Flattening Queries over Nested Data Types

Joeri van Ruth

Samenstelling van de promotiecommissie:

prof. dr. P.M.G. Apers, promotor

dr. M.M. Fokkinga, assistent promotor
dr. ir. M. van Keulen, referent

prof. dr. H. Brinksma
prof. dr. P. Hartel
prof. dr. T. Grust (TU München)
prof. dr. M.L. Kersten (UvA)

Centre for Telematics and Information Technology (CTIT)
P.O. Box 217, 7500 AE Enschede, The Netherlands

SIKS Dissertation Series No. 2006-11
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Graduate School for Information and Knowledge Systems.

ISBN: 90-9020723-6
ISSN: 1381-3617 (CTIT Ph.D. Thesis Series no. 06-86)

Cover design: Nienke Valkhoff

Print: PrintPartners Ipskamp, Enschede, The Netherlands

Copyright © 2006, Joeri van Ruth, Amsterdam, The Netherlands

FLATTENING QUERIES OVER NESTED DATA TYPES

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
Prof. dr. W.H.M. Zijm,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 2 juni om 15.00 uur

door

Joeri van Ruth

geboren op 7 maart 1974

te Amsterdam

Dit proefschrift is goedgekeurd door

Prof.dr. P.M.G. Apers, promotor.

DANKWOORD

Graag wil ik iedereen bedanken die direct of indirect aan het voltooien van dit proefschrift heeft bijgedragen. Allereerst had ik me geen betere begeleiders kunnen wensen dan Maarten en Maurice. Met hun totaal verschillende blik op de materie vulden ze elkaar perfect aan. Ik ben dankbaar voor alle aandacht en energie die ik gekregen heb, zonder hun steun had ik het nooit gehaald. Ondanks zijn volle agenda is mijn professor, Peter Apers, er altijd geweest als ik hem nodig had. Mijn gesprekken met hem hebben me erg geholpen de big picture in het oog te houden, zowel wetenschappelijk als wat betreft het promotietraject.

Ik heb het in de DB-groep enorm naar mijn zin gehad. In het bijzonder wil ik Sandra en Suze bedanken, samen het hart van DBIS, en alle AIO's van het informele DB AIO-seminarium, de Almost Weekend Meetings. En natuurlijk mijn kamergenoot Ander, voor het prettige gezelschap, het helpen met het vinden van denkfouten op whiteboards en alle hulp bij het regelen van UT-zaken toen ik zelf alweer in Amsterdam woonde. Rick van Rein ging al bijna weg toen ik bij de DB-groep kwam. Gelukkig was ik er nog op tijd bij, en zijn we goede vrienden geworden. Zonder Rick had ik een stuk minder fijne tijd gehad in Twente.

In mijn familie wil ik graag mijn ouders bedanken, die altijd voor me klaar stonden. En mijn oma van Ruth, ook voor haar bijdrage aan het feest. Verder ben ik buitengewoon trots dat Cornelis en Pieter Tol, mijn opa en mijn oom, tijdens de promotieplechtigheid mijn paranimfen zullen zijn. Zij hebben samen aan het begin gestaan van de weg die me uiteindelijk naar dit proefschrift leidde.

Het meest wil ik Nienke bedanken. Zonder haar had ik het nooit gered. Mijn promotie is op veel manieren ten koste van haar gegaan, omdat ik door de week in Twente woonde, en omdat het allemaal steeds maar niet af kwam. Ik ben haar enorm dankbaar dat ze dit met me heeft willen doorstaan en dat ze me al die tijd zo gesteund heeft. Ik kijk uit naar de tijd dat we onze promoties achter de rug hebben en weer aan ons leven samen toe kunnen komen.

Contents

1 Introduction **1**
- 1.1 Goal . . . 2
- 1.2 Approach . . . 3
- 1.3 Dodo and the Multi-model DBMS . . . 8

2 Function-based data model **15**
- 2.1 Dodo . . . 15
- 2.2 Nested data model and sublanguage . . . 17
  - 2.2.1 Types . . . 18
  - 2.2.2 Query language . . . 21
- 2.3 Flat data model and sublanguage . . . 24
- 2.4 Flattening data . . . 26
- 2.5 Flattening queries . . . 29
  - 2.5.1 Point-free form . . . 32
  - 2.5.2 Translation to point-free form . . . 34
  - 2.5.3 Handling nested scopes . . . 37
  - 2.5.4 Further translation . . . 40
- 2.6 Extension writer obligations . . . 41
- 2.7 Summary . . . 43

