(+)$ and $(\times)$ are F-algebras:

$$\begin{array}{rcl} (+) & : & \mathsf{F}\mathbb{R} \to \mathbb{R}; \\ (\times) & : & \mathsf{F}\mathbb{R} \to \mathbb{R}. \end{array}$$

Moreover, exp has the property that $exp(x + y) = exp(x) \times exp(y)$. In point-free form, this reads $exp \circ (+) = (\times) \circ \mathsf{F} exp$, which is exactly the definition of an F-homomorphism. In the world of data types, the well-known function $sum : L \to \mathbb{Z}$ is an INS-homomorphism $(nil_L \triangledown cons_L) \to_{\mathsf{INS}} (\underline{0} \triangledown (+))$. The following two diagrams demonstrate how the homomorphism equation

$$sum \circ (nil_L \triangledown cons_L) = (\underline{0} \triangledown (+)) \circ \mathsf{INS}\ sum$$

holds for both the *nil*- and the *cons*-case of the list data type:

$$\begin{array}{ccc} inl\ \dagger & \xmapsto{nil \triangledown cons} & [\,] \\ \downarrow{\scriptstyle id_1 + id_{\mathbb{Z}} \times sum} & & \downarrow{\scriptstyle sum} \\ inl\ \dagger & \xmapsto[\underline{0} \triangledown (+)]{} & 0 \end{array} \qquad \begin{array}{ccc} inr(2, [3,5]) & \xmapsto{nil \triangledown cons} & [2,3,5] \\ \downarrow{\scriptstyle id_1 + id_{\mathbb{Z}} \times sum} & & \downarrow{\scriptstyle sum} \\ inr(2,8) & \xmapsto[\underline{0} \triangledown (+)]{} & 2+8 \end{array} \tag{2.4}$$

Especially note how in the right-hand diagram, the arrow $\mathsf{F}\ sum = id_1 + id_{\mathbb{Z}} \times sum$ recursively applies sum to the sublist $[3, 5]$.

41 Definition (Initiality). An algebra τ in $\mathcal{A}lg(\mathsf{F}, E)$ is *initial* if there exists precisely one homomorphism $\tau \to_{\mathsf{F}} \sigma$ for every σ in $\mathcal{A}lg(\mathsf{F}, E)$. This unique homomorphism is written $(\!|\tau \to \sigma|\!)_{\mathsf{F},E}$, abbreviated to $(\!|\sigma|\!)_{\mathsf{F},E}$ or even $(\!|\sigma|\!)$. Homomorphisms from an initial algebra are called *catamorphisms*.

Lemma 43 provides a perhaps more intuitive interpretation.

42 Definition (Type functor). A *type functor* is a functor T for which an initial algebra $\tau : \mathsf{F}\ \mathsf{T}A \to \mathsf{T}A$ exists. Note that the choice of F must be inferred from the context. In this report, it will usually be INS.

43 Lambeks lemma. An algebra $\tau : \mathsf{F}A \to A$ is initial in $\mathcal{A}lg(\mathsf{F}, E)$ if and only if is is a bijection between E-equivalence classes of F-trees and values in A. For example, in paragraph 35 we saw that every list can be written using nil_L and $cons_L$ in exactly one way. We also saw that in the case of bags, every bag corresponds to precisely one $[cons(x, cons(y, \ell)) = cons(y, cons(x, \ell))]$-equivalence class. In contrast, the algebra $(\underline{0} \triangledown (+)) : \mathsf{F}\mathbb{Z} \to \mathbb{Z}$ is not initial in $\mathcal{A}lg(\mathsf{F}, \text{eqn. } (2.3))$: the number 5 can be written in different ways that are not (2.3)-equivalent:

$$\begin{array}{rcl} 5 & = & 2 + 3 = (\underline{0} \triangledown (+))\ (inr\ (2, 3)), \\ 5 & = & 1 + 4 = (\underline{0} \triangledown (+))\ (inr\ (1, 4)), \\ & & \ldots \end{array}$$

44 Obtaining initiality. When a functor F is polynomial, i.e., when it can be expressed completely in terms of sum-, product, constant and identity functors, $\mathcal{A}lg(\mathsf{F}, E)$ has an initial algebra. This covers most of the algebra signatures we come across in practice, INS in particular. The algebra $nil_L \triangledown cons_L$ is the initial INS-algebra. The algebra $nil_B \triangledown cons_B$ is the initial algebra under equation (2.3).

Second, an algebra τ initial in $\mathcal{A}lg(\mathsf{F}, E)$ is also initial in $\mathcal{A}lg(\mathsf{F}, E \cup E')$. A proof for the case $E = \emptyset$ can be found in [6], paragraph 39 on page 60. We are not aware of a proof for the general case. We use this in paragraph 57.

45 Initiality in operation. The pictures in paragraph 40 suggest that *sum* functions as if every *cons* is replaced by a $(+)$ and every *nil* by $\underline{0}$. This property holds for catamorphisms in general. *Catamorphisms are exactly the functions that can be expressed by induction to the recursive construction of their arguments.*

To see this, consider an arbitrary catamorphism $h = (\!|\sigma \to \tau|\!)_{\mathsf{F}}$ in $\mathcal{A}lg(\mathsf{F})$. Because h is a homomorphism, we have

$$h \circ \sigma = \tau \circ \mathsf{F}\ h.$$

Initiality implies bijectivity, therefore,

$$h = \tau \circ \mathsf{F}\ h \circ \sigma^{-1}.$$

Specializing to $\mathsf{F} = \mathsf{INS}$ for greater familiarity, $h = \tau \circ (id_1 + id_{\mathbb{Z}} \times h) \circ \sigma^{-1}$ as applied to a value v can be expressed in three steps. The first step σ^{-1} essentially maps v either to $inl\ \dagger$ or $inr(x, xs)$, depending on whether v is constructed using *nil* or *cons*, respectively. The final step τ replaces *inl* by, say, $\underline{0}$ and *inr* by $(+)$. The middle step ensures that this procedure is applied to every position in the *nil*/*cons* decomposition of v. This argument can be generalized to general F-homomorphisms and also to algebras with laws.

46 Use of initiality. Two common things to express as a catamorphism are conversions ("casts") and aggregate functions. Given a list, we can apply the catamorphism $(\!|nil_B \triangledown cons_B|\!)$ to it to obtain a bag. This is a conversion. We can also apply $(\!|\underline{0} \triangledown (+)|\!)$ to calculate the sum of the numbers in the list. This is an aggregate function. Writing *sum* as a catamorphism clearly exposes its structure: one initial intermediate result $\underline{0}$ and one function $(+)$ that takes a value and an intermediate result. Readers familiar with functional programming will surely recognize this familiar *fold*-pattern. In the sequel, catamorphisms will be indispensible in the implementation of the comprehension syntax. We will define $List[f\ x \mid \ldots]$ comprehensions, and also $Sum[f\ x \mid \ldots]$ comprehensions. The only difference between them is that when they are rewritten to point-free form, the former uses a $(\!|nil \triangledown cons|\!)$ catamorphism, where the latter uses a $(\!|\underline{0} \triangledown (+)|\!)$ catamorphism.

There are several useful laws about catamorphisms, the most obvious of which reads

$$h : \alpha \to_{\mathsf{F}} \beta \quad \Longrightarrow \quad h \circ (\!|\alpha|\!)_{\mathsf{F},E} = (\!|\beta|\!)_{\mathsf{F},E}.$$

Given $\tau : \mathsf{F}A \to A$ initial in $\mathcal{A}lg(\mathsf{F}, E)$, this theorem uses the uniqueness and existence of the homomorphism $(\!|\phi|\!)_{\mathsf{F},E}$ for every ϕ to simplify compositions of homomorphisms that start in τ. Although the peculiarities of the Dodo approach introduce some interesting complications, this law can be useful for for deriving alternative query plans.

This theorem and some similar ones are collectively known as *fusion* theorems because they combine multiple homomorphisms into one. More on the application of fusion in Dodo can be found in paragraph 54.

47 Polymorphism. In the preceding paragraphs we talked about the type L of lists of integers and used it to illustrate the concept of an algebra, homomorphism, etc. But in section 2.1 we introduced the List functor which could construct list types over arbitrary types, not just over

integers. To extend the algebra concept to parametrized types, we need to fix the signature functor INS. Until now, we defined the INS functor as

$$\begin{aligned} \mathsf{INS}\ X &= 1 + \mathbb{Z} \times X, \\ \mathsf{INS}\ f &= id_1 + id_{\mathbb{Z}} \times f, \end{aligned}$$

and the "list of integers" type as the carrier of the initial INS-algebra $\tau : \mathsf{INS}\ L \to L$. With this definition, the integer type $\mathbb{Z}$ is hard-coded into INS. We solve this by making INS a bifunctor, with the additional parameter being the element type:

$$\begin{aligned} \mathsf{INS}(A, X) &= 1 + A \times X, \\ \mathsf{INS}(f, g) &= id_1 + f \times g. \end{aligned}$$

The homomorphism condition $h \circ \tau = \sigma \circ \mathsf{F}\tau$ is reformulated to $h \circ \tau = \sigma \circ \mathsf{F}(id, h)$. With these modifications, we again define the List type former to yield the carrier of the initial algebra in $\mathcal{A}lg(\mathsf{INS})$, i.e.,

$$\tau_A : \mathsf{INS}(A, \mathsf{List}\ A) \to \mathsf{List}\ A.$$

When no confusion can arise, we de-emphasize the dependence on the element type by writing $\mathsf{INS}_A X$ or even $\mathsf{INS}\ X$ instead of $\mathsf{INS}(A, X)$.

48 Comprehension examples. Comprehension notation provides a convenient way to write down collection values and aggregations. A comprehension expression $M[\ |\ \ldots]$ consists of a comprehension name, a head and a sequence of qualifiers. The comprehension name is required; in contrast to other literature and programming languages, the notation $[\ |\ ]$ has no meaning in itself. Examples of comprehensions with $xs = [1, 2, 3]$ are

$$\begin{aligned} List[x^2 \mid x \leftarrow xs] &= [1, 4, 9], \\ Sum[x^2 \mid x \leftarrow xs] &= 1 + 4 + 9 = 14, \\ List[x + y \mid x \leftarrow xs,\ y \leftarrow xs] &= [2, 3, 4,\ 3, 4, 5,\ 4, 5, 6], \\ List[x + y \mid x \leftarrow xs,\ y \leftarrow xs, x < y] &= [3, 4, 5] \\ Set[x + y \mid x \leftarrow xs,\ y \leftarrow xs] &= \{2, 3, 4,\ 3, 4, 5,\ 4, 5, 6\} \\ &= \{2, 3, 4, 5, 6\}. \end{aligned}$$

On the right-hand side of the *List* comprehension, the spacing suggests a grouping of the elements in three sublists, the first corresponding to $x = 1$, the second to $x = 2$ and the third to $x = 3$. One can imagine the list $[2, 3, 4,\ 3, 4, 5,\ 4, 5, 6]$ to have been formed by first generating

$$\begin{aligned} & List[List[x + y \mid y \leftarrow xs] \mid x \leftarrow xs] \\ = \ & [List[1 + y \mid y \leftarrow xs], List[2 + y \mid y \leftarrow xs], List[1 + y \mid y \leftarrow xs]] \\ = \ & [[2, 3, 4],\ [3, 4, 5],\ [4, 5, 6]] \end{aligned} \tag{2.5}$$

and then crossing out the inner brackets to yield $[2, 3, 4,\ 3, 4, 5,\ 4, 5, 6]$. We will use the *monad* concept to make this more precise.

49 Definition (Monad). A T-monad is a triple $(\mathsf{T}, unit, unnest)$ with T a type former and *unit* and *unnest* families of functions $unit_A : A \to \mathsf{T}A$ and $unnest_A : \mathsf{T}\mathsf{T}A \to \mathsf{T}A$ that satisfy the equations in the following commutative diagrams:

$$\begin{array}{ccccc} \mathsf{T}A & \xrightarrow{unit_{\mathsf{T}A}} & \mathsf{T}\mathsf{T}A & \xleftarrow{\mathsf{T}unit_A} & \mathsf{T}A \\ & {\scriptstyle id_{\mathsf{T}A}} \searrow & \downarrow {\scriptstyle unnest_A} & \swarrow {\scriptstyle id_{\mathsf{T}A}} & \\ & & \mathsf{T}A & & \end{array} \qquad \begin{array}{ccc} \mathsf{T}\mathsf{T}\mathsf{T}A & \xrightarrow{\mathsf{T}unnest_A} & \mathsf{T}\mathsf{T}A \\ {\scriptstyle unnest_{\mathsf{T}A}} \downarrow & & \downarrow {\scriptstyle unnest_A} \\ \mathsf{T}\mathsf{T}A & \xrightarrow{unnest_A} & \mathsf{T}A \end{array} \tag{2.6}$$

The intuition here is that $unit_A$ adds an extra level of nesting while $unnest$ removes one. As an example, consider

$$\begin{array}{rcl} \mathsf{T} & = & \mathsf{List}, \\ unit_A\ x & = & [x], \\ unnest_A\ [[a_1, \ldots, a_n], \ldots, [z_1, \ldots, z_m]] & = & [a_1, \ldots, a_n, \ldots, z_1, \ldots, z_m] \end{array}$$

It is interesting to try out the equalities from (2.6) on this monad:

$$\begin{array}{ccc} [1,2] & \xmapsto{unit_{\mathsf{T}A}} & [[1,2]] \\ & \searrow^{id_{\mathsf{T}A}} & \downarrow unnest_A \\ & & [1,2] \end{array} \qquad \begin{array}{ccc} [[1],[2]] & \xleftarrow{\mathsf{T}unit_A} & [1,2] \\ \downarrow unnest_A & \swarrow_{id_{\mathsf{T}A}} & \\ [1,2] & & \end{array} \qquad \begin{array}{ccc} [[[1],[2,3]],[]] & \xmapsto{\mathsf{T}unnest_A} & [[1,2,3],[]] \\ \downarrow unnest_{\mathsf{T}A} & & \downarrow unnest_A \\ [[1],[2,3]] & \xmapsto{unnest_A} & [1,2,3] \end{array}$$

50 Definition (Monad with Zero). A monad with zero $(\mathsf{T}, unit, unnest, zero)$ is a T-monad with an additional function $zero : X \to \mathsf{T}A$ that returns an "empty" T. As $zero$ ignores its argument, any type X will do. The zero function must satisfy

$$\begin{array}{ccc} \mathsf{TT}A & \xleftarrow{\mathsf{T}\ zero_A} & \mathsf{T}X \\ & \searrow^{unnest_A} & \downarrow zero_A \\ & & \mathsf{T}A \end{array} \qquad \begin{array}{ccc} X & \xrightarrow{zero_{\mathsf{T}A}} & \mathsf{TT}A \\ \downarrow zero_A & \swarrow_{unnest_A} & \\ \mathsf{T}A & & \end{array}$$

The obvious $zero$ candidate for List is $zero\ x = [\,]$, which indeed satisfies the equations:

$$\begin{array}{ccc} [[\,]] & \xleftarrow{\mathsf{T}\ zero_A} & [3] \\ & \searrow^{unnest_A} & \downarrow zero_A \\ & & [\,] \end{array} \qquad \begin{array}{ccc} 3 & \xmapsto{zero_{\mathsf{T}A}} & [\,] \\ \downarrow zero_A & \swarrow_{unnest_A} & \\ [\,] & & \end{array}$$

51 Comprehension syntax. The Dodo comprehension syntax demonstrated in paragraph 48 consists of three parts. First, the name of the comprehension type, i.e., List. Second, the head of the comprehension, i.e., x^2 or $x + y$. This describes the constituents of the new value in terms of variables bound in the third part, the tail. The tail consists of generators and filters. Generators bind variables, and filters impose conditions on the values of the variables. Generators are of the form $name \leftarrow value$, where $value$ must have type $\mathsf{T}A$ with T a type functor.

A *comprehension type* M is a pair $(\mu, (\mathsf{T}, unit_M, unnest_M))$ of an F-algebra $\mu : \mathsf{F}\ \mathsf{T}A \to \mathsf{T}A$ and a T-monad. The semantics of comprehensions is given by translating to comprehensionless syntax according to the following rules:

- A comprehension $M[e \mid\ ]$ is rewritten to $(unit_M\ e)$. So, $List[3 \mid\ ] = unit_{\mathsf{List}}\ 3 = [3]$.
- A comprehension $M[e \mid x \leftarrow xs]$ with $xs : \mathsf{T}'A$ is rewritten to $(\!|\mu|\!)(\mathsf{T}'(\lambda x \bullet e)\ xs)$. So,

$$\begin{array}{rcl} Sum[x^2 \mid x \leftarrow xs] & = & (\!|\underline{0} \triangledown (+)|\!)\ (\mathsf{List}\ (\lambda x \bullet x^2)\ xs) \\ & = & (\!|\underline{0} \triangledown (+)|\!)\ [1,4,9] = 1 + 4 + 9 = 14. \end{array}$$

- As hinted at in paragraph 48, a comprehension $M[e \mid qs, qs']$ where qs and qs' are parts of the tail is rewritten to $unnest_M\ M[M[e \mid qs'] \mid qs]$. So, first an intermediate result is generated that contains an extra level of nesting, then the nesting is removed using $unnest$. Referring back to equation (2.5),

$$\begin{array}{rl} & List[x + y \mid x \leftarrow xs,\ y \leftarrow xs] \\ = & unnest_{\mathsf{List}}\ List[List[x + y \mid y \leftarrow xs] \mid x \leftarrow xs] \\ = & unnest_{\mathsf{List}}\ [[2,3,4],\ [3,4,5],\ [4,5,6]] \\ = & [2,3,4,\ 3,4,5,\ 4,5,6] \end{array} \tag{2.7}$$

- Finally, if M has a monad with zero, $M[e \mid b]$ with b a boolean expression is rewritten to

$$\textbf{if } b \textbf{ then } unit_M\ e \textbf{ else } zero_M\ e \textbf{ fi}.$$

Using these four rules, every Dodo comprehension is transformed into an equivalent expression that does not use the comprehension syntax.

52 Common Comprehensions. Every type functor gives rise to a comprehension type. Let T be a type functor with corresponding initial INS-algebra $\tau = nil \triangledown cons$. Then we define the monad type T to be $(\tau, (\mathsf{T}, unit_{\mathsf{T}}, unnest_{\mathsf{T}}, zero_{\mathsf{T}}))$ with

$$\begin{aligned} unit_T\ x &= cons(x, nil\ \dagger), \\ zero_T\ z &= nil\ \dagger, \\ unnest_T\ xs &= (\!| nil \triangledown concat |\!)\ xs, \end{aligned}$$

where

$$concat_T\ (xs, ys) = (\!| \underline{ys} \triangledown cons |\!)\ xs.$$

This gives comprehension syntax for data types that can be expressed in insert notation, such as the collection types list, bag and set. It can readily be generalized to algebras of other signatures, such as union representation,

Comprehension for aggregates are defined using an Id-monad. For instance, the function $\underline{0} \triangledown (+)$ has type $\mathsf{INS}\ \mathsf{Id}A \to \mathsf{Id}A$. In such a case we define the *Sum* comprehension type as

$$((\!|\underline{0} \triangledown (+)|\!), (\mathsf{Id}, unit = id, unnest = id, zero = \underline{0})).$$

Because $A = \mathsf{Id}A = \mathsf{Id}\ \mathsf{Id}A$, the *unit* and *unnest* functions do nothing.

2.3 Application in Dodo

53 Mapping to storage layout. In the nested data model, data types are defined in terms of constructor algebras like $\tau = nil \triangledown cons$. But the point of Dodo is that they are actually stored in a very different, flattened way. Data types are added to Dodo in the form of extensions. The extension writer specifies the flattened level storage layout using columns (binary relations) and defines a mapping from nested level operations to flattened level operations. The extension writer should choose the storage layout in such a way that nesting-related operations such as *unnest* map onto relatively efficient relational operations such as semijoins. In paragraph 17 we already saw an example of such a mapping.

54 No arbitrary catamorphisms. As a consequence of storing data in a flattened form, Dodo cannot evaluate arbitrary catamorphisms. If the data is stored in a nested form, then it is always possible to take two arbitrary functions f and e of suitable type and walk the *INS*-structure, performing an e operation on *nil*-nodes and an f operation on *cons* nodes. This is nested loop processing, so its use is discouraged, but as a last resort it can be done. But Dodo does not store its data according to its algebraic structure, it stores it as a bunch of columns grouped in a frame. Consequently, the operations it can perform on it are only those column/frame operations that are provided by extension writers.

In paragraph 46 we mentioned theorems that can be used to combine adjacent catamorphisms and homomorphisms, eliminating the materialisation of an intermediate result. The risk is, however, that we end up with a catamorphism for which no column/frame equivalent is known. Determining how the theorems can still be used without losing the capability of breaking the fused catamorphisms up again into known frame operations is an interesting line of future research. Paragraph 56 sketches a couple of elementary optimizations that are possible using a tool called the *homomorphism graph*, which is useful because Dodo cannot do without this homomorphism graph anyway.

55 Homomorphism graph. In order to express catamorphisms such as $(\!|\mu|\!)$ from paragraph 51 in terms of known operations, Dodo maintains a homomorphism graph. The nodes of this graph are algebras. Algebras are connected by an arc if Dodo knowns a homomorphism between them. If the homomorphism is actually implemented in an extension, the arc is labeled with the name of the implemented function. Anonymous arcs can be used for optimization but cannot occur in the final query plan because Dodo does not have an implementation for them.

Exactly how the algebras are represented depends on the implementation. One can imagine naive Dodos storing algebras simply as a name, and sophisticated Dodos storing them in a more detailed representation that makes it possible to derive more optimizations. For a first attempt, it seems sufficient to represent the algebras as user-provided names, just like we often use greek letters instead of $_ \triangledown _$ formulas in the examples. However, there is one exception: if τ is an initial F-algebra for a type T, then the lifted function $\mathsf{T}f$ can be expressed as $(\!|\tau \circ \mathsf{F}(f, id)|\!)$. Using fusion theorems (paragraph 46) it can be shown that if h is an homomorphism $\tau \rightarrow_{\mathsf{F}} \beta$, it is also an homomorphism $\tau \circ \mathsf{F}(f, id) \rightarrow_{\mathsf{F}} \beta \circ \mathsf{F}(f, id)$ for any f. Because lifted functions are so common, it seems that adding a representation for $\phi \circ \mathsf{F}(f, id)$ will make the homomorphism graph much more useful.

56 Homomorphism graph optimizations. The homomorphism graph can also be used for simple optimizations. Consider the following system:

$$\begin{array}{ccccc} \rho & \xleftarrow{rev} & \tau & \xrightarrow{\mathsf{T}\,f} & \tau \circ \mathsf{F}(f, id) \\ & \searrow^{lb} & \downarrow{lb} & & \downarrow{lb} \\ & & \beta & \xrightarrow{\mathsf{B}\,f} & \beta \circ \mathsf{F}(f, id) \\ & & \downarrow{sum} & & \\ & & \sigma & & \end{array} \qquad (2.8)$$

with τ declared the initial F-algebra and β the initial (F, E)-algebra. Assume that $E = \{(2.3)\}$ expresses the indifference of an algebra towards the insertion order of its elements. We briefly describe every arc in the graph and justify why it is reasonable Dodo is made aware of it.

The function $sum : \mathsf{B}\mathbb{Z} \rightarrow \mathbb{Z}$ is an homomorphism $\beta \rightarrow_{\mathsf{F}} \sigma$. Therefore, $sum = (\!|\sigma|\!)_{\mathsf{F},E}$. Likewise, we have the list reversal function $rev = (\!|\rho|\!)_{\mathsf{F}} : \mathsf{T}A \rightarrow \mathsf{T}A$ and the conversion function $lb = (\!|\beta|\!)_{\mathsf{F}} : \mathsf{T}A \rightarrow \mathsf{B}A$. That these functions are indeed homomorphisms cannot be checked by Dodo. They are just declared as such by the extension writer.

The dashed arrow from $\tau \circ \mathsf{F}(f, id)$ to $\beta \circ \mathsf{F}(f, id)$ expresses the fact referred to in the previous paragraph that first transforming the elements one by one ($\mathsf{T}\ f$) and then converting to a bag (lb) is equivalent to first converting to a bag (lb) and then transforming the elements ($\mathsf{B}\ f$). This is a useful rule to have built-in to the system because lifting a function f to $\mathsf{T}\ f$ occurs so often.

Finally, reversing a list and then converting to a bag is a waste of time. The definition of commutative diagrams (page 14) together with the existence of an arrow $lb : \rho \rightarrow_{\mathsf{F}} \beta$ expresses that every composition $lb \circ rev$ can immediately be replaced by just lb.

Now consider the query $Sum[f\ x \mid x \leftarrow rev\ xs]$, which in time gives rise to the query fragment

$$(\!|\sigma|\!)_{\mathsf{F}} \circ \mathsf{T}\ f \circ rev.$$

Looking at the graph, Dodo notices that $(\!|\sigma|\!)_{\mathsf{F}} : \tau \rightarrow_{\mathsf{F}} \sigma$ can be written $sum \circ lb$, yielding

$$sum \circ lb \circ \mathsf{T}\ f \circ rev.$$

The fragment $lb \circ T\ f$ connects τ to $\beta \circ \mathsf{F}(f, id)$ and there is another route: $\mathsf{B}\ f \circ lb$, allowing us to write

$$sum \circ \mathsf{B}\ f \circ lb \circ rev.$$

Applying the same trick again, we replace $lb \circ rev$ by a shorter path from τ to β: just lb. In the resulting query, no list reversal is performed, and it is also conceivable that operating on bags is cheaper than operating on lists because we no longer need to keep track of the ordering. This concludes our brief example of optimizations using the homomorphism graph.

57 No Bag type needed. In example 56, the function *sum* was defined on bags. Initiality allowed Dodo to derive a $(\!|\sigma|\!)$ for lists, and because the extension writer had declared lb to also be an homomorphism from ρ to β we could simplify the expression considerably. The question is: could we also have done this if no convenient bag type had been available? The answer is yes.

In paragraph 56 we picked the algebra $\beta : \mathsf{INSB}A \to \mathsf{B}A$ as a convenient initial object of $\mathcal{A}lg(\mathsf{INS}, E)$, where $E = \{(2.3)\}$ represents indifference to order. If there is no bag type available, we can just use another initial (INS, E)-algebra.

Paragraph 44 promises the existence of an algebra τ' that constructs lists just as τ does, but with the added promise that the order of the elements in the list shall never be considered. The catamorphism $(\!|\tau \to \tau'|\!)$ can be implemented as id because the underlying implementation remains the same. The updated homomorphism graph becomes

$$\begin{array}{ccccc} \rho & \xleftarrow{rev} & \tau & \xrightarrow{\mathsf{T}\,f} & \tau \circ \mathsf{F}(f, id) \\ & \searrow^{id} & \downarrow id & & \downarrow id \\ & & \tau' & \xrightarrow{\mathsf{B}\,f} & \tau' \circ \mathsf{F}(f, id) \\ & & \downarrow sum & & \\ & & \sigma & & \end{array} \tag{2.9}$$

58 Weakness of homomorphism graph optimizations. The primary purpose of the homomorphism graph is to derive implementations for catamorphisms. The optimization $(\!|\sigma|\!) \circ \mathsf{T}f \circ rev = sum \circ \mathsf{B}f \circ lb$ looks very nice but may lead the reader to expect more than the homomorphism graph is able to provide. For instance, it is hard to see how laws such as $rev \circ \mathsf{T}f = \mathsf{T}f \circ rev$ can be derived from it.

59 No full bifunctors. Functors take a function and lift it to another type. Similarly, bifunctors take a pair of functions. However, to simplify the intermediate query representation used in section 3.3, we will not allow bifunctors in the prototype. Instead of writing $f \times g$ and $f + g$, we define functors *onl*, *onr*, *ifl* and *ifr* in such a way that

$$\begin{aligned} f \times g &= (f \times id) \circ (id \times g) =: onl\ f \circ onr\ g, \\ f + g &= (f + id) \circ (id + g) =: ifl\ f \circ ifr\ g \end{aligned} \tag{2.10}$$

and use these instead. To support $\triangledown$ and $\vartriangle$, we also need $\nabla : X + X \to X$ and $\Delta : X \to X \times X$ defined as

$$\begin{aligned} \Delta x &= (x, x) \\ \nabla x &= \begin{cases} x_1 & \text{if } x = inl\ x_1, \\ x_2 & \text{if } x = inr\ x_2. \end{cases} \end{aligned}$$

These definitions allow us to write $f \triangledown g = \nabla \circ ifl\ f \circ ifr\ g$ and $f \vartriangle g = ifl\ f \circ ifr\ g \circ \Delta$. The names *onl* and *onr* have been chosen because they apply a function **on** the left- or right side. Likewise, *ifl* and *ifr* apply a function **if** their argument is left-handed or right-handed, respectively.

The functions *ifl*, *ifr*, *onl* and *onr* are functors but not type functors: they have no initial algebra even though their bifunctor counterpart obviously has.

60 Summary. In this rather heavy-going chapter we first described the type system for the nested-structures part of Dodo. It is used for the query language, the intermediate point-free representation and for the frames as they fit into point-free form. The column expressions within the frames have their own type system, described in section 3.1.

After describing the way data types can be combined into new types, we looked at how values of those types can be constructed. Using sum types, the algebra of constructors for a type can be combined into a single function $\tau : \mathsf{F}X \to X$, which we use for our formal definition of an algebra. Algebras with equations can be modelled by building a pair (T, T') of transformers that each transform an algebra into an equation term. Using this pair we can tell whether an algebra satisfies the equation. Just like functions can be regarded as arrows between sets, homomorphisms are arrows between algebras. Some algebras α, including the algebras that correspond to data types, have the property that there is exactly one outgoing arrow (a catamorphism) for every algebra β in the same class. This is a very important property because it implies that the operation of these homomorphisms is completely determined by the target algebra and the recursive structure of its argument. The Fusion Theorems allow one to combine adjacent catamorphisms into one, potentially allowing a more efficient query.

Initiality and catamorphisms play an important role in the definition of the comprehension syntax. We give a translation scheme based on that in [8] that translates comprehensions into a chain of *unit*, *unnest* and catamorphism applications. The catamorphisms are used to aggregate the values in the comprehension or to convert from one collection type to another.

We end the chapter by discussing the role all this plays in Dodo. First we remind the reader that Dodo only pretends to build data structures using their constructor algebra because the data is actually stored in a flattened way. As a consequence, Dodo can only perform those catamorphisms that it can patch together out of homomorphisms defined on the flattened storage of the data structures. We introduce the homomorphism graph as a simple tool to derive a chain of defined operations for catamorphisms and demonstrate that it can also be used for simple optimizations. More optimizations are possible with a more detailed representation of the algebras in the graph that allows the use of fusion theorems.

Chapter 3

Language

This chapter describes the Dodo Query Language, the Frame/Column language and the translation to point-free form. Although these were presented in chapter 1 as two separate languages, they are in fact two faces of the same language, which we shall refer to as the Dodo language. The Dodo Query Language is the part of the language that expresses solely the nested structures described in chapter 2. The Frame/Column language is the part that deals with the flattened nested structures. During rewriting, the expression contains a mixture of the two sublanguages. Rewrite rules are applied until the expression is completely expressed in the Frame/Column sublanguage.