3 Realization **45**
- 3.1 Basic column operators . . . 46
- 3.2 Key representation . . . 48
- 3.3 Data type definitions . . . 52
- 3.4 The Dodo type system . . . 60
- 3.5 The prototype . . . 66
- 3.6 Summary . . . 67

4 Pathfinder **69**
4.1 Pathfinder in a nutshell . . . 70
4.2 Pathfinder extension to Dodo . . . 77
4.2.1 The $seq\langle\rangle$ frame . . . 78
4.2.2 The $xmlnode\langle\rangle$ frame . . . 82
4.3 Conclusion . . . 87

5 Categorical background **89**
5.1 Algebras, monads and comprehensions . . . 89
5.2 Monads and monad comprehensions . . . 96
5.3 Application in Dodo . . . 99

6 Inductively defined types **103**
6.1 Inductively defined types and catamorphisms . . . 104
6.2 Frame representation for fixpoint types . . . 105
6.3 Rewrite rule for the initial algebra . . . 110
6.4 Implementing catamorphisms by iteration . . . 112
6.5 Implementing iteration in the column layer . . . 118
6.6 Summary . . . 122

7 Summary and future work **125**

Bibliography **131**

Samenvatting **137**

SIKS Dissertation Series **141**

Chapter 1

Introduction

The translation of queries from complex data models to simpler data models is a recurring theme in the construction of efficient data management systems. We propose a general framework to guide the translation from data models with nested types to a flat relational model (query flattening). In this framework, the query is first rewritten into a point-free form. The point-free form is a good starting point for converting nested operations into machine friendly relational bulk operations at the flattened level.

1 Recurring theme. Flattened representations of nested data types are a recurring theme in database research. For instance, Geographical (GIS) applications often involve geometrical structures such as points, lines and polygons. In the MAGNUM project [BWK98] it has been demonstrated that for certain query workloads, decomposition onto a relational back end extended with suitable primitive operations may yield significant performance improvements [BQK96]. A more general example are object oriented databases (OO-DBMS) such as O_2 [BDK92, Clu98], where, for performance, object navigation is implemented using join algorithms. More recently, XML databases have emerged, which to a varying degree use XML as their data model. Again, implementations turn to flat storage representations for performance. For instance, the *Pathfinder* [PTS$^+$05] XQuery engine uses a flat representation of the document structure and implements FLWOR[1] expressions [PTM$^+$05] and XPath traversals [GvKT04] as joins. Finally we note that the non-first-normal-form (NF2) research of the late eighties also struggled with this problem.

The common pattern is that data management systems which need to support more high-level nested data structures turn to relation, or at least flat,

[1] for-let-where-order-by-return

storage to obtain performance. Usually, this involves the use of *join*-operations. Sometimes these are the "classic" joins provided by every commercial relational DBMS, sometimes specific join operators are added. See Mayer, Grust et al. [MGvKT04] for an example of a join algorithm integrated into the PostgreSQL query engine to improve performance on tree retrieval, and Steenhagen [SABdB94] for more general work on *nestjoins*, join algorithms for nested data models. Altogether, there is reason enough to study the phenomenon of query flattening in isolation.

1.1 Goal

2 Query flattening. Query flattening is the process of rewriting a query from a data model with complex, possibly nested data types to a data model with only simpler, non-nested types. The most important example of the latter is of course the relational model in first normal form. The need for query flattening arises in two contexts. Sometimes it is desirable to access existing relational data through a complexly structured view (e.g., an object-oriented or XML-based view) that presents a more convenient data model. Any queries posed to the view have to be translated to queries over the data as it is actually stored.

The other reason is performance. If the storage layout is too similar to the logical organization of the data, complex queries often suffer disappointing performance. This is due to several factors, most notably the unpredictable access patterns caused by excessive object navigation (pointer chasing). As general-purpose hardware becomes more and more dependent on caching, the penalty for random memory access keeps growing [BK99]. It also seems that complex data models tend to encourage item-at-a-time thinking as opposed to the set-at-a-time operations facilitated by the relational model. Expressing the query in terms of whole sets rather than individual items gives the optimizer much more leeway in choosing an appropriate query plan.