In the first section we first have a look at the column type system. Then we list all syntactic constructs of the Dodo language and the corresponding type rules. In the third section we describe the translation scheme that translates all expressions to point-free form.

61 Expressive power. Dodo implements a first order typed λ-calculus without an Y-operator or similar. This means that it cannot directly express recursive functions; such functions are hard to define in terms of column- and frame transformations. Consider a function that walks a list by pattern matching on its *nil*, *cons*-structure. In general it would be very hard to translate this function into something that takes a frame representing the list and constructs a frame representing the result. Higher-order functions and recursive functions can only be defined in Dodo if their semantics are expressed in terms of rewrite rules on frames. In the case of recursive functions on data types this means declaring catamorphisms from one algebra to another. Higher-order lambda terms are not permitted.

As an example of a higher-order function defined using a rewrite rule consider the List functor. Assume that lists are implemented as frames $\mathit{list}\langle d, r, F\rangle$ with d and r columns and F a frame. If the rewriting scheme explained in paragraph 16 encounters a composition

$$\mathsf{List} f \circ \mathit{list}\langle d, r, F\rangle,$$

it uses the rewrite rule accompanying List to transform this into

$$\mathit{list}\langle d, r, (f \circ F)\rangle.$$

62 Predefs. An identifier declared in an extension and defined in terms of a rewrite rule that works on or at least produces frame expressions is often called a *predef*. Examples of identifiers that are not predefs are names bound by **let**, lambda terms or comprehensions.

3.1 Column type system

63 Column types. Columns are sets of pairs, each consisting of a *head* and a *tail* element of primitive type. All heads have the same type, and so do the tails. The type of a column consists of its head- and tail type, together with flags that indicate

- that the head (tail) elements are all distinct;
- that they are *complete*, ie., that every meaningful value in the domain is present as a head (tail);
- that every head is equal to its tail.

A column with head type K and tail type L is denoted $\mathcal{R}_{K-L}$. Partial functions, i.e., columns where all heads are distinct, are written $\mathcal{R}_{K\rightarrow L}$. Columns with distinct tails are written $\mathcal{R}_{K\leftarrow L}$, and head-complete and tail-complete columns are written $\mathcal{R}_{K\vdash L}$ and $\mathcal{R}_{K\dashv L}$, respectively. Finally, all tails being equal to the heads can be indicated by replacing the tail type with an $=$ symbol: the column $\mathit{idunit} = \{(\dagger, \dagger)\}$ has type $\mathcal{R}_{1\leftarrowtail\!\!\twoheadrightarrow=}$. The arrow ornaments can be combined, allowing us to write $\mathcal{R}_{K\mapsto L}$ for total functions from K to L. We use the arrows as subscripts to the letter $\mathcal{R}$ to avoid confusion with the arrows used by function types in the nested-structure layer of the type system. There, the arrow $\rightarrow$ denotes a total function rather than a partial function.

A column expression usually denotes a column, i.e., a value of type $\mathcal{R}_{K-L}$ as described above. However, sometimes we need to represent a single value of type K. Such singular column values mostly arise as arguments to an operator, i.e., $\mathit{settail}(\mathit{some_col}, 3)$. Therefore we extend the type system for the column layer with the possibility of a simple primitive type K. In paragraph 64 we introduce the convention of using the letters X and Y to denote either primitive types or columns.

64 Type letters. Primitive types are usually denoted using the letters K, L and M. The letter M is also used for monad types. We write greek letters α, β, ... if subsets are allowed, see section p:partitionprim. For types formed using primitive types and the type formers listed in section 2.1 we use the letters T and S. In the common case where function arrows $\rightarrow$ do not occur in the type, we write A and B. Therefore, the type $A \rightarrow B$ denotes the set of first-order function.

As described above, column types are written $\mathcal{R}_{\alpha-\beta}$. In frame expressions, the components of the frame can be either columns or expressions of type $\alpha \rightarrow A$. In cases where an expression can have either of these types, we use the letter X.

Algebras are also often written as greek letters. In practice, this does not lead to confusion.

65 Partitioning primitive types. Consider the implementation of sum types $A + B$. We represent functions $F : K \rightarrow A + B$ as frames

$$F = \mathit{either}\langle G, H\rangle$$

where $G : K_1 \rightarrow A$ and $H : K_2 \rightarrow B$ with $K = K_1 \cup K_2$ and $K_1 \cap K_2 = \emptyset$. The question is: how can we represent this in the type system? A related question: with $c : \mathcal{R}_{\alpha\rightarrow\beta} = \{(\alpha_1, \beta_1), \ldots, (\alpha_n, \beta_n)\}$ a column and $F : \beta \rightarrow A = \{\beta_1 \mapsto a_1, \ldots, b_k \mapsto a_k\}$ a frame, we will soon introduce the construct $c * F : \alpha \rightarrow A$, a frame representing the function $\{x \mapsto z \mid (x, y) \in c,\ (y, z) \in F\}$. If column c is complete, that is, if every $x \in \alpha$ occurs as a head in c, then the domain of $c * F$ is obviously α. But what is the domain if c is not complete? The best we can say is that the domain of $c * F$ is a subset of α. The latter question is a typical example of the problems occuring at the boundary between functions and general relations: not every element of its domain or range need occur in a relation. An extreme example of this is the empty relation $\emptyset$. We need a way in the type system to reason about unions and subdivisions of sets.

The Dodo type system allows primitive types to have a *subset indicator.* Syntactically, this looks like $\alpha[pq + r]$. In a subset indicator, a letter p stands for a subset, a juxtaposition pq for an intersection, and $p + q$ stands for a union. We write $\alpha[\overline{p}] = \alpha \setminus \alpha[p]$ and the indicator 0 and 1 represents the empty and complete set: $\alpha[0] = \emptyset$ and $\alpha[1] = \alpha$. All laws suggested by this notation hold, i.e., $p(q + r) = pq + pr$ and $0 + p = p = p + 0$. Moreover, subsets of subsets can be obtained using the law $\alpha[p_1 + \cdots + p_n][q_1 + \cdots + q_m] = \alpha[(p_1 + \cdots + p_n)(q_1 + \cdots + q_m)]$.

All indicator expressions can be reduced to a normal form in which the indicator is a disjunction of terms, each of which is a a conjunction in which every letter occurs, i.e., the normal form of $\alpha[p + q]$ is

$$\alpha[p + q] = \alpha[p1 + 1q] = \alpha[p(q + \overline{q}) + (p + \overline{p})q] = a[pq + p\overline{q} + \overline{p}q].$$

The reason we require every letter to occur is that this way, α is split into a collection of disjoint subsets that are atomic in the sense that they cannot be subdivided using p and q. Union and intersection of two two subsets of α can now be determined purely using the terms in their respective subset indicators:

$$\alpha[p] \cup \alpha[q] = \alpha[pq + p\overline{q}] \cup \alpha[pq + \overline{p}q] = \alpha[pq + p\overline{q} + \overline{p}q].$$

66 Unification of subset indicators. Type inference typically proceeds by assigning each node in the parse tree a very general type and then determining the most general type substitution that allows the nodes to fit together. In the absence of subsets, unification of types formed by the type formers in section 2.1 is straightforward, but with the addition of subset types, we need a new rule.

Consider the identity

$$\alpha[x_1 + \cdots + x_n] = \alpha[y_1 + \cdots + y_m].$$

We assume that the two subset indicators have already been made compatible as in the previous paragraph. The sets may have some terms in common. Without loss of generality we assume that they have the first k terms in common, giving

$$\alpha[x_1 + \cdots x_k + x_{k+1} + \cdots + x_n] = \alpha[x_1 + \cdots x_k + y_{k+1} + \cdots + y_m].$$

Each of the indicators $x_1, \ldots, x_n, y_{k+1}, \ldots y_m$ denotes a distinct disjoint subset of α. From x_{k+1}, …, x_n only occurring on the left hand side and $y_{k+1}, \ldots, y_m$ only occurring on the right hand side we can immediately derive the minimal indicator substitutions that make the equation true:

$$\begin{aligned} x_i &= 0 \quad \text{for} \quad i = k+1, \ldots, n, \\ y_i &= 0 \quad \text{for} \quad i = k+1, \ldots, m. \end{aligned}$$

3.2 Syntax

In this section we list all syntactic constructs of the Dodo language, usually together with a typing rule. The constructs are grouped by type. Constructs that construct a value of a type that in paragraph 64 would get the letter A are described in paragraph 68. Constructs that build values of the more general type T are in paragraph 69. The constructs in paragraph 70 always construct first-order functions $A \to B$. The final two paragraphs construct frames (type $\alpha \to A$) and columns (type $\mathcal{R}_{\alpha-\beta}$), respectively.

Conceptually, everything in these paragraphs can be regarded as part of a huge context free grammar starting with e ::=. It is tempting to attempt to put more syntactical structure in it by putting, say, the frame constructs in their own kind of nonterminal. This looks nice if one only considers the nested-structure-only expressions that go into the rewrite process and the frame-only expressions that come out of it, but it makes the mixed trees that occur during the rewrite process very hard to describe. The most striking example is that after rewriting, a frame only has columns and frames in it, but during rewriting it is allowed to contain any expression that can conceivably be translated to a frame, that is, every expression of type $\alpha \to A$. Because of these issues, we just call everything an *expression* and use the type system to impose structure on it.

67 Identifier syntax. The prototype uses a peculiar format for identifiers. Identifiers consist of letters, digits and underscores, but they may also contain `_{*}` substrings, where the star denotes a sequence of any printing character except }. This permits identifiers like `op_{=}`. We take this approach because it is a cheap way to obtain a very large namespace with room for avoiding name clashes. This is useful because we often would like to use the same name, i.e., *equal*, at both the nested-structure level, the column level and in the underlying system. In paragraph 73 we make another concession to this problem by prefixing all column names with an exclamation mark.

Also, due to the way the prototype is constructed, extensions can define identifiers dynamically. That is, extensions need not provide a complete list of the identifiers they provide, instead an extension can decide to respond to all identifiers that, say, match the pattern `label_*`. This can be used to implement the labeling functors mentioned in paragraph 25. These dynamic symbol tables are not an essential part of the Dodo approach but they do provide many possibilities to work around its deficiencies.

68 Value building constructs. The following syntactical constructs can be used to construct simple values, i.e., values of a type A, not $X \to Y$.

k	:	K	literal	
$\mathbf{let}\ x = e\ \mathbf{in}\ e'$	:	A	abbrev. for $(\lambda x \bullet e')\ e$	if $e'_{x:=e} : A$
(e, e')	:	$A \times B$	pair formation	if $e : A$ and $e' : B$
$\mathbf{if}\ b\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$	:	A	conditional	if $b : \mathbb{B}$ and $e_1 : A$ and $e_2 : A$
$M[e \mid\ ]$	:	$\mathsf{T}A$	monad-comprehension	if $e : A$
$M[e \mid x \leftarrow e']$	:	$\mathsf{T}B$	monad-comprehension	if $e_{x:A} : B$ and $e' : \mathsf{T}'A$ with τ init. algebra for T' and $(\!\mid \tau \to \mu \mid\!)$ well-defined
$M[e \mid b]$	:	$\mathsf{T}A$	monad-comprehension	if $e : A$ and $e' : \mathbb{B}$
$M[e \mid q, qs]$	:	$\mathsf{T}A$	monad-comprehension	if $M[M[e \mid qs] \mid q] : \mathsf{TT}A$.

In this table, T is the type functor belonging to the comprehension type M, and μ is the catamorphism.

69 Potentially complex function builders. The following constructs may have function types other than $A \to B$.

x	:	T	identifier	defined in schema or environment
$f\ e$	:	T	function application	if $f : S \to T$ and $e : S$
$f \circ g$	:	$T \to T'$	function composition	if $f : S \to T'$ and $g : T \to S$

70 Simple function builders. The following construct always have first-order function type $A \to B$.

$\lambda x \bullet e$	:	$A \to B$	lambda term	if $e_{x:A} : B$
$(\!\mid \alpha \to \beta \mid\!)$	:	$A \to B$	catamorphism	if $\alpha : \mathsf{F}A \to A$ initial , $\beta : \mathsf{F}B \to B$
$\mathit{const}\ v$	:	$A \to B$	constant function sometimes written $\underline{v}$	A arbitrary, $v : B$

71 Frame expressions. Frames can be formed out of other frames or written explicitly.

$f\langle e_1, \ldots, e_n \rangle$	:	$\alpha \to A$	frame	if $e_1 : t_1, \ldots, e_n : t_n$, each t_i either $\alpha_i \to A_i$ or X_i, and $\mathcal{F}_f(\alpha \to A, t_1, \ldots, t_n)$
$F +\!\!+ G$	:	$\alpha \to A$	frame union	if $F : \alpha[p] \to A$ and $G : \alpha[\overline{p}] \to A$.
$c * F$	:	$\alpha \to A$	frame translation	if $c : \mathcal{R}_{\alpha \mapsto \beta}$ and $F : \beta \to A$

Every frame type f has a type predicate $\mathcal{F}_f$ that relates the type of its components to the type of the whole frame. Every component should either be a column expression (type X_i) or

something translatable to a frame (type $\alpha_i \to A_i$). Recall the *bag* frame from paragraph 10. The predicate $\mathcal{F}_{bag}$ for the *bag* frame reads

$$\mathcal{F}_{bag}(t, t_1, t_2, t_3) \iff t = \alpha \to A \;\wedge\; t_1 = \mathcal{R}_{\alpha \leftarrowtail\twoheadrightarrow =} \;\wedge\; t_2 = \mathcal{R}_{\alpha - \beta} \;\wedge\; t_3 = \beta \to A.$$

In chapter 4 we abbreviate this to

$$bag\langle \mathcal{R}_{\alpha \leftarrowtail\twoheadrightarrow =},\ \mathcal{R}_{\alpha-\beta},\ \beta \to A\rangle : \alpha \to A.$$

Frame union combines two frames with non-intersecting domains into a combined frame. A typical use is the ∇ operator on sum types:

$$\nabla \circ either\langle F, G\rangle = F +\!\!+ G.$$

Frame translation is used to give a frame a new domain type by composing it with the relation between the new domain and the old. The relation c must be functional because otherwise the translated frame no longer has function type. It must be head-complete because if a key in α is not present in c, it is not present in $c * F$, which implies it does not conform to the total function type $\alpha \to A$.

72 The *none* frame. There is one special frame, the $none\langle\rangle$-frame. This frame has type $\emptyset \to X$ for any X and is used as a placeholder in situations where a certain subframe contains no data. For instance, in $inl \circ f\langle\cdots\rangle = either\langle f\langle\cdots\rangle, none\langle\rangle\rangle$ all the values from f go in the left-hand slot of the $either\langle\rangle$-frame, and $none\langle\rangle$ is put in the right-hand slot because no right-hand elements are required there. The $none\langle\rangle$ frame has a special property with respect to the frame union operator:

$$none\langle\rangle +\!\!+ f\langle\cdots\rangle = f\langle\cdots\rangle = f\langle\cdots\rangle +\!\!+ none\langle\rangle,$$

for any frame $f\langle\rangle$.

Note: in the equations above, the *none*-frame has no components. The *none*-frame as defined by the prototype has a single component e, which is an empty column. This is explained in paragraph 91.

73 Column expressions. Column expressions denote primitive values but most often columns. They occur only within frames expressions.

k	:	K	literal	
$!x$	:	$\mathcal{R}_{\alpha-\beta}$	column name	
$op(c_1, \ldots, c_n)$	:	X	prefix operator	if $\mathcal{C}_{op}(X, t_1, \ldots, t_n)$
				and $c_1 : t_1,\ \ldots,\ c_n : t_n$
$c * c'$	:	$\mathcal{R}_{\alpha-\gamma}$	semijoin	if $c : \mathcal{R}_{\alpha-\beta}$ and $c' : \mathcal{R}_{\beta-\gamma}$
				note property propagation rules below
$c +\!\!+ c'$	:	$\mathcal{R}_{\alpha-\beta}$	concatenation	if $c, c' : \mathcal{R}_{\alpha-\beta}$
dom F	:	$\mathcal{R}_{\alpha \leftarrowtail\twoheadrightarrow =}$	domain column	if $F : \alpha \to A$

Literals are used in column expressions such as $settail(!some_col, 3)$. Column names are preceded by an exclamation mark to avoid name clashes with frames; often, a frame called *foo* has an underlying column for which *foo* would also be the natural name. Prefix operations *op* have a type predicate $\mathcal{C}_{op}$ that describes the permitted argument types. The semijoin operator preserves the column attributes: if both arguments are head-complete, then so is their semijoin. The same holds for tail-complete, tail-distinct and (very important) head-distinct, a.k.a functional. The concatenation operator preserves head-uniqueness if it exists, which is essential when combining, e.g. *bag* domains. The domain operator takes a frame and returns it domain as a binary identity relation. This is used together with frame translation to implement the *const* v construct. Let v

be a value, either a literal or a complex value defined by the schema. Let it be represented as the single range element of a frame $F : 1 \to A$, in other words, let $v = F\ \dagger$. The point-free expression $const\ v \circ G$ can be rewritten

$$const\ v \circ G = settail(\mathrm{dom}\ \ G, \dagger) * F,$$

resulting in a frame that maps every key in the domain of G to v.

74 Core predefs. In order to support the rewrite process described in section 3.3, certain predefined identifiers are needed. Every Dodo implementation supports the unit type, sum- and product types and the *Id* functor. Useful implementations also provide some primitive types to work with other than the unit type, and typically one or more type functors that provide collection types such as lists or bags. To support the minimal set of data types described above, we need type formers 1, $+$, $\times$ and Id, and predefs for id, Δ, onl, onr, exl, exr, ∇, ifl, ifr, inl and inr defined in section 2.1 and paragraph 59.

For every functor F, such as List or inl, the rewrite process needs a *distribution function* $D_{\mathsf{F}} : A \times \mathsf{F}B \to \mathsf{F}(A \times B)$ that implements the following behaviour:

$$D_{\mathsf{F}}(x, xs) = \mathsf{F}\ (\lambda x' \bullet (x, x'))\ xs, \quad \text{i.e.,} \quad D_{\mathsf{List}}(3, [10, 20, 30]) = [(3, 10), (3, 20), (3, 30)].$$

Distribution functions are used in section 3.3 to "straighten out" nested subexpressions with more than one free variable.

For type functors T Dodo needs a designated initial algebra τ_{T} that describes how to construct values of the type. It is used as the source-algebra of a catamorphism when values of the type are used as a generator in a comprehension. For every comprehension type M Dodo needs a corresponding functor T_M, algebra τ_M and $zero_M$, $unit_M$, $unnest_M$ and $concat_M$ functions. As described in paragraph 52, there are two common cases. For collection comprehensions, T_M is the type functor, τ_M the corresponding initial algebra, and *unit*, *unnest* and the other functions are non-trivial. For aggregations, T_M is Id, τ_M is the algebra that does the actual work, and the helper functions are usually trivial. Finally, support for conditional expressions requires a function $bool2sum : \mathbb{B} \to 1 + 1$. It is used to rewrite **if** b **then** e_1 **else** e_2 **fi** to

$$\big(\nabla \circ \big((\lambda x \bullet e_1) + (\lambda x \bullet e_2)\big)\big)\ (bool2sum\ b).$$

3.3 To point-free form

In this section we define point-free form in the context of Dodo and give a translation scheme by giving a translation rule for every syntactical construct.

75 Definition (Point-free form for Dodo). A Dodo expression is in point-free form if it has type $A \to B$ for simple A and B and it is a predefined function, a constant function $const\ v$ with v predefined, a composition of point-free Dodo expressions, a predefined higher-order function operating on a point-free Dodo expression or a frame expression.

The above definition implies that frame expressions are in point-free form. This is a bit strange but it does no harm. We go to point-free form in order to eliminate variables and other references to individual values. Once we are in point-free form, all operations are bulk-operations. Afterwards, as the expression is translated to a frame, the bulk operations remain. Moreover, if the user enters a frame expression by hand, we do not attempt to eliminate it. Therefore it is reasonable in the context of Dodo to include frame expressions in the definition of point-free form.

76 Translation scheme. Every Dodo expression of type $A \to B$ that is closed except for identifiers defined in an extension or schema can be written in point-free form. In the following paragraphs we show this by systematically considering every way an expression E can have such a type. We use the bracket notation $[\![E]\!]$ to denote the point-free form of E.

77 Function builders. Consider the constructs of paragraphs 70. Lambda expressions are treated in paragraph 78. Catamorphisms are replaced by a suitable sequence of predefs found in the homomorphism graph. Constant functions and frames are point-free already.

If E is something from paragraph 69, there are three possibilities: If E is an identifier, it is an identifier of type $A \to B$ defined by an extension or schema, so $[\![E]\!] = E$. Similarly, $[\![f \circ g]\!] = [\![f]\!] \circ [\![g]\!]$.

In the case $[\![f\ e]\!]$, f is a higher-order function. Therefore, f itself must be one of the constructs from paragraph 69. If f is a predef, we can write $[\![f\ e]\!] = f\ [\![e]\!]$, which is in point-free form by definition. If $f = f_1 \circ f_2$ then $[\![(f_1 \circ f_2)\ e]\!] = [\![f_1(f_2(e))]\!]$. If f is of the form $(g\ h)$ for some expressions g and h, then one of the identifiers in g is itself higher-order and should bring a rewrite rule for this situation.

78 Simple lambda term. By far the most common situation is that E is of the form $E = \lambda y \bullet E'$ with $E' : B$ a construct from paragraph 68 or 69. The translations for paragraph 68 are listed in the following table.

$$\begin{aligned}
[\![\lambda y \bullet k]\!] &= \mathit{const}\ k \\
[\![\lambda y \bullet \mathbf{let}\ x = e\ \mathbf{in}\ e']\!] &= [\![\lambda y \bullet e'_{x=e}]\!] \\
[\![\lambda y \bullet (e, e')]\!] &= \mathit{onl}\ [\![\lambda y \bullet e]\!] \circ \mathit{onr}\ [\![\lambda y \bullet e']\!] \circ \Delta \\
[\![\lambda y \bullet \mathbf{if}\ b\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\ \mathbf{fi}]\!] &= [\![\lambda y \bullet \big(\nabla \circ (\mathit{const}\ e_1 + \mathit{const}\ e_2)\big)\ (\mathit{bool2sum}\ b)]\!] \\
[\![\lambda y \bullet M[e \mid\ ]]\!] &= [\![\lambda y \bullet \mathit{unit}_M\ e]\!] \\
[\![\lambda y \bullet M[e \mid x \leftarrow e']]\!] &= [\![\lambda y \bullet (\!|\tau_{\mathsf{T}'} \to \tau_M|\!)\big(\mathsf{T}'\ (\lambda x \bullet e)\ xs\big)]\!] \\
[\![\lambda y \bullet M[e \mid b]]\!] &= [\![\lambda y \bullet M[e \mid \mathit{dummy} \leftarrow \mathbf{if}\ b\ \mathbf{then}\ \mathit{unit}_M\ \dagger\ \mathbf{else}\ \mathit{zero}_M\ \dagger\ \mathbf{fi}]]\!] \\
[\![\lambda y \bullet M[e \mid qs, qs']]\!] &= [\![\mathit{unnest}_M M[M[e \mid qs'] \mid qs]]\!]
\end{aligned}$$

Most of these rules only eliminate syntactical sugar such as **if** and comprehension syntax.

The first construct from paragraph 69, the single identifier can be handled similar to literals: $[\![\lambda y \bullet x]\!] = \mathit{const}\ x$. As far as the rewrite scheme is concerned, a literal is nothing more than a special kind of predefined identifier. Obviously, E' cannot be a composition; that would not be type correct. The only other possibility for E' is a function application $\lambda y \bullet f\ e$. This case is handled in the next paragraph.

In paragraph 116, an alternative rewrite rule for $[\![\lambda y \bullet M[e \mid b]]\!]$ is proposed:

$$[\![\lambda y \bullet M[e \mid b]]\!] = \mathsf{T}\ (\lambda y \bullet e) \circ \mathit{FILTER}_M\ (\lambda y \bullet b)$$

with FILTER_M defined by the equation

$$\mathit{FILTER}_M\ f = \mathit{unnest}_M \circ \mathsf{T}\ (\lambda x \bullet \mathbf{if}\ (f\ x)\ \mathbf{then}\ \mathit{unit}_M\ xv\ \mathbf{else}\ \mathit{zero}_M\ x\ \mathbf{fi}).$$

In principle, this is equivalent, however, the presence of *FILTER* is easier to spot than the equivalent ∇, *bool2sum* and other operators.

79 Application within lambda. In the case $\lambda y \bullet f\ e$ there are two possibilities. Either y occurs free in f or it does not. If y is not free in f, rewriting is very simple:

$$[\![\lambda y \bullet f\ e]\!] = [\![f]\!] \circ [\![\lambda y \bullet e]\!].$$

The interesting case is when y does occur in f. By applying suitable extension-provided rules to any occuring higher-order functions in f, we can assume that f is either a lambda term or a higher-order function Σ applied to something else. In the first case we apply the reduction rule

$$[\![\lambda y \bullet (\lambda z \bullet e)e']\!] = [\![\lambda y \bullet e_{z:=e'}]\!].$$

The second case is exemplified by $\Sigma = \mathsf{List}$. It is interesting because in $\lambda y \bullet (\mathsf{List}\ g)\ e$, the function g refers not only to the elements of e fed to it by the List functor, but also to the value of y. Moreover, e can also depend on y.

In paragraph 74, we required extensions to provide a function $D_\Sigma : A \times SB \to \Sigma(A \times B)$ defined by $D_\Sigma(x, xs) = \Sigma\ (\lambda z' \bullet (x, x'))\ xs$ for every functor Σ. The rewrite rule

$$[\![\lambda y \bullet (\Sigma\ g)\ e]\!] = \Sigma\ (\lambda z \bullet g_{y:=exl\ z}\ (exr\ z)) \circ D_\Sigma \circ onr\ (\lambda y \bullet e) \circ \Delta$$

uses these distribution functions to explicitly pair every constituent x of e with the surrounding y. The function g is transformed into a $g' = (\lambda z \bullet g_{y:=exl\ z}(exr\ z))$ that works on the resulting pairs.

We will see that in most cases, it will not be hard to map the operation of D_Σ to a corresponding frame transformation. For instance,

$$D_{\mathsf{Bag}} \circ pair\langle F, bag\langle d, r, G\rangle\rangle = bag\langle d, mkprod(r), pair\langle prodleft(r) * F,\quad prodright(r) * G\rangle\rangle.$$

For an explanation of the *mkprod*, *prodleft* and *prodright* operators see paragraph 86.

80 Examples of distribution functions. The table below illustrates the behaviour of the distribution functions for the built-in functors for products, sums and the identity functor. The definitions in terms of frames and columns are given in chapter 4.

$$\begin{array}{rcl}
D_{\mathsf{Id}}\ (a, b) & = & (a, b) \\
D_{onl}\ \big(a, (b, c)\big) & = & \big((a, b), c\big) \\
D_{onr}\ \big(a, (b, c)\big) & = & \big(b, (a, c)\big) \\
D_{ifl}\ (a, inl\ b) & = & inl\ (a, b) \\
D_{ifl}\ (a, inr\ b) & = & inr\ b \\
D_{ifr}\ (a, inl\ b) & = & inl\ b \\
D_{ifr}\ (a, inr\ b) & = & inr\ (a, b)
\end{array}$$

Chapter 4

The prototype

We intend to implement a small prototype of the rewrite process. This section describes in some detail the types and rewrite rules for all the core operations from paragraph 74, plus some extra operations needed to give a meaningful demonstration.

The operations defined in the prototype are the bare minimum needed to demonstrate interesting cases. For types, that means $\mathbb{B}$, $\mathbb{Z}$ and $\mathbb{S}$ types, plus List and Bag functors. Having two primitive types makes examples much easier to comprehend because they can be used as "before" and "after" types, i.e., *to_str*. We have an $lb : \mathsf{List}A \to \mathsf{Bag}A$ operation and a $sum : \mathsf{Bag}A \to \mathbb{Z}$ operation, both catamorphisms between appropriate algebras. Furthermore, we define functions *any* and *all* of type $\mathsf{Bag}\mathbb{B} \to \mathbb{B}$ that calculate the catamorphisms $(\!| false \triangledown (\vee) |\!)$ and $(\!| true \triangledown (\wedge) |\!)$, respectively. Together with these catamorphisms we define list-, bag-, sum-, exists- and forall-comprehensions.

We also define an equality function *eq* for at least the unit type, the primitive types, sum types and product types. Equality of bags and lists will probably be left unimplemented for the time being.

81 Bag columns. The columns used in the prototype are *bags* of pairs rather than sets of pairs, because the same is true in MonetDB. This makes surprisingly little difference to the implementation of the frame operations in this chapter, because the use of column properties such as head-completeness does not really change, and because we are already very careful whenever we take the union of two sets that the sets are disjoint.

It does mean, however, that the semantics of the column operations in section 4.1 is defined in terms of bag-braces $\{\!\!\{ \ \}\!\!\}$ rather than set-braces $\{ \ \}$.

4.1 Column operators

The choice of the operations here is based on the ideas in paragraph 12. Practical considerations may make it more attractive to have a column algebra in which domain restriction and all the other things that are expressable using join and twin are present as explicit operators. Fortunately, the line of thinking underlying the translation scheme does not depend on the particular form of the core algebra.

82 Column creation. The schema defines columns identified by their name. These can be used in queries. In this paragraph we introduce some operators useful for constructing columns by hand. The first is *colpair*, which takes a pair of primitive values and returns a column containing just this pair:

$$colpair(x : \alpha, y : \beta) = \{\!\!\{(x, y)\}\!\!\} : \mathcal{R}_{\alpha-\beta}.$$

The second operator is *emptycol*. It creates an empty column. It takes one argument, which is a column from which it takes the type of the column to return:

$$emptycol(c : \mathcal{R}_{\alpha-\beta}) = \{\!\!\{\}\!\!\} : \mathcal{R}_{\alpha[0]\mapsto\!\!\!\rightarrow\beta[0]}.$$

Having it take an "example column" argument may seem strange for a function that constructs empty relations, but it helps avoid defining a mechanism for functions to take types as an argument. It greatly simplifies the type checker if the result type of a column operator is completely determined by its argument types.

Among other uses, *colpair* is used to implement $idunit = colpair(\dagger, \dagger)$, and *emptycol* is used to construct the domain of the *none*-frame.