3 The Multi-model DBMS *Dodo*. In non-traditional application domains, such as multimedia retrieval and GIS, such performance problems are known to frequently occur, as are extensibility problems. A *multi-model data management architecture* has been shown to be effective in solving both kinds of problems [dV99]. A multi-model data management architecture allows an application-oriented (nested) data model at the upper layer and a machine-oriented (flat) data model in the bottom layer. The system can be extended at each layer independently, and typically extensions at a given layer can reuse functionality provided in the layers below. This facilitates cross-extension optimization.

We explore the issue of query flattening in the context of a system called Dodo. Dodo [vR05] is a continuation of the Moa system, which served as an early prototype of the multi-model DBMS. Dodo shares with its ancestor the translation from application-oriented to machine oriented query algebras and the capability to extend the system at every appropriate layer.

Extensions in current systems are often restricted to defining a byte representation of a new data type, plus the signatures of a collection of operators on this type. In Dodo, and the multi-model architecture in general, extensions have full access to all facilities provided by the system and the other extensions. This allows extensions to be defined at a higher level of abstraction: the storage structure of our new types is defined in bulk form using binary relations as the fundamental building block, rather than a byte representation of individual values. This makes it easier to re-use existing algorithms implemented by the database kernel and allows the optimizer to understand the building blocks used by the new extension. Van Keulen et al [vKVdV⁺03] label this approach an *open implementation* approach rather than a *black box* approach.

4 Problem statement. Establish a systematic mapping from queries over extensible nested data models to queries based on bulk operations on a flat data model, allowing cross-extension optimization.

1.2 Approach

5 Research strategy. We look at category theory for inspiration. The categorical point of view is interesting because on the one hand, there exists a well developed categorical theory of data types. On the other hand, due to the way they are expressed, categorical proofs are already in bulk form (point-free). We focus on analytical query processing, where the query workload is dominated by large, computationally intensive queries. Hence, we focus on query processing and ignore updates.

In this thesis, we propose a framework for flattening queries over nested data types. It employs a point-free intermediate form both as a means of introducing bulk operations and as a convenient platform for structural optimizations. We hope that this general framework can serve as an aid to understanding and designing query flattening strategies.

We validate our approach by examining specific examples of query flattening in the XML domain and showing how the patterns made explicit in our framework are already present there. At the same time, we hope to convince the reader that point-free reasoning indeed simplifies the construction of a correct and consistent query flattening system when compared to developing it in an ad-hoc fashion.

6 Data vs. query flattening. First, we introduce some more terminology around the concept of query flattening. The general pattern of query flattening is illustrated in the following diagram:

$$\begin{array}{ccc} N & \xrightarrow{q} & N \\ \uparrow & & \uparrow \\ F(N) & \xrightarrow{F(q)} & F(N) \end{array} \tag{1.1}$$

Here, N represents the domain of complexly structured ("Nested") data and q represents a query on the nested data. A query is a function that takes the data in the database and transforms it into a return value. As such, it is a mapping from N to N. Under water, the data is stored in a flattened form $F(N)$. The basic idea is to derive a flattened version $F(q)$ of the original query q in such a way that evaluating $F(q)$ against flattened data and then unflattening is equivalent to evaluating q naively at the nested level.

A flattening strategy F has two components. The *schema flattening* $F(N)$ expresses how to decompose complex types at the nested level into simple types at the flattened level. It might, e.g., implement a set-valued attribute using auxiliary relations. The *query flattening* $F(q)$ brings the query from the nested to the flattened level. In the case of set-valued attributes, it generates the extra `join` operations needed to connect the elements of the attribute with their containing object. Notice that we denote both the data- and the query components with the letter F because they are so interrelated. One cannot be designed without having the other in mind.

Notice the direction of the arrow in diagram (1.1). Although $F(N)$ is the flattened representation of N, the arrow is drawn the other way around. The reason is that the arrows denote the flow of information. At query time, the data already resides in the database in flattened form. The diagram shows the property the query flattening $F(q)$ must satisfy. For any query q, executing $F(q)$ and then assembling the flat result into a nested return value should give the same result as first assembling the database back into nested form, and then executing q in its original form.

7 Data flattening. Our proposed query flattening strategy boils down to the following. In this paragraph, we consider flattened representations for collections of items from the nested layer. In the next paragraph, we consider how operations on nested values are translated to operations on the flattened data representation.