83 Twin and sethead. There are several operations which overwrite head and/or tail with either a constant, head or tail.

$$\begin{array}{rcl}
converse(c : \mathcal{R}_{\alpha-\beta}) & = & \{\!\!\{(y, x) \mid (x, y) \leftarrow c\}\!\!\} : \mathcal{R}_{\beta-\alpha} \\
sethead(c : \mathcal{R}_{\alpha-\beta}, v : \gamma) & = & \{\!\!\{(v, y) \mid (x, y) \leftarrow c\}\!\!\} : \mathcal{R}_{\gamma-\beta} \\
settail(c : \mathcal{R}_{\alpha-\beta}, v : \gamma) & = & \{\!\!\{(x, v) \mid (x, y) \leftarrow c\}\!\!\} : \mathcal{R}_{\alpha-\gamma} \\
twin(c : \mathcal{R}_{\alpha-\beta}) & = & \{\!\!\{(x, x) \mid (x, y) \leftarrow c\}\!\!\} : \mathcal{R}_{\alpha-=} \\
rtwin(c : \mathcal{R}_{\alpha-\beta}) & = & \{\!\!\{(y, y) \mid (x, y) \leftarrow c\}\!\!\} : \mathcal{R}_{\beta-=}
\end{array}$$

Of these functions, *settail* and *twin* preserve head-properties, *sethead* and *rtwin* preserve tail-properties and *converse* switches them. Moreover, the *twin* and *rtwin* propagate the properties also to the other attribute. To keep our type system satisfied, we also provide special *twinu* and *rtwinu* operators that weed out duplicates:

$$\begin{array}{rcl}
twinu(c : \mathcal{R}_{K-L}) & = & distinct\ \{\!\!\{(x, x) \mid (x, y) \leftarrow c\}\!\!\} : \mathcal{R}_{K\leftrightarrow=}, \\
rtwinu(c : \mathcal{R}_{K-L}) & = & distinct\ \{\!\!\{(y, y) \mid (x, y) \leftarrow c\}\!\!\} : \mathcal{R}_{L\leftrightarrow=}.
\end{array}$$

These, too, propagate completeness properties. The most important use of *twinu* is in constructing frame domains.

84 Boolean operators. In the prototype, we assume that the underlying system supports booleans (possibly represented as integers that are either zero or not) and we provide the usual operations.

$$\begin{array}{rcl}
op_{not}(b : \mathcal{R}_{\alpha\mapsto\!\!\!\rightarrow\mathbb{B}}) & = & \{\!\!\{(h, \neg v) \mid (h, v) \leftarrow b\}\!\!\} : \mathcal{R}_{\alpha\mapsto\!\!\!\rightarrow\mathbb{B}} \\
op_{and}(b_1 : \mathcal{R}_{\alpha\mapsto\!\!\!\rightarrow\mathbb{B}}, b_2 : \mathcal{R}_{\alpha\mapsto\!\!\!\rightarrow\mathbb{B}}) & = & \{\!\!\{(h, b_1(h) \wedge b_2(h)) \mid (h, x) \leftarrow b_1\}\!\!\} : \mathcal{R}_{\alpha\mapsto\!\!\!\rightarrow\mathbb{B}} \\
op_{or}(b_1 : \mathcal{R}_{\alpha\mapsto\!\!\!\rightarrow\mathbb{B}}, b_2 : \mathcal{R}_{\alpha\mapsto\!\!\!\rightarrow\mathbb{B}}) & = & \{\!\!\{(h, b_1(h) \vee b_2(h)) \mid (h, x) \leftarrow b_1\}\!\!\} : \mathcal{R}_{\alpha\mapsto\!\!\!\rightarrow\mathbb{B}}
\end{array}$$

We also define a column-wise equality operator on primitive values as the primary sources of booleans:

$$op_{eq}(f_1 : \mathcal{R}_{\alpha\mapsto\!\!\!\rightarrow\beta}, f_2 : \mathcal{R}_{\alpha\mapsto\!\!\!\rightarrow\beta}) \quad = \quad \{\!\!\{(h, f_1(h) = f_2(h)) \mid (h, x) \leftarrow f_1\}\!\!\} : \mathcal{R}_{\alpha\mapsto\!\!\!\rightarrow\mathbb{B}}.$$

In paragraph 102 we implement the function $bool2sum : \mathbb{B} \rightarrow 1 + 1$ using two column operators *selecttrue* and *selectfalse* defined such that for every f, there is an indicator p such that

$$\begin{array}{rcl}
selecttrue(f : \mathcal{R}_{\alpha\mapsto\!\!\!\rightarrow\mathbb{B}}) & = & \{\!\!\{(h, \dagger) \mid (h, x) \leftarrow f,\ x \text{ true}\}\!\!\} : \mathcal{R}_{\alpha[\mathrm{p}]\mapsto\!\!\!\rightarrow\mathbf{1}} \\
selectfalse(f : \mathcal{R}_{\alpha\mapsto\!\!\!\rightarrow\mathbb{B}}) & = & \{\!\!\{(h, \dagger) \mid (h, x) \leftarrow f,\ x \text{ false}\}\!\!\} : \mathcal{R}_{\alpha[\overline{\mathrm{p}}]\mapsto\!\!\!\rightarrow\mathbf{1}}
\end{array}$$

The tail type is set to **1** because it gains us a tail-completeness property.

85 The operator *mksum* and friends. We have seen the frame combination operator $+\!\!+$. It constructs the union frame $F_1 +\!\!+ F_2 : \alpha \to A$ out of $F_1 : \alpha[p] \to A$ and $F_2 : \alpha[\overline{p}] \to A$. Every extension that defines a new frame type must implement its version of this operation. For the $pair\langle\rangle$ frame, this is simple: $pair\langle F_1, G_1\rangle +\!\!+ pair\langle F_2, G_2\rangle = pair\langle F_1 +\!\!+ F_2, G_1 +\!\!+ G_2\rangle$. But consider $bag\langle d_1, r_1, F_1\rangle +\!\!+ bag\langle d_2, r_2, F_2\rangle$ with

$$\begin{array}{lcl@{\qquad\qquad}lcl} d_1 & : & \mathcal{R}_{\alpha[p]\to=}, & d_2 & : & \mathcal{R}_{\alpha[\overline{p}]\to=}, \\ r_1 & : & \mathcal{R}_{\alpha[p]-\beta}, & r_2 & : & \mathcal{R}_{\alpha[\overline{p}]-\gamma}, \\ F_1 & : & \beta \to A, & F_2 & : & \gamma \to A, \end{array}$$

Note that because element frames F_1 and F_2 have different domain types β and γ, we cannot simply concatenate them together. We rewrite $bag\langle d_1, r_1, F_1\rangle +\!\!+ bag\langle d_2, r_2, F_2\rangle$ into

$$bag\langle d_1 +\!\!+ d_2, mksum(r1, r2), sumleft(r_1, r_2) * F_1 +\!\!+ sumright(r_1, r_2) * F_2\rangle,$$

where the operator *mksum* is used to create a new common domain type δ. The associated operators *sumleft* and *sumright* translate back from δ to the original types α and β:

$$\begin{array}{rcl} mksum(\mathcal{R}_{\alpha[p]-\beta}, \mathcal{R}_{\alpha[\overline{p}]-\gamma}) & : & \mathcal{R}_{\alpha-\delta} \\ sumleft(\mathcal{R}_{\alpha[p]-\beta}, \mathcal{R}_{\alpha[\overline{p}]-\gamma}) & : & \mathcal{R}_{\delta[q]\mapsto\beta} \\ sumright(\mathcal{R}_{\alpha[p]-\beta}, \mathcal{R}_{\alpha[\overline{p}]-\gamma}) & : & \mathcal{R}_{\delta[\overline{q}]\to\gamma} \\ mksum(r_1, r_2) * sumleft(r_1, r_2) & = & r_1 \\ mksum(r_1, r_2) * sumright(r_1, r_2) & = & r_2 \end{array}$$

86 The operator *mkprod* and friends. In a way, *mkprod* is the dual operation of *mksum*. Where *mksum* introduces a new index space to implement sum types and unions, *mkprod* helps implement product types. Suppose we have a comprehension $Bag[(dn\ d, pn\ p) \mid p \leftarrow persons, d \leftarrow dogs_of\ p]$. Skipping over the details of the translation, it seems reasonable that at the heart of *dogs_of* there is a relation $owns : \mathcal{R}_{Person-Dog}$. Evaluating the query means restricting *owns* to the persons mentioned in *persons*, and then producing a person name and dog name *for each pair* in the restricted relation $owns'$. Therefore, the elements of the result bag are identified by pairs:

$$\{\!\!\{(\text{alice}, \text{fido})_{10,1}, (\text{bob}, \text{spot})_{20,7}, (\text{bob}, \text{rex})_{20,8}\}\!\!\}_{\dagger}.$$

We have omitted the *Person* keys for simplicity when we used this example earlier in chapter 1. That was possible because *owns* and $owns'$ are one-to-many relationships, so the dog key implies the person key. However, in the general case where the relation is many-to-many, both keys are necessary to identify a pair.

The operator *mkprod* is used to introduce an index space γ of $(Person, Dog)$-pairs. As with *mksum*, there are two helper operators that translate back to the original index spaces:

$$\begin{array}{rcl} mkprod(\mathcal{R}_{\alpha-\beta}) & : & \mathcal{R}_{\alpha-\gamma}, \\ prodleft(\mathcal{R}_{\alpha-\beta}) & : & \mathcal{R}_{\gamma\to\alpha}, \\ prodright(\mathcal{R}_{\alpha-\beta}) & : & \mathcal{R}_{\gamma\to\beta}. \\ prodleft(r) * converse(prodright(r)) & = & r. \end{array}$$

In paragraph 79 we have already seen how *mkprod* can be used to implement the distribution function D_{Bag}:

$$D_{\mathsf{Bag}} \circ pair\langle F, bag\langle d, r, G\rangle\rangle = bag\langle d, mkprod(r), pair\langle prodleft(r) * F,\ \ prodright(r) * G\rangle\rangle.$$

87 Example implementation of *mkprod* and *mksum*. Given

r_1

1	a
2	b
2	c

r_2

4	P
5	Q
6	Q

the functions *mksum*, *sumleft* and *sumright* can be implemented as follows:

$mksum(r_1, r_2)$	
1	10
2	11
2	12
4	23
5	24
6	25

$sumleft(r_1, r_2)$	
10	a
11	b
12	c

$sumright(r_1, r_2)$	
23	P
24	Q
25	Q

Similarly, with r as given below, *mkprod*, *prodleft* and *prodright* can be implemented as

r	
1	10
1	20
2	30
3	40
4	40

$mkprod(r)$	
1	100
1	101
2	102
3	103
4	104

$prodleft(r)$	
100	1
101	1
102	2
103	3
104	4

$mkprod(r)$	
100	10
101	20
102	30
103	40
104	40

88 Order-preserving versions. We assume our primitive types, or at least the ones used as index types, to be ordered. The ordering of the element-keys is used in the *List* frame to represent the order of the elements in the list. Because in practice most index types are variants of integers, this is not a problem. However, it necessitates the introduction of special, order-preserving versions of *mkprod*, *mksum* and friends, called *mkprod_ord*, *prodleft_ord*, *prodright_ord*, *mksum_ord*, *sumleft_ord* and *sumright_ord*. By order-preserving we mean that for $mkprod(r)$ the ordering on the new keys of type γ mirrors the lexical ordering on the corresponding (α, β)-pairs in r:

$$x < y \iff (prodleft_ord(r)(x),\ prodright_ord(r)(x)) < (prodleft_ord(r)(y),\ prodright_ord(r)(y)).$$

Similarly, for $mksum(r_1, r_2)$ the ordering of the new δ-keys mirrors that of the β- and the γ-keys whenever appropriate:

$$\forall (d, b), (d', b') \in sumleft(r_1, r_2) \bullet \quad d < d' \iff b < b'$$
$$\forall (d, c), (d', c') \in sumright(r_1, r_2) \bullet \quad d < d' \iff c < c'$$

When implementing list concatenation it is convenient if all elements originally from r_1 sort before those originally from r_2:

$$\forall (d_1, b) \in sumleft(r_1, r_2) \quad \forall (d_2, c) \in sumright(r_1, r_2) \bullet \quad d_1 < d_2.$$

Note that every implementation of *mkprod_ord* also suffices as an implementation of *mkprod*, but not the other way around. This illustrates how bags give the implementation more freedom than lists. Optimization such as the ones in paragraph 56 enable Dodo to transform list queries that do not really use "list-ness" into bag queries.

89 Example for *mksum_ord*. Consider

r_1	
1	a
3	b
5	c

r_2	
2	q
4	p

with $a < b < c$ and $p < q$. Then the order preserving *mksum* can be implemented as

$mksum_ord(r_1, r_2)$	
1	10
2	21
3	11
4	20
5	12

$sumleft_ord(r_1, r_2)$	
10	a
11	b
12	c

$sumright_ord(r_1, r_2)$	
20	p
21	q

4.2 Core frames and rewrite rules

In this section we give frames and rewrite rules for the identity function and functor, for primitive types, sum- and product types. For each frame, we give the type rule and the implementation of $c\, *$, $+\!\!+$ and the *dom* operator as required in section 3.2. We also give pseudo code for an interpretation function *lookup* that creates a string representation of the requested value.

90 The Id functor and *id* function. The Id functor does not have its own frame type. The rewrite rules

$$\begin{aligned} \mathsf{Id}\ h \circ F &= h \circ F \\ D_{\mathsf{Id}} \circ pair\langle F, G\rangle &= pair\langle F, G\rangle \end{aligned}$$

are sufficient to translate any expression to frame form. The identity function has a very simple rewrite rule:

$$id \circ F = F.$$

91 The *none* frame. The $none\langle\rangle$ frame is used to denote a function with an empty domain. As explained in paragraph 72, it is used as a placeholder in other frames. Its type rule is

$$none\langle \mathcal{R}_{\alpha \mapsfrom\!\!\!\mapsto =}\rangle : \alpha \to A$$

for arbitrary A. The *none* frame has one component, an empty domain column. Having a column to copy the type from simplifies the implementation of the domain operator, because *emptycol* requires an "example column" as an argument. There is no sensible interpretation function. It is an error to attempt to retrieve a value from a $none\langle\rangle$ frame.

$$\begin{aligned} \text{dom}\ \ none\langle d\rangle &= d \\ r * none\langle d\rangle &= none\langle emptycol(twin(r))\rangle \\ none\langle d\rangle +\!\!+ F &= F \\ F +\!\!+ none\langle d\rangle &= F \\ lookup(none\langle d\rangle, x) &= \text{ERROR!} \end{aligned}$$

92 The *atom* frame. The frame $atom\langle f\rangle$ is used to denote values of a primitive type. Type rule:

$$atom\langle \mathcal{R}_{\alpha \mapsto \beta}\rangle : \alpha \to \beta$$

Frame rules:

$$\begin{aligned} dom\ \ atom\langle f\rangle &= twin(f) \\ r * atom\langle f\rangle &= atom\langle r * f\rangle \\ atom\langle f1\rangle +\!\!+ atom\langle f2\rangle &= atom\langle f1 +\!\!+ f2\rangle \\ lookup(atom\langle f\rangle, x) &= f(x) \end{aligned}$$

93 The *pair* frame. The $pair\langle\rangle$ frame denotes pairs. Type rule:

$$pair\langle \alpha \to A, \alpha \to B\rangle : \alpha \to (A \times B)$$

Frame rules:

$$\begin{aligned} \text{dom}\ \ pair\langle F, G\rangle &= domF = domG \\ r * pair\langle F, G\rangle &= pair\langle r * F, r * G\rangle \\ pair\langle F_1, G_1\rangle +\!\!+ pair\langle F_2, G_2\rangle &= pair\langle F_1 +\!\!+ F_2, G_1 +\!\!+ G_2\rangle \\ lookup(pair\langle F, G\rangle, x) &= {}'(' \ lookup(F, x)\ ',' \ lookup(G, x)\ ')' \end{aligned}$$

Additional pair operations from paragraph 74: this section:

$$\begin{aligned}
exl \circ pair\langle F, G\rangle &= F \\
exr \circ pair\langle F, G\rangle &= G \\
onl\ \ h \circ pair\langle F, G\rangle &= pair\langle h \circ F, G\rangle \\
onr\ \ h \circ pair\langle F, G\rangle &= pair\langle F, h \circ G\rangle \\
D_{onl} \circ pair\langle H, pair\langle F, G\rangle\rangle &= pair\langle pair\langle H, F\rangle, G\rangle \\
D_{onr} \circ pair\langle H, pair\langle F, G\rangle\rangle &= pair\langle F, pair\langle H, G\rangle\rangle
\end{aligned}$$

94 The *either* frame. The $either\langle F, G\rangle$ frame denotes a sum type. The idea is that the key exists either in F or in G. Type rule:

$$either\langle \alpha_{(0)} \to A, \alpha_{(1)} \to B\rangle : \alpha \to (A + B).$$

Sum frame rules:

$$\begin{aligned}
\text{dom}\ \ either\langle F, G\rangle &= domF +\!\!+\ domG \\
r * either\langle F, G\rangle &= either\langle r * F, r * G\rangle \\
either\langle F_1, G_1\rangle +\!\!+\ either\langle F_2, G_2\rangle &= either\langle F_1 +\!\!+\ F_2, G_1 +\!\!+\ G_2\rangle \\
lookup(either\langle F, G\rangle, x) &= lookup(F, x) \quad \text{if } x \text{ in dom} F, \\
lookup(either\langle F, G\rangle, x) &= lookup(G, x) \quad \text{if } x \text{ in dom} G.
\end{aligned}$$

Additional sum type operations from paragraph 74

$$\begin{aligned}
inl \circ F &= either\langle F, none\langle emptycol(\text{dom}\ \ F)\rangle\rangle \\
inr \circ F &= either\langle none\langle emptycol(\text{dom}\ \ F)\rangle, F\rangle \\
ifl\ \ h \circ either\langle F, G\rangle &= either\langle h \circ F, G\rangle \\
ifr\ \ h \circ either\langle F, G\rangle &= either\langle F, h \circ G\rangle \\
D_{ifl} \circ pair\langle H, either\langle F, G\rangle\rangle &= either\langle pair\langle H, F\rangle, G\rangle \\
D_{ifr} \circ pair\langle H, either\langle F, G\rangle\rangle &= either\langle F, pair\langle H, G\rangle\rangle
\end{aligned}$$

4.3 Lists, bags, etc.

In this section we give implementations for the other types and functions mentioned in the introduction of this chapter.

95 The *bag* frame. The $bag\langle d, r, F\rangle$ frame is used to denote bags. Type rule:

$$bag\langle \mathcal{R}_{\alpha \rightarrowtail\!\!\!\twoheadrightarrow =}, \mathcal{R}_{\alpha - \beta}, \beta \to A\rangle : \alpha \to \mathsf{Bag} A$$

The column r gives the relation between the outer keys, which identify bags, and the inner keys, which identify elements in the bag. If a bag identified by $a :: \alpha$ happens to be empty, it contains no elements, and therefore a does not occur in r. To ensure that we are able to recover the domain of the frame, column d keeps track of the keys of all bags in the frame.

$$\begin{aligned}
\text{dom}\ \ bag\langle d, r, F\rangle &= d \\
r' * bag\langle d, r, F\rangle &= bag\langle twin(r' * d), r' * r, F\rangle \\
bag\langle d_1, r_1, F_1\rangle +\!\!+\ bag\langle d_2, r_2, F_2\rangle &= bag\langle d_1 +\!\!+\ d_2, mksum(r_1, r_2), sumleft(r_1, r_2) * F_1 +\!\!+\ sumright(r_1, r_2) * F_2\rangle \\
lookup(bag\langle d, r, F\rangle, x) &= '\{\!|'\ \ (\textbf{for}\ y\ \textbf{in}\ r(x) :\ \ lookup(F, y)\ ','\ )\ \ '|\!\}'
\end{aligned}$$

The frame union operator uses *mksum* to coerce F_1 and F_2 to the same type in order to combine the elements into a single frame. The *lookup* function is written informally. It means: look up the relational image of x in r. Look up in F each y of the relational image. Intersperse the results with commas and put everything betweeen bag-brackets.

Our name for the initial algebra for bags is *bag*. For the *Bag* monad we define

$$\begin{array}{rcl} \mathit{zero}_{\mathsf{Bag}} \circ F & = & \mathit{bag}\langle \mathrm{dom}\, F, \mathit{emptycol}(\mathrm{dom}\, F), \mathit{none}\langle \mathit{emptycol}(\mathrm{dom}\, F)\rangle\rangle \\ \mathit{unit}_{\mathsf{Bag}} \circ F & = & \mathit{bag}\langle \mathrm{dom}\, F, \mathrm{dom}\, F, F\rangle \\ \mathit{unnest}_{\mathsf{Bag}} \circ \mathit{bag}\langle d_1, r_1, \mathit{bag}\langle d_2, r_2, F\rangle\rangle & = & \mathit{bag}\langle d_1, r_1 * r_2, F\rangle \\ \mathsf{Bag}\, f \circ \mathit{bag}\langle d, r, F\rangle & = & \mathit{bag}\langle d, r, f \circ F\rangle \\ D_{\mathsf{Bag}} \circ \mathit{pair}\langle F, \mathit{bag}\langle d, r, G\rangle\rangle & = & \mathit{bag}\langle d, \mathit{mkprod}(r), \mathit{pair}\langle \mathit{prodleft}(r) * F, \quad \mathit{prodright}(r) * G\rangle\rangle \\ \text{Comprehension type } \mathit{Bag} & : & (\mathit{bag}, \mathsf{Bag}, (\mathit{zero}_{\mathsf{Bag}}, \mathit{unit}_{\mathsf{Bag}}, \mathit{unnest}_{\mathsf{Bag}})) \end{array}$$

The *unit* rule takes the frame F and uses it three times in the result. Because $\mathit{unit}_{Bag} \circ F$ has the same domain as F, the result frame gets $\mathrm{dom}\, F$ as its first component. It gets $\mathrm{dom}\, F$ as the outer-inner relation because every bag it constructs (outer) contains precisely one element (inner). We use here that $\mathrm{dom}\, F$ is a binary identity relation. Finally, it uses F itself as the mapping from element keys to elements.

The *zero* rule is very much alike, except that it creates an empty inner-outer relation to signify that all bags in the frame are empty. This version of the rule puts a *none* frame in the third component, but it would be equally valid to just put F there. The only important thing is that F has a type that satisfies the type rule for *bag* frames.

96 The *list* frame. The *list* frame has the same structure as the *bag* frame, but it needs to maintain the ordering of the elements within it. We choose to do so using the native ordering of the element keys, so in the frame $\mathit{list}\langle d, r, F\rangle$ with $r : \mathcal{R}_{\alpha-\beta}$, the ordering of the β keys determines the order of the elements in the list.

Most rewrite rules can simply be copied from the *bag* frame, but frame union needs to use the order-preservering variants of *mksum* and friends. Moreover, where $\mathit{unnest}_{\mathsf{Bag}}$ translates to a simple column semijoin, $\mathit{unnest}_{\mathsf{List}}$ needs to use the order preserving *mkprod_ord* operator to ensure proper ordering of the flattened list. Consider

$$\mathit{unnest}_{List} \circ \mathit{list}\langle d_1, r_1, \mathit{list}\langle d_2, r_2, F\rangle\rangle.$$

Assume the types $r_1 : \mathcal{R}_{\alpha-\beta}$ and $r_2 : \mathcal{R}_{\beta-\gamma}$. The order of the elements in the unnested list is determined by both the β and the γ keys in r_2. The α-keys are irrelevant because they identify result lists, not elements. In the rewrite rule

$$\mathit{unnest}_{List} \circ \mathit{list}\langle d_1, r_1, \mathit{list}\langle d_2, r_2, F\rangle\rangle = \mathit{list}\langle d_1, r_1 * \mathit{mkprod_ord}(r_2), \mathit{prodright_ord}(r_2) * F\rangle,$$

the expression $\mathit{mkprod_ord}(r_2)$ gives a relation between the β-keys and new identifiers for the (β, γ)-pairs in r_2, so $r_1 * \mathit{mkprod_ord}(r_2)$ gives a relation between the outer α-keys and the new element keys. Because we use the $*$*_ord* variants, the new element keys preserve the original ordering. In the third component, we translate the elements in F to the new properly ordered key space.

For completeness the adjusted rules for *list*. Type rule:

$$\mathit{list}\langle \mathcal{R}_{\alpha \leftarrowtail\!\twoheadrightarrow =}, \mathcal{R}_{\alpha-\beta}, \beta \to A\rangle : \alpha \to \mathsf{List} A$$

Frame operations:

$$\begin{array}{rcl} \mathrm{dom}\ \ \mathit{list}\langle d, r, F\rangle & = & d \\ r' * \mathit{list}\langle d, r, F\rangle & = & \mathit{list}\langle \mathit{twin}(r' * d), r' * r, F\rangle \end{array}$$

$$
\begin{aligned}
\mathit{list}\langle d_1, r_1, F_1\rangle +\!\!+\; \mathit{list}\langle d_2, r_2, F_2\rangle &= \mathit{list}\langle d_1 +\!\!+\; d_2, \mathit{mksum_ord}(r_1, r_2), \\
&\qquad \mathit{sumleft_ord}[(r_1, r_2) * F_1 +\!\!+\; \mathit{sumright_ord}(r_1, r_2) * F_2\rangle \\
\mathit{lookup}(\mathit{list}\langle d, r, F\rangle, x) &= {}'[' \;\; (\textbf{for } y \textbf{ in } r(x) : \; \mathit{lookup}(F, y) \; ',' \,) \;\; ']'
\end{aligned}
$$

Our name for the initial algebra for bags is *list*. For the *List* monad we define

$$
\begin{aligned}
\mathit{zero}_{\mathsf{List}} \circ F &= \mathit{list}\langle \mathrm{dom}\, F, \mathit{emptycol}(\mathrm{dom}\, F), \mathit{none}\langle \mathit{emptycol}(\mathrm{dom}\, F)\rangle\rangle \\
\mathit{unit}_{\mathsf{List}} \circ F &= \mathit{list}\langle \mathrm{dom}\, F, \mathrm{dom}\, F, F\rangle \\
\mathit{unnest}_{\mathsf{List}} \circ \mathit{list}\langle d_1, r_1, \mathit{list}\langle d_2, r_2, F\rangle\rangle &= \mathit{list}\langle d_1, r_1 * r_2, F\rangle \\
\mathsf{List}\; f \circ \mathit{list}\langle d, r, F\rangle &= \mathit{list}\langle d, r, f \circ F\rangle \\
D_{\mathsf{List}} \circ \mathit{pair}\langle F, \mathit{list}\langle d, r, G\rangle\rangle &= \mathit{list}\langle d, \mathit{mkprod_ord}(r), \mathit{pair}\langle \mathit{prodleft_ord}(r) * F, \;\; \mathit{prodright_ord}(r) * G\rangle\rangle \\
\text{Comprehension type } \mathit{List} &: (\mathit{list}, \mathsf{List}, (\mathit{zero}_{\mathsf{List}}, \mathit{unit}_{\mathsf{List}}, \mathit{unnest}_{\mathsf{List}}))
\end{aligned}
$$

97 Constant functions. The constant function constructor *const* from paragraph 70 is used for two purposes. The first is to introduce literals into the point-free form, i.e., the simple query 3 has point-free form *const* 3. The other use is to introduce literals from the Dodo schema. For instance, the point-free expression Q' in paragraph 15 contain the subexpression *const nesteddogs*. In paragraph 14, *nesteddogs* is defined as

$$\mathit{nesteddogs} : \mathsf{Bag\ Bag}\ \mathit{Dog} := (\mathit{bag}\langle d_1, r_1, \mathit{bag}\langle d_2, r_2, \mathit{atom}\langle d_3\rangle\rangle\rangle) \;\; \dagger\, .$$

If the schema defines $d = F\ \dagger$ where F is a frame of type $\mathsf{1} \to A$, and if $G : \alpha \to B$, then the expression $\mathit{const}\ d \circ G$ can be translated as follows:

$$\mathit{const}\ \ d \circ G = \mathit{settail}(\mathrm{dom}\ \ G, \dagger) * F.$$

What happens here is that $\mathit{settail}(\mathrm{dom}\ \ G, \dagger)$ creates a column that maps every key in α (the domain of G) to †. This column is then used to transform F from domain $\mathsf{1}$ to domain α. In the case of literals, there is a more direct way:

$$\mathit{const}\ 3 \circ G = \mathit{atom}\langle \mathit{settail}(\mathrm{dom}\ \ G, 3)\rangle.$$

98 The list-to-bag function *lb*. Switching from lists to frames is just a matter of changing the frame name.

$$
\begin{aligned}
\mathit{lb} &: \mathsf{List} A \to \mathsf{Bag} A \\
\mathit{lb} &: \mathit{list} \to_{\mathsf{INS}} \mathit{bag} \\
\mathit{lb} \circ \mathit{list}\langle d, r, F\rangle &= \mathit{bag}\langle d, r, F\rangle.
\end{aligned}
$$

The first entry gives the *type* of *lb*. The second its place in the homomorphism graph. The third its rewrite rule.

99 The *sum* function. The sum function has type $\mathsf{Bag}\mathbb{Z} \to \mathbb{Z}$. That means that its rewrite rule is of the form

$$\mathit{sum} \circ \mathit{bag}\langle d, r, \mathit{atom}\langle f\rangle\rangle = \mathit{atom}\langle XX\rangle.$$

Assuming types $d : \mathcal{R}_{\alpha \twoheadleftarrow\!\!\rightarrowtail =}$, $r : \mathcal{R}_{\alpha - \beta}$ and $f : \mathcal{R}_{\beta \mapsto \mathbb{Z}}$, the mystery column XX must have type $\alpha \to \mathbb{Z}$ and map each $a \in \alpha$ to the sum of the numbers in the relational image $(r * f)(a)$. We take $XX = \mathit{op}_{\mathit{sum}}(d, r * f)$ where $\mathit{op}_{\mathit{sum}}$ is the column operator of type $\mathit{op}_{\mathit{sum}}(\mathcal{R}_{\alpha \twoheadleftarrow\!\!\rightarrowtail =}, \mathcal{R}_{\alpha \vdash \mathbb{Z}}) : \mathcal{R}_{\alpha \mapsto \mathbb{Z}}$ that sums the tails of its second argument while grouping the heads by its first argument. Due to the possibility that $(r * f)(a) = \mathit{emptyset}$, the translation not only depends on $r * f$ but also

on d. Because of this, implementing op_{sum} in SQL involves outer joins and special handling of the resulting NULL-values.

A general pattern in Dodo is that aggregate functions in the nested-structure language are translated to grouping aggregates in the underlying database system. If the head-type is 1, the grouping could be omitted, but it is uncertain whether this is a worthwhile optimization.

The function *sum* is a homomorphism from *bag* to the special algebra $\underline{0} \triangledown (+)$. In the prototype, the latter is called *sum*. The comprehension type below allows us to write things such as $Sum[x^2 \mid x \leftarrow some_bag_or_list]$.

$$
\begin{aligned}
sum &: \mathsf{List}\mathbb{Z} \to \mathbb{Z} \\
sum &: bag \to_{\mathsf{INS}} sum \\
sum \circ bag\langle d, r, atom\langle f \rangle\rangle &= atom\langle op_{sum}(d, r * f)\rangle \\
\text{Comprehension type } Sum &: (sum, \mathsf{Id}, (zero_{sum}, unit_{sum}, unnest_{sum})) \\
zero_{sum} &= const\ 0 \\
unit_{sum} &= id \\
unnest_{sum} &= id
\end{aligned}
$$

100 The functions *any* and *all*. The functions *any* and *all* are entirely similar to *sum*, except that they calculate the logical ($\vee$) with unit element *false* and the logical ($\wedge$) with unit element *true*, respectively. Their rewrite rules assume suitable grouping aggregate operations to exist at the column level.