At the flat level, we have several primitive types plus one composite type,

the binary relation.[2] At the nested level, an extension defines a new type X by specifying how to represent a function $\alpha \to X$ using these binary relations. In other words, extensions do not specify how to represent a single element of X, they specify how to represent a collection of Xes, each identified by a unique key of type α. Example: a collection of dog names identified by numeric dog identifiers is represented as

$$atom\langle \begin{array}{c|l} \multicolumn{2}{c}{dognames} \\ \hline 1 & \mathit{fido} \\ 7 & \mathit{spot} \\ 9 & \mathit{rex} \end{array} \rangle.$$

Here, *dognames* is a binary relation from the flat level, which is wrapped in a so-called *atom*-frame. The name *atom* is chosen to suggest that *atom*-frames serve as wrappers for primitive values. The frame above stands for, and is often identified with, the mapping $\{1 \mapsto \mathit{fido}, 7 \mapsto \mathit{spot}, 9 \mapsto \mathit{rex}\}$.

Notice how the binary relation *dognames* has been wrapped in a $atom\langle\rangle$ structure. This is called a *frame*. As an example of a more complex frame, consider the type $\mathsf{Bag}X$ of bags (multisets) containing values of type X. We represent a collection of type $\alpha \to \mathsf{Bag}X$ using the *bag*-frame

$$bag\langle d, r, F\rangle$$

where F is a frame of type $\beta \to X$ and r is a binary relation between the element identifiers β and the bag identifiers α. The first component, d, is an explicit representation of α. It is needed for some rewrite steps. With

$$\begin{array}{c|c} \multicolumn{2}{c}{d} \\ \hline 100 & 100 \end{array}, \quad \begin{array}{c|l} \multicolumn{2}{c}{r} \\ \hline 100 & 1 \\ 100 & 7 \\ 100 & 9 \end{array}, \quad F = atom\langle \begin{array}{c|l} \multicolumn{2}{c}{dognames} \\ \hline 1 & \mathit{fido} \\ 7 & \mathit{spot} \\ 9 & \mathit{rex} \end{array} \rangle,$$

the frame $bag\langle d, r, F\rangle$ represents, and is often identified with, the function $\{100 \mapsto \{\!|\mathit{fido}, \mathit{spot}, \mathit{rex}|\!\}\}$.

8 Query flattening. In the previous paragraph, we saw that extensions define a new type X by giving a *frame* representation for a function $\alpha \to X$. Such a frame represents many X-items at the same time. Similarly, the definition of operations on X is given by specifying how to perform them on many items at the same time. The function semantics of the frames are the key

[2] For reasons outlined in paragraph 33, we base our system on a binary relational model. This is not a fundamental property of our design; the binary model simply fits our needs well.

to how this is organized. Every operation is defined by giving a rewrite rule for the functional composition of the operation with a frame. As an example, consider the pair type $X \times Y$. Functions $\alpha \to X \times Y$ are constructed using a *pair*-frame $pair\langle F, G\rangle$ with $F : \alpha \to X$ and $G : \alpha \to Y$. Conceptually, the projection function $exl : X \times Y \to X$ is defined at the level of a single pair (a, b):

$$exl\ (a, b) = a.$$

In terms of frames, however, it is defined by the rule

$$exl \circ pair\langle F, G\rangle = F.$$

Interpreted as a function, the frame $pair\langle F, G\rangle$ takes a key $a \in \alpha$ and maps it to a pair $(F(a), G(a)) \in X \times Y$. Function $exl : X \times Y \to X$ then maps $(F(a), G(a))$ to $F(a)$, thus showing that the rewrite rule is correct:

$$\begin{aligned}(exl \circ pair\langle F, G\rangle)(a) &= exl(pair\langle F, G\rangle(a)) \\ &= exl((F(a), G(a))) \\ &= F(a).\end{aligned}$$

9 Point-free form. In the previous paragraph we introduced the function *exl*. It is most conveniently defined by the equation

$$exl\ (x, y) = x.$$

This is called a *point-wise* definition because the terms of the equation refer to individual elements. The left-hand side refers to *exl* being applied to a single pair. The second definition in paragraph 8 is given as an equation of functions. Here, the left-hand term of the equation has type $\alpha \to X$. Because this definition does not refer to individual elements but only to properties of functions, we call this a *point-free* definition of *exl*. The immediate benefit of a point-free definition is that it is stated in bulk terms. Hence, it is an excellent starting point for translating the query to bulk operations at the lower level.