Because of the way they are typically used in comprehensions, the corresponding comprehension types are named *Exists* and *Forall* rather than *Any* and *All*.

$$
\begin{aligned}
any &: \mathsf{Bag}\mathbb{B} \to \mathbb{B} \\
any &: bag \to_{\mathsf{INS}} any \\
any \circ bag\langle d, r, atom\langle f \rangle\rangle &= atom\langle op_{any}(d, r * f)\rangle \\
\text{Comprehension type } Exists &: (any, \mathsf{Id}, (zero_{any}, unit_{any}, unnest_{any})) \\
zero_{any} &= const\ false \\
unit_{any} &= id \\
unnest_{any} &= id
\end{aligned}
$$

$$
\begin{aligned}
all &: \mathsf{Bag}\mathbb{B} \to \mathbb{B} \\
all &: bag \to_{\mathsf{INS}} all \\
all \circ bag\langle d, r, atom\langle f \rangle\rangle &= atom\langle op_{all}(d, r * f)\rangle \\
\text{Comprehension type } Forall &: (all, \mathsf{Id}, (zero_{all}, unit_{all}, unnest_{all})) \\
zero_{all} &= const\ true \\
unit_{all} &= id \\
unnest_{all} &= id
\end{aligned}
$$

Note that if the prototype would have a Set type, the *all* and *any* functions would be homomorphisms from *set* instead of *bag*.

101 The *to_str* function. The function *to_str* maps integers to their decimal string representation. It is not a homomorphism between any algebras Dodo is aware of.

$$
\begin{aligned}
to_str &: \mathbb{Z} \to \mathbb{S} \\
to_str \circ atom\langle f \rangle &= atom\langle op_{to_str}(f)\rangle
\end{aligned}
$$

102 Boolean operations. Booleans can very well be implemented as sum types $1 + 1$, but in the prototype we will simply implement them using an underlying boolean type, which can either be a proper two-valued boolean type or something in the C style of integers where zero represents false and the other values represent true. We abstract from this choice by introducing *selecttrue* and *selectfalse* column operators.

$$\begin{array}{rcl} not & : & \mathbb{B} \to \mathbb{B} \\ and & : & \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\ or & : & \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\ bool2sum & : & \mathbb{B} \to 1 + 1 \\ not \circ atom\langle b \rangle & = & atom\langle op_{not}(b)\rangle \\ and \circ pair\langle atom\langle b\rangle, atom\langle b'\rangle\rangle & = & atom\langle op_{and}(b, b')\rangle \\ or \circ pair\langle atom\langle b\rangle, atom\langle b'\rangle\rangle & = & atom\langle op_{or}(b, b')\rangle \\ bool2sum \circ atom\langle b\rangle & = & either\langle selecttrue(b), selectfalse(b)\rangle \end{array}$$

103 Equality *eq* for primitive types. Comparing primitive values is left to a suitable column operator. Its implementation depends on the arguments type and on the way booleans are implemented.

$$eq \circ pair\langle atom\langle f\rangle, atom\langle g\rangle\rangle = atom\langle op_{eq}(f, g)\rangle.$$

104 Equality for product types. Pairs are equal if their left components are equal and their right components are equal. This translates into the following rewrite rule:

$$eq \circ pair\langle pair\langle F_1, G_1\rangle, pair\langle F_2, G_2\rangle\rangle = and \circ pair\langle eq \circ pair\langle F_1, F_2\rangle, eq \circ pair\langle G_1, G_2\rangle\rangle.$$

105 Equality for sum types. Equality for sum types is easy to understand but more complex to implement than for product types. Two sum values are *not* equal if their left-, right-handedness differs, or if they are equally handed but their encapsulated value differ. The rewrite rule separates the four cases left/left, left/right, right/left and right/right.

$$eq \circ pair\langle either\langle F_1, G_1\rangle, either\langle F_2, G_2\rangle\rangle = LL +\!\!+ LR +\!\!+ RL +\!\!+ RR$$

where

$$\begin{array}{rcl} LL & = & eq \circ pair\langle intersect(\mathrm{dom}\, F_1, \mathrm{dom}\, G_1) * F_1, intersect(\mathrm{dom}\, F_1, \mathrm{dom}\, G_1) * G_1\rangle \\ LR & = & atom\langle settail(intersect(\mathrm{dom}\, F_1, G_2), false)\rangle \\ RL & = & atom\langle settail(intersect(\mathrm{dom}\, F_2, G_1), false)\rangle \\ RR & = & eq \circ pair\langle intersect(\mathrm{dom}\, F_2, \mathrm{dom}\, G_2) * F_2, intersect(\mathrm{dom}\, F_2, \mathrm{dom}\, G_2) * G_2\rangle \end{array}$$

Explicitly calculating all these intersections is necessary because we work in a function based representation, so *eq* expects its arguments to have exactly the same domain. It is interesting to explore the possibility of a higher-order function eq' that calculates equality only on the intersection of its arguments' domains, together with a separate operator that helps insert the default value *false* where the domains to not overlap. The eq' operator for primitive types maps very nicely to operations supported by most platforms:

```
select x.h, x.v = y.v
from x, y
where x.h = y.h
```

Optimizing equality on sum types is worthwhile because in, e.g., XML applications, deeply nested sum types will be very common.

106 Equality for lists and bags. Equality testing for lists and bags is doable but will not be very efficient. In the prototype we omit implementations for the *eq* operator on lists and bags.

4.4 Rewriting example

In paragraph 15 we presented a sample query and also its translation to point-free form. Later, in section 3.3 we have seen how the translation to point-free form came about. In this section we show how the expression was subsequently rewritten to the frame form displayed in paragraph 17. To keep the example readable, some steps use laws on column operators that in the present report are only hinted at as possible optimizations.

$$
\begin{aligned}
& \mathsf{Bag}(\mathit{onl}\ \mathit{dn} \circ \mathit{onr}\ \mathit{da} \circ \Delta) \circ \mathit{unnest}_{\mathsf{Bag}} \circ \mathit{const}\ \mathit{nesteddogs} \circ \mathit{atom}\langle \mathit{idunit} \rangle \\
= \quad & \{ \text{ definition nesteddogs in paragraph 14, rule for } \mathit{const} \ \} \\
& \mathsf{Bag}(\mathit{onl}\ \mathit{dn} \circ \mathit{onr}\ \mathit{da} \circ \Delta) \circ \mathit{unnest}_{\mathsf{Bag}} \circ ((\mathrm{dom}\ \ \mathit{atom}\langle \mathit{idunit} \rangle) * \mathit{bag}\langle d_1, r_1, \mathit{bag}\langle d_2, r_2, \mathit{atom}\langle d_3 \rangle\rangle\rangle) \\
= \quad & \{ \text{ dom } \ \mathit{atom} \text{ in parg 92, } \mathit{twin}(\mathit{idunit}) = \mathit{idunit} \ \} \\
& \mathsf{Bag}(\mathit{onl}\ \mathit{dn} \circ \mathit{onr}\ \mathit{da} \circ \Delta) \circ \mathit{unnest}_{\mathsf{Bag}} \circ (\mathit{idunit} * \mathit{bag}\langle d_1, r_1, \mathit{bag}\langle d_2, r_2, \mathit{atom}\langle d_3 \rangle\rangle\rangle) \\
= \quad & \{ * \text{ for bags, join with complete bin. id. rel changes nothing } \} \\
& \mathsf{Bag}(\mathit{onl}\ \mathit{dn} \circ \mathit{onr}\ \mathit{da} \circ \Delta) \circ \mathit{unnest}_{\mathsf{Bag}} \circ \mathit{bag}\langle d_1, r_1, \mathit{bag}\langle d_2, r_2, \mathit{atom}\langle d_3 \rangle\rangle\rangle \\
= \quad & \{ \text{ definition } \mathit{unnest}_{\mathsf{Bag}} \text{ in paragraph 95 } \} \\
& \mathsf{Bag}(\mathit{onl}\ \mathit{dn} \circ \mathit{onr}\ \mathit{da} \circ \Delta) \circ \mathit{bag}\langle d_1, r_1 * r_2, \mathit{atom}\langle d_3 \rangle\rangle \\
= \quad & \{ \text{ definition } \mathsf{Bag} \text{ functor } \} \\
& \mathit{bag}\langle d_1, r_1 * r_2, \mathit{onl}\ \mathit{dn} \circ \mathit{onr}\ \mathit{da} \circ \Delta \circ \mathit{atom}\langle d_3 \rangle\rangle \\
= \quad & \{ \text{ definition } \Delta \text{ in paragraph 93 } \} \\
& \mathit{bag}\langle d_1, r_1 * r_2, \mathit{onl}\ \mathit{dn} \circ \mathit{onr}\ \mathit{da} \circ \mathit{pair}\langle \mathit{atom}\langle d_3 \rangle, \mathit{atom}\langle d_3 \rangle\rangle\rangle \\
= \quad & \{ \text{ definition } \mathit{onl} \text{ and } \mathit{onr} \ \} \\
& \mathit{bag}\langle d_1, r_1 * r_2, \mathit{pair}\langle \mathit{dn} * \mathit{atom}\langle d_3 \rangle, \mathit{da} * \mathit{atom}\langle d_3 \rangle\rangle\rangle \\
= \quad & \{ \text{ schema in paragraph 14 } \} \\
& \mathit{bag}\langle d_1, r_1 * r_2, \mathit{pair}\langle \mathit{atom}\langle d_3 * f \rangle, \mathit{atom}\langle d_3 * g \rangle\rangle\rangle
\end{aligned}
$$

In this calculation, the identifier *idunit* was defined by the Dodo core system, and d_1, r_1, d_2, r_2, r_3, f and g by Dodo schema 14.

Chapter 5

Issues

In this chapter we we state several remarks on the material in the preceding chapters. Although the distinction is not always very clear, we separate them them in two classes: General Remarks and Future Work.

5.1 General Remarks

107 Monads with plus and zero. We defined *monads* and *monads with zero*. In the literature one finds also *monads with plus and zero*. In the case of lists, the *plus* operation takes two lists and returns the concatenation. It is related to *zero* in that *zero* gives the neutral element of *plus*. We have seen *plus* under the name *concat* in the definition of *unnest* (paragraph 52).

A monad with plus and zero comes with some useful laws about unnest, zero and plus, which would be very useful for optimizing Dodo queries. Extending the current formalism with *plus* seems very straightforward.

108 Columns may not be sets of pairs. In this report we have treated columns as sets of pairs and suggested that columns map conveniently onto MonetDB BATs. This is not entirely true. In the MonetDB documentation it is reluctantly admitted that BATs in fact behave as bags (multisets) of pairs, not sets. And because of operations like *sort* in the BAT-algebra, it would in fact be even more accurate to say that BATs have list semantics. A similar thing happens in SQL, being the most prominent though not the most faithful implementation of the relational model. The SQL `select` statement contains an `order by` clause that explicitly introduces an ordering. In order to support this clause, the underlying algebra needs to have some notion of ordering defined in it.

One approach is to define the underlying algebra not in terms of sets of pairs but in terms of lists of pairs, with the proviso that the result of most of the operators is only defined up to element reordering and duplication. In other words, let the data type underlying the algebra be as detailed as possible (lists), but let most operators have set-semantics. In paragraph 38 (algebras with equations) we touched upon a particularly elegant way of expressing this. Note that by "operators with set-semantics" we do not mean that after every operation, duplicates are removed. Rather, we mean that the operators are indifferent to the order of elements within the list and the existence of duplicates. Making this explicit allows one to reason more easily about where duplicate elimination needs and does not need to be performed.

Note that we are not trying to throw the relational model out the window. Set semantics are very natural for writing a database schema and the semantics of operators. It is just that at the lower levels of query processing, and on input/output, it is beneficial to acknowledge that the underlying data storage has more structure than just set structure.

Considering the above, and the fact that both MonetDB and SQL essentially provide bag semantics, we believe that it would be beneficial to expose the bag structure in the column layer

of Dodo. This would imply providing several variants of operators that preserve different amounts of structure. A prominent example would be the join operator $*$ which could come in a variation that preserves the precise number of duplicates, and another variant that is allowed not to preserve this. The frame definitions know which assumptions they make about the columns in the frames, and are therefore able to pick the proper operators. Note that from the users point of view, set semantics are provided by a *Set* type former, regardless of the semantics of the underlying columns.

We do not propose columns to have list semantics. Having list semantics only makes sense if there are enough operations that preserve it, and in MonetDB, even though the order of elements in a BAT is quite visible, few operations make any guarantees about it. It also seems that having access to the order of the pairs in the column does not open up many possibilities for optimizations, so overall, list semantics are not worth the effort.

It is important not to confuse the issue of order as in list semantics for columns with the order that is preserved by the order preserving operations *mkprod_ord*, *prodleft_ord*, *prodright_ord*, *mksum_ord*, *sumleft_ord* and *sumright_ord* defined in paragraph 88, which preserve the relative ordering of tail values with respect to corresponding head values.

109 Synchronized BATs. Synchronized BATs are an important issue in MonetDB. They also emphasize that BATs are essentially list-based. Two BATs are said to be synchronized if their head-attributes contain the same values *in the same order.* Knowing that two BATs are synchronized allows huge optimizations. If, for instance, r_1 and r_2 are synchronized, the head-wise equality operator

$$[=](r_1, r_2) := \{(h, t_1 = t_2) \mid (h, t_1) \in r_1, (h, t_2) \in r_2\}$$

can be implemented using a simple scan through r_1 and r_2, instead of having to search in r_2 for every head in r_1.

MonetDB contains a mechanism to track whether columns are synchronized, automatically picking optimized variants of operators when possible. There is also a built-in operator that forces synchronization. It seems that for the kind of queries Dodo produces, most resulting BATs will be synchronized already. If this turns out not to be the case, it might be an option to track synchronization also within Dodo, perhaps in the type system.

110 Zip. Of the well known functional programming power tools, Dodo has *map* in the form of type functors, it has *fold* in the form of catamorphisms, but it has no *zip*. The *zip* function on lists has type $\mathsf{List}A \times \mathsf{List}B \rightarrow \mathsf{List}(A \times B)$, mapping the pair of lists $([1, 2, 3], [a, b, c])$ to $[(1, a), (2, b), (3, c)]$. Essentially, it exchanges the List and $\times$ type formers. I have not encountered instances of *zip* in the example Moa queries. This is not very surprising, as zipping tends to be position-based and database queries tend to avoid lists in favor of bags and sets, which have no notion of "position." In an XML processing, lists do occur, but zipping is not an important operation in that context.

111 Name clashes. One practical problem with our approach is name clashes. Consider the identifier *sum*. It could be used as $sum : \mathsf{Bag}\mathbb{Z} \rightarrow \mathbb{Z}$ in the nested-structure layer, $sum(\mathcal{R}_{\alpha-\mathbb{Z}}) : \mathcal{R}_{\alpha\rightarrow\mathbb{Z}}$ in the column layer, and probably also as the name of the summation operator of the underlying database system. And then we not have counted yet its use as the name of the *sum*-algebra $\underline{0} \triangledown (+)$ and the comprehension type $Sum[\ldots \mid \ldots]$.

There are two approaches to this problem. The first is to use the very large name space of the Dodo prototype: *sum*, op_{sum}, etc. The other is to maintain separate namespaces for all uses. We already did this a little by using the name *sum* as the algebra name. We also prefixed the column names in paragraph 73 with an exclamation mark, and this treatment might be extended to the column operators.