It turns out that for many operations on many types, a point-free rewrite rule is easy to state. Sometimes the rewrite rule involves no actual processing of data. As with *exl*, it just rearranges the frame structure. In other cases, rewrite rules replace nested operators by flat operators. For instance, the operator

$$unnest : \mathsf{BagBag}X \to \mathsf{Bag}X$$

introduces a relational join between the r-component of the outer *bag*-frame and the r-component of the inner *bag*-frame:

$$unnest \circ bag\langle d_1, r_1, bag\langle d_2, r_2, F\rangle\rangle = bag\langle d_1, r_1 * r_2, F\rangle.$$

10 Exploiting categorical knowledge. One of the reasons we cite Category Theory as an inspiration is that category theory is based entirely around the notion of reasoning about mathematical objects in a point-free fashion. In the transformational programming community, a lot of work has been done on exploiting categorical knowledge during program transformation. We mention in particular the notion of catamorphisms, which are programs (query operators) that compute their result by induction to the shape of their argument. One well-known example is the *sum* function on lists, often defined (in Haskell notation) as

```
sum [] = 0
sum (n:ns) = n + sum ns
```

Various theorems are known which allow to fuse and otherwise manipulate functions which follow this pattern. One difference between our framework and preceding systems is that in our system, data types which seem to have a recursive structure at the nested level, are in fact implemented in a flat way. Therefore, recursive functions such as the above cannot be implemented directly. We look at various ways to deal with this problem. Ideally, we try to use the efficient versions of these functions defined at the frame level, if they exist. Otherwise, we attempt to translate the catamorphism into a fix point equation at the relational level.

11 Query language. The exact details of the query language may differ depending on the nature of the system implementing the point-free query flattening strategy. Typically it will be a functional language. In principle, the incoming expression is completely translated before execution begins at the lower level, and the flattened query we generate can only depend on properties of the query, not on the data in the database. This means that our language does not support recursion. Instead, queries use higher-order functions defined at the frame level. One example of a common higher-order function usually defined by recursion is the *map* operator on lists, bags, and sets. Point-wise, it looks like $map\ f\ \{\!|x, y, z|\!\} = \{\!|f\ x, f\ y, f\ z|\!\}$. At the frame level, such a function can elegantly be defined as follows:

$$map\ f \circ bag\langle d, r, F\rangle = bag\langle d, r, f \circ F\rangle.$$

Another beneficial concept brought by the categorical influence is the *monad*. It allows us to define a comprehension syntax in a simple way:

$$\begin{aligned} Bag[f\ x \mid x \leftarrow e] &:= map\ (\lambda x \bullet f\ x)e, \\ Bag[e \mid x \leftarrow e', y \leftarrow e''] &:= \end{aligned}$$

$$unnest\ Bag[Bag[e \mid y \leftarrow e''] \mid x \leftarrow e'],$$
$$\ldots$$

where $(\lambda x \bullet e)$ is a lambda term mapping x to e. In contrast to most literature and programming languages, a comprehension has to have a name, like *Bag* above and *Sum* below, which indicates the monad. A fragment [|] without the name has no meaning. For details, see paragraph 129.

Extensions can easily add new comprehension types, such as a

$$Sum[x \mid x \leftarrow E] = \sum_{x \in E} x$$

comprehension by defining some appropriate functions, see also paragraph 30. Comprehension syntax is a useful tool for expressing queries. We shall later see how several aspects of query flattening in other systems can be mapped to the flattening of comprehensions in Dodo.

12 Applications in the XML domain. The pattern made explicit in our framework can be observed in systems in the real world. For instance, in Chapter 4 we show how the design of the Pathfinder XQuery engine can be fitted into the framework very neatly. A Pathfinder extension can be defined by introducing flattened type definitions for the fundamental XML data structures. Furthermore, specific algorithms to accelerate XPath axis steps can be implemented as a low-level extension, whereby its application is nicely hidden in the operations defined on the new types. Pathfinder's approach for efficiently handling nested FLWOR expressions maps almost directly to our framework.

1.3 Dodo and the Multi-model DBMS

In this section we take a closer look at the Multi-model DBMS architecture mentioned in paragraph 3, in particular how Dodo implements and contributes to it. The multi-model database architecture is advocated by de Vries [dV99] and others [vKVdV+03, dVLB03] as a way to deal with the tension between *data model expressiveness and extensibility* on the one hand and *efficiency and optimization* on the other.

As discussed in paragraph 1, in order to make database technology more suitable for non-traditional applications such as multi-media and semi-structured data, a more expressive data model than the one usually provided by current DBMS vendors is desirable. At the same time, efficiency is expected from such a DBMS supporting a more expressive data model.

It should be noted that by "more expressive," we mean expressive in the sense of what can *conveniently* be expressed in it. The relational model in itself is of course perfectly able to express any data structure one might conceivably wish to put into a database. We do, however, want to make it more convenient for applications to properly put complex data structures into a database management system and avoid the common situation where they are maintained in a separate system *next to*, rather than *within* a DBMS.

13 Tension. The tension arises when we try to implement more expressive data models efficiently. We observe two issues. First, as mentioned in paragraph 2, current trends in hardware strongly favour simple data structures over complex ones. In order to perform well on modern hardware, data needs to be arranged in a simple, cache-friendly way with predictable access patterns and little inter-datum dependency. Otherwise, computation becomes latency bound and performance suffers.

The other, more fundamental issue is that of *data independence*, a more well-known tension between expressiveness and optimizability. More complex data models are harder to understand for query optimizers than simple data models. In this respect, the relational model is a local optimum, because

- it is simple, and based on a sound mathematical foundation, set theory;
- due to this foundation, it is able to express any structure we require of it;
- it is simple enough for automated query optimizers to get a grip on;
- it is abstract enough to allow several very different implementations of both storage structure and algorithms, thus giving the optimizer something to choose between;
- the theory of normal forms makes it easier to design a data representation which is not tied too close to the first application using the data, making it more likely to be useful for later applications as well.

All in all, the relational model truely is one of great achievements of computer science in the last 50 years. The goal of the multi-model architecture is to allow more expressive data models, but carry over the data independence benefits of the relational model.

14 Data independence. Data independence is the corner stone of classical relational database theory, but curiously overlooked by most newcomers with extended data models. The ANSI/SPARC architecture [TK78] separates the DBMS in three layers, the external, conceptual and internal layer. Three layers means two interfaces, and thus two opportunities for decoupling:

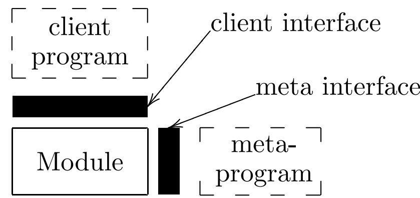


Figure 1.1: Open implementation approach.

- *Physical data independence* means that the storage structure of data can be changed without changing the conceptual structure. The main benefit of this is that performance-inspired changes to the storage structure do not affect the conceptual schema.
- *Logical data independence* means that the conceptual structure can be changed without changing the external schemas presented to existing applications. We stress the importance of the plural "schemas": one of the purposes of the conceptual schema is to define a common world view which can be mapped to several application-specific external schemas specialized for specific applications. This makes it easier to adapt data of one application for use by other (possibly future) applications.

It is interesting to notice how nowadays, physical data independence is taken so much for granted that it is often overlooked, whereas logical data independence remains underused. Here, we focus on the importance of physical data independence for performance, in particular query optimization.

Physical data independence is not a database-only concept. There are more layers hiding physical issues. Below the database we find file systems, RAID and volume managers, and disk-internal block remappings. Each hides storage level complexity from higher layers. These abstractions have grown over time. For instance, one of the reasons that the classical relational model papers of the 70s [Cod70, CAB$^+$81] put so much emphasis on abstracting away from file storage details is that at the time, files were often only thin abstractions over consecutive ranges of disk addresses (think of disk-number/cylinder/head/sector tuples), rather than the convenient hierarchical name spaces we have nowadays. Similarly, one can argue that application programmers nowadays think of relational databases as "physical storage" in the sense that the interface to the DBMS is where one stops thinking about storage details.

15 Data independence and optimization. A downside of layering, or modularity in general, is that not every implementation issue can and should

be hidden behind an abstract interface. There is a tension between software engineering benefits favouring hiding and execution performance requiring insight in and influence on implementation details in lower layers. Referring to Kiczales [KB96], De Vries cites the *Open Implementation* approach as an important guideline for handling this tension followed in the multi-model architecture. In the open implementation approach, there is still abstraction, but instead of treating a layer or module as a black box, it provides a meta-interface (see Figure 1.1) allowing a client of the layer or module to make certain performance choices. A common example is the Unix call `madvise(2)`, which is used to advise the kernel about the expected access patterns for a memory region, allowing the kernel to choose an appropriate paging strategy.

16 Extensibility and optimization. To embrace non-traditional application domains, a DBMS need not only support a more expressive data model, it needs to be extensible with new data types as well. For performance, it is important that an optimizer gets a grip on the new structures. Many approaches to this problem have been tried, with varying success. We mention Garlic [CHS+95] and Predator [Ses98]. Garlic integrates heterogeneous data sources using *Object Wrappers*. Wrappers may provide statistics to Garlics optimizer. Even then, processing queries outside the underlying database may lead to performance loss [dVEK98].

Predator features *E-ADTs*, Enhanced Abstract Data Types. The Enhancement in an E-ADT lies primarily in the optimizer interface the E-ADT provides to the Predator framework. This allows extensions to add knowledge to the optimizer, and to a limited extent, to perform cross-extension optimization. This is a form of meta-interface in terms of the open implementation approach.

The E-ADT approach is limited by the fact that every E-ADT manages its own data within a common storage manager, Shore. For instance, cross-extension pipelining is not possible. Moreover, because every E-ADT is built directly on top of the storage manager, it is hard to re-use functionality implemented within another E-ADT.

17 Multi-model architecture. In summary of the preceding paragraphs, we note that databases with enriched data models may benefit from data independence in ways similar to what purely relational databases typically provide, and from better inter-extension interaction. The Multi-model DBMS architecture [dV99, vKVdV+03] is an attempt to provide this. Recall that the ANSI/SPARC model defined three layers with two interfaces. Commonly, all layers implement the relational model, but this is not required by ANSI/SPARC. The multi-model architecture proposes to use different data models for each layer. The nature of non-traditional applications requires from the conceptual layer to support complex, usually nested types.

Generalizing from the properties of the relational model mentioned in paragraph 13, the interface between layers needs to be chosen in such a way that the language is simple enough for the optimizer to grasp, with clear algebraic properties for the optimizer to exploit. Hence, for the logical and physical layers below it, simpler data models are better. The physical layer should typically use a machine-friendly relational model. De Vries [dV99] and van Keulen [vKVdV+03] suggest to use a variant of XNF^2, Extended Non-First Normal Form, for the logical layer.

The multi-model architecture addresses the concerns mentioned in the preceding paragraphs in the following way. With regard to extensibility and cross-extension interaction, notice that in the multi-model architecture, the system is extensible *at every layer*. Usually, an extension defining new data types and operations at the conceptual layer also brings a couple of support operations at lower layers. At the same time, however, it has access to all existing functionality at the lower layers. In particular, there is no need to reimplement various join algorithms already provided by other extensions. Recall that in the E-ADT approach described above, there was the possibility for an extension to add knowledge to the optimizer, but at the end of the road every extension had to implement its functionality itself, directly on top of the storage manager. In the multi-model approach, an extension at the conceptual layer can borrow logical and physical operations from other extensions, and only implement those physical operators not provided by others.

Data independence is improved because every layer, or more precisely, every interface between layers provides an opportunity for abstraction. Usually many physical implementations are possible for one conceptual type or operation. A meta-interface is provided by allowing an extension to define its own optimization rules, in particular ones that recognize possibilities to make use of related special-purpose operations implemented as extensions in a lower layer. Furthermore, as mentioned earlier, it is important for performance, that it is possible to map application-oriented object-at-a-time operation definitions to efficient bulk operations at the physical level. Therefore, the interface needs to expose its functionality with a suitable *unit of work*. If the application presents the database only with microsteps, as often happens in purely Object Oriented systems, there is nothing for the optimizer to optimize. The theory in this thesis is intended to help programmers design the mapping of conceptual operations to efficient bulk operations at the physical level in a controlled and verifyable way.

18 Multi-model related research. The roots of the multi-model architecture lie in the Magnum-project [BWK98, BQK96]. Magnum was a structurally object-oriented DBMS intended to efficiently integrate spatial and the-

matic data for GIS purposes. Magnum used MonetDB as its physical layer. MonetDB [BK99] is an efficient and extensible main-memory database kernel designed with the cache hierarchy of modern CPUs in mind [Bon02]. In our context, an important feature of MonetDB is its binary relational data model, where every relation (BAT, binary association table) is either unary or binary.

On top of MonetDB, Magnum featured **Moa**, the Magnum Object Algebra. Moa allows one to add new data types by describing their decomposition into BATs, and to use newly added operators both at the nested Moa level and the flat Monet level. Moa made heavy use of the concept of an *IVS*, an Indexed Value Set. Basically, if an extension added a new data type, it would give two structural definitions. One, the *Value* definition, expressed how a single value of the new type was decomposed into flattened values, and the other, the *IVS* definition, expressed how a *collection* of values was flattened. By choosing a suitable IVS representation for data types, it was possible to construct bulk versions of operators which were considerably more efficient than repeated application of the Value version of the operator. This allowed Magnum to perform well on, for example, the Sequoia benchmark [Sto93].

In follow-up projects, Moa was adapted to other applications. The Mirror project [dVvDBA99, dV99] addressed content-based multimedia retrieval for text, images and music. In the AMIS project [BdVBA01, BHC$^+$01], the focus was on mapping top-N information retrieval queries to parallel evaluation over fragmented data, demonstrating the performance benefits of data independence. Finally, in the SUMMER project, Moa was modified to function as a common middleware federating Monet- and SQL-based databases and an XML/XQuery-based conceptual data model with multimedia retrieval extensions.

Dodo's *frames* described in paragraph 7 were inspired by Moa's notion of IVS. Cast in the category theoretical language of objects and arrows, we express an IVS as an arrow from keys to values. In this way, value operations become function applications and IVS operations can be written as function compositions. Because the value representation can be regarded as a special case of the IVS representation where the domain consists of a single key, Dodo needs only to deal with data in IVS form. Writing queries in point-free form, where the query is fully written as the composition of functions, provides a natural way to move the query to bulk orientation.

Chapter 2

Function-based data model

In the previous chapter we sketched the trade-off between expressiveness of the data model and the performance of query evaluation over that data model. The problem is that implementing complex data structures in a straightforward fashion ties the storage structure too tightly to the logical structure of the data. This makes it hard for the query engine to derive efficient execution plans, since efficient execution requires bulk processing. In this chapter we describe our approach and make it concrete using Dodo, a prototype system.

The purpose of the current chapter is to explain how the simple mathematical concept of *point-free reasoning* can be used to guide the extension programmer in designing a flattened representation of complex data structures and especially operations. Our slogan is

Replace function application by function composition!

2.1 Dodo

We describe our approach to query flattening in the context of our prototype, Dodo [RFK04]. Dodo is a translation service that sits between the application and the underlying (relational) database. It provides a simple language capable of expressing the queries both as seen at the logical layer and as seen at the storage layer. At the logical layer, the query is stated in terms in terms of complex, nested data types, and often in an item-at-a-time fashion. The query then is gradually transformed into a form where processing takes place using bulk operations from the storage layer.

19 One language. Why do we stress that Dodo provides *one* language? Why not two, one for the complex data and one for the flattened data? It is, after all, fairly common to define two languages and a "semantic bracket $[\![\cdots]\!]$" to separate them during translation. If, for instance, the first language is arithmetic expressions written in infix style $(x+y)$, and the second language arithmetic in prefix style $(+\ x\ y)$, then rewrite rules can be defined similar to

$$[\![x+y]\!] = +\ [\![x]\!]\ [\![y]\!].$$

This is a natural and tidy approach, but with Dodo, the situation is more complicated. In our setting, the data is stored in flattened form, but the query result is to be presented in nested form. Therefore, regardless of the query plan, at some point a complex value has to be assembled out of flattened parts. Query flattening in Dodo merely changes where this happens. If a query is stated using complex operations, the query plan first assembles flattened data into a complex value, and then the operation can be performed. After translation, the query plan first performs flattened operations and then assembles the result into a complex value. The underlying assumption is that for computation-intensive queries, the latter approach is more efficient.

Dodo combines all these elements in one language. If we were to split it into a complex language and a flattened language, the question is: where do we put the reassembly sublanguage? The whole point of splitting into two languages is that we can start with an expression completely in one language and finish with an expression completely in the other, and the reassembly step breaks this symmetry, wherever we put it.

20 Roles. It is helpful to distinguish the following, possibly overlapping roles. *Users* formulate queries. They think in terms of complex data structures and formulate queries over them. *Extension writers* define extensions in terms of a mapping from these complex data structures to flattened storage. They understand the semantics of both the complex and flattened layer very well, because they also need to provide a mapping from complex operations to corresponding flattened operations. *Database administrators* use the mapping established by the extension writers to construct a user-friendly nested view on the existing relational data at hand in a specific case. Essentially, the extension writer looks at the mapping