112 Column Type Notation. The column type notation is not strong enough to easily capture all semantics of the column operations. In particular, it supports type variables α and β

as placeholders for types, but it has no placeholders for properties. One possibility would be to declare the propagation of each property separately. For instance, the behaviour of the semijoin operator $*$ could be expressed as

$$\begin{array}{rcl} \mathcal{R}_{\alpha-\beta} * \mathcal{R}_{\beta-\gamma} & : & \mathcal{R}_{\alpha-\gamma} \\ \mathcal{R}_{\alpha\rightarrow\beta} * \mathcal{R}_{\beta\rightarrow\gamma} & : & \mathcal{R}_{\alpha\rightarrow\gamma} \\ \mathcal{R}_{\alpha\leftarrow\beta} * \mathcal{R}_{\beta\leftarrow\gamma} & : & \mathcal{R}_{\alpha\leftarrow\gamma} \\ \mathcal{R}_{\alpha\vdash\beta} * \mathcal{R}_{\beta\vdash\gamma} & : & \mathcal{R}_{\alpha\vdash\gamma} \\ \mathcal{R}_{\alpha\dashv\beta} * \mathcal{R}_{\beta\dashv\gamma} & : & \mathcal{R}_{\alpha\dashv\gamma} \end{array}$$

113 Optimizations in the frame/column language. Many rewrite rules depend on rewrite rules at the column level to optimize away inefficiencies they introduce. We already saw in paragraph 17 the propagation of *sethead* through $*$. A more detailed demonstration can be found in 4.4, where several column-level identities are used. In the latter case, the purpose of the optimizations is not efficiency; they are applied simply to make the intermediate expressions fit on one line.

The majority of these optimization rules is very simple; we expect to be able to write down dozens of them at the moment we look at the first column expressions generated by the Dodo prototype. Usually the rules depend on the completeness and uniqueness information incorporated in the type system (paragraph 63). For instance, the equation

$$\mathit{sethead}(d_2 * r_2, \dagger) = \mathit{sethead}(r_2, \dagger)$$

from paragraph 17 holds only if d_2 is tail-complete. If d_2 is not tail-complete, joining r_2 with d_2 may filter out some tuples. If we assume bag semantics for columns as in paragraph 108, d_2 must also be tail-unique, otherwise the number of duplicates may not be correct.

114 Booleans and equality. There are two ways of implementing booleans. In the prototype we just use the underlying boolean or integer type, taking 0 as false and everything else as true. Logic operations are simple streaming BAT operations. This has the advantage that it is easy to understand and fits well with the typical back-end system. Another way of implementing booleans is analogous to the sum type. Left-handed is true and right-handed is false. The operations *not* and *bool2sum* are trivial, and the other logical operations are implemented using intersections and unions.

A common optimization is to "short circuit" boolean operations, that is, in evaluating $p \wedge q$ the term q is not evaluated if p yielded *false*. In Dodo, the expression $\lambda x \bullet p \wedge q : A \to \mathbb{B}$ is translated into a frame expression, and p and q are both computed in one go for all $x$es in the domain. For implementing the short circuit optimization, sum type booleans have the advantage that after $\lambda x \bullet p$ has been evaluated, the subset of A for which p is true has already been determined and is ready to have $(\lambda x \bullet q)$ applied to it.

115 Operator Overloading. In section 4, we implemented the equality operator by giving rewrite rules for $eq \circ F$ for various frames F. The question is, which type do we declare for eq in the nested structure layer? One option would be to declare a separate equality operator for every type, i.e., *eqAtom*, *eqPair*, etc. This is not really a workable solution. Another simple solution is to just declare $eq : A \times A \to \mathbb{B}$ and have the rewrite process crash or get stuck if eq turns out to be applied to a type it is not implemented on. This is a real possibility because the prototype does not implement equality for lists and bags.

In the prototype we essentially employ the second option, but with an added safeguard. Type checking takes place in three phases. In the first phase, Dodo walks the expression tree and collects type equations. In the second phase, it computes for each node the most general type that satisfies these equations. In the third phase, it makes a copy of the tree where each node is assigned its type. The trick is that in the last phase, a node is allowed to refuse its assigned type and give an error message, such as "equality for this type." This verification phase is also used to check

whether catamorphisms are well-defined; if τ is the list algebra then the catamorphism $(\!|\tau|\!)$ must not be applied to a value that during unification turned out to be a bag.

116 Eliminating cartesian products. Consider a database containing

$$\begin{aligned} dogs &= bag\langle idunit, r_d, atom\langle f_d\rangle\rangle : 1 \to Dog \\ persons &= bag\langle idunit, r_p, \langle f_p\rangle\rangle : 1 \to Person \\ owner &= atom\langle r_o\rangle : Dog \to Person \end{aligned}$$

In this database, we could ask for

$$Bag[(owner\ \ d, d) \mid d \leftarrow dogs].$$

But suppose we asked for

$$Bag[(p, d) \mid p \leftarrow Persons, d \leftarrow Dogs, owner\ \ d = p].$$

This query first constructs the cartesian product of *Person* and *Dog*, and then filters out everything that does not meet its criterium. It would be very nice if Dodo could implement this filter after a cartesian product using some kind of join operator at the column level. Note that in general this will only be feasible if the filter involves only comparisons of primitive values, as the column algebra typically has no operator available that joins on complex values such as bags.

First consider what the query looks like in point-free form. Using the alternative *FILTER* function from paragraph 78, we can rewrite the expression $Bag[e \mid x \leftarrow xs, y \leftarrow ys, p(x, y)]$ as follows. We write $B = Bag$, $un = unnest_{Bag}$ and $F = FILTER_{Bag}$, and assume e is an expression involving x and y.

$$\begin{aligned} & [\![\lambda w \bullet B[e \mid x \leftarrow xs, y \leftarrow ys, p(x, y)]]\!] \\ = \ & \dots \\ = \ & B\ [\![\lambda z \bullet e]\!] \circ un \circ B\ un \circ B\ B\ F\ p \circ B\ D_B \circ B\ onr\ \underline{ys} \circ B\ \Delta \circ \underline{xs} \end{aligned}$$

In this expression we see the function $\underline{ys} = const\ ys$ applied to every element of xs. If $xs = bag\langle d_1, r_1, F_1\rangle$ and , $ys = bag\langle d_2, r_2, F_2\rangle$, the rewrite rules in chapter 4 will deliver something like

$$settail(\text{dom}\ \ F_1, \dagger) * F_2,$$

effectively materializing the cartesian product.

The *dogs-persons-owner* query from above corresponds to $p = eq \circ (id \times owner)$. Predicates of the form $p = eq \circ (f \times g)$ can be transformed into a join in the following way. If we encounter the query

$$un \circ B\ un \circ B\ B\ F\ (eq \circ (f \times g)) \circ B\ D_B \circ B\ onr\ \underline{ys} \circ B\ \Delta \circ bag\langle d_1, r_1, F_1\rangle, \qquad (*)$$

we can first apply f to the elements in the frame F and g to the elements in frame G. The result will be two *atom* frames $atom\langle\hat{f}\rangle = f \circ F$ and $atom\langle\hat{g}\rangle = g \circ G$ containing elements of the primitive type K.[1] We can then use the standard semijoin operator to quickly construct a column containing precisely those pairs $(a, b) \in A \times B$ for which $f\ \ a = g\ \ b$:

$$r_{eq} = \hat{f} * converse(\hat{g}).$$

Using r_{eq}, we can build an efficient replacement for $(*)$:

$$bag\langle d, mkprod(r_{eq}), pair\langle prodleft(r_{eq}) * F, prodright(r_{eq}) * G\rangle\rangle.$$

In our example, $f = id$ and $g = owner$, so $r_{eq} = id * converse(owner) = converse(owner)$. Given suitable operators at the column level, this approach can be generalized to other operations than equality.

[1] In fact, we should take $atom\langle\hat{f}\rangle = f \circ rtwin(r_1) * F$, because r_1 might not use all elements from F.

117 Equality of complex data types. Implementing *eq* and related functions for types other than sum and product types is harder than it may seem at first sight. For instance, consider two frames $B_1 = bag\langle d, r_1, F_1\rangle$ and $B_2 = bag\langle d, r_2, F_2\rangle$, both of type $\alpha \to \mathsf{Bag}A$. Usually, F_1 and F_2 are indexed on a different key space, i.e., $F_1 : \beta \to A$ and $F_2 : \gamma \to A$. Our task is to construct a frame $F = atom\langle f\rangle : \alpha \to \mathbb{B}$ with the property that for every $a \in \alpha$, the boolean $F\ a$ is true if and only if $B_1\ a = B_2\ a$.

For bag frames we know of no clear solution. Of course, bags whose sizes are unequal are easy to spot, so we concentrate on the case where the sizes match. If we have *list* frames instead of *bag* frames, can use the order to rearrange F_1 and F_2 in such a way that corresponding list elements "line up." This allows us to calculate key-wise equality between the transformed frames and then grouping the resulting "element-booleans" to obtain "list-booleans." For *set*-frames, one can compute for each $a \in \alpha$ all corresponding $\beta \times \gamma$ pairs and then use element-wise equality to determine whether the two sets are proper subsets of each other. Unfortunately, this trick cannot be applied to *bag* frames, because it does not account for the number of times the elements occur in the bags. Two workarounds are: turning the bags into lists by sorting them or turning them into sets by replacing the elements by (element, multiplicity)-pairs.

Unfortunately, these operations are complicated themselves. Sorting the elements involves comparing them using less-than relation $<$, which is not that well-defined and has all the problems equality has. The other approach requires a frame-operator that identifies groups of equal values within a frame F, possibly by constructing a frame $c = atom\langle \tilde{c}\rangle : \alpha \to \alpha'$ with the property that $c\ x = c\ y \iff F\ x = F\ y$. On the bright side, both approaches succeed in turning a difficult problem on *bag* frames into a difficult problem on the element-frame within the *bag*. Moreover, identifying groups of equal elements within a frame is a useful operation for other purposes as well; the most obvious purpose is implementing a *distinct* operator.

Whatever solution we come up with, it will surely not be very efficient. Fortunately, most real-world queries do not often compare bags and sets directly. Typically, they compare properties of an object *enclosing the set*, or properties that are derived *from* the set. For the moment we conclude this subsection stating that determining equality on flattened data structures is a hard problem, especially if one wants to express it in a way that is general enough for both SQL and MonetDB. We expect that implementing equality operators such as the operator that identifies equal values within a frame, the join-like operations used by the optimizations in paragraph 116 and the *eq*-operator will be a serious burden on the extension writer.

5.2 Future Work

118 Array types. The theory behind Dodo is geared towards the kind of data structures that is defined recursively and is operated on by induction to that recursive structure. Array-like types do not fit very well in that frame work. This is unfortunate as such types are commonly used in multimedia applications, which are an important application area for Moa and therefore also Dodo. It would be interesting to gather and review both categorical attempts to capture array semantics as well as array-based programming languages with static typing and see whether their approaches can beneficially be applied to Dodo. Of particular interest is RAM [11], an array language layered on top of MonetDB. Like Dodo, RAM has comprehension syntax, and the handling of multidimensional arrays is similar to the way Dodo handles nesting, except that comprehensions do not flatten automatically. If $a : \mathbb{Z}[3]$ and $b : \mathbb{Z}[2]$, then $[x + y \mid x \leftarrow a, y \leftarrow b]$ is of type $\mathbb{Z}[2][3]$ rather than $\mathbb{Z}[6]$.

119 Fusion theorems. In paragraph 54 we already mentioned that it is interesting to investigate the restrictions one needs to impose on the fusion theorems in order to guarantee that the resulting catamorphisms have a known equivalent sequence of built-in frame transformations.

120 Multiple initial algebras. Let τ be the initial INS-algebra for the List type former and let $E = \{(2.3)\}$ the set of equations that specify that an algebra ignores order. Then τ has

a "shadow" τ' that is initial in $\mathcal{A}lg(\mathsf{INS}, E)$. Informally, this corresponds to the type of lists on whose order cannot be relied. This τ' was used in paragraph 57 to express the fact that *sum* does not care about the order in which its summands are presented, and that *rev* only affects the order of its list argument:

$$\begin{array}{ccc} \rho & \xleftarrow{rev} & \tau \\ & \searrow^{id} & \downarrow{\scriptstyle id} \\ & & \tau' \\ & & \downarrow{\scriptstyle sum} \\ & & \sigma \end{array}$$

On the other hand, the Bag algebra $\beta \in \mathcal{A}lg(\mathsf{INS}, E)$ is also initial. Having two initial algebras for the same equation E is not a problem. It just means that there is a unique homomorphism from β to τ' and vice versa. If we make this explicit by providing homomorphisms from τ' to β and vice versa, Dodo can exploit it. The homomorphism from τ' is the same as from τ: the list-to-bag function *lb*. The homomorphism the other way around is new. It must take a bag, and produce a list. The order in which it puts the elements in the list does not matter, because τ' does not care about it anyway. We shall name this function *arborder*. For most implementations of Bag and List, such a function is easy to create and cheap to execute.

With these homomorphisms in place, Dodo can immediately evaluate queries such as

$$Sum[x|x \leftarrow a_bag],$$

because the required catamorphism $(\!|\beta \to \sigma|\!)$ can be implemented as $(\!|\beta \to \sigma|\!) = sum \circ arborder$. The bag extension can also quickly define $bagsum = sum \circ arborder$. However, if performance can be gained by doing so, the extension can of course also define *bagsum* directly in terms of frames.

What makes all this interesting is that it shows how even with the primitive homomorphism graph, Dodo can combine information from different extensions. The order-free list algebra τ' is used by both the bag extension and the extension to express information about the behaviour of their functions. Dodo uses this to automatically provide *Sum*, *Exists* and *Forall* comprehensions, and in this example also to optimize away *rev* when it is applied to something that is going to be converted to a bag.

Bibliography

[1] P. A. Boncz and M. L. Kersten. Monet: An Impressionist Sketch of an Advanced Database System. In *Proceedings Basque International Workshop on Information Technology*, San Sebastian, Spain, July 1995.

[2] P. A. Boncz, W. Quak, and M. L. Kersten. Monet and its Geographical Extensions: a Novel Approach to High-Performance GIS Processing. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, volume 1057 of *Lecture Notes in Computer Science/Lecture Notes in Artificial Intelligence (LNCS/LNAI), Springer-Verlag*, pages 147–166, Avignon, France, June 1996.

[3] P. A. Boncz, A. N. Wilschut, and M. L. Kersten. Flattening an Object Algebra to Provide Performance. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 568–577, Orlando, FL, USA, February 1998.

[4] Peter Alexander Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, University of Amsterdam, 2002.

[5] A.P. de Vries. *Content and multimedia database management systems*. PhD thesis, University of Twente, Enschede, The Netherlands, December 1999.

[6] Maarten M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, 1992.

[7] T. Grust, M. van Keulen, and J. Teubner. Accelerating XPath Evaluation in Any RDBMS. *Transactions on Database Systems (TODS)*, 29(1):91–131, March 2004.

[8] Torsten Grust. *Comprehending Queries*. PhD thesis, University of Konstanz, September 1999.

[9] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing Database Architecture for the New Bottleneck: Memory Access. *The VLDB Journal*, 9(3):231–246, December 2000. The original publication is available in LINK, Springer-Verlag.

[10] M. Petkovic. *Content-based Video Retrieval Supported by Database Technology*. PhD thesis, University of Twente, February 2003.

[11] Alex R. van Ballegooij, Arjen P. de Vries, and Martin L. Kersten. Ram: Array processing over a relational dbms. Technical report, CWI, 2003.

[12] M. van Keulen, J. Vonk, A.P. de Vries, J. Flokstra, and H.E. Blok. Moa and the multi-model architecture: a new perspective on NF2. In V. Marik, W. Retschitzegger, and O. Stepankova, editors, *Proceedings of the 14th International Conference on Database and Expert Systems Applications (DEXA2003), Prague, Czech Republic*, number 2736 in LNCS, pages 67–76. Springer-Verlag, September 2003.

[13] Maurice van Keulen, Jochem Vonk, Arjen P. de Vries, Jan Flokstra, and Henk Ernst Blok. Moa: extensibility and efficiency in querying nested data. Technical Report TR-CTIT-02-19, Centre for Telematics and Information Technology, University of Twente, The Netherlands, July 2002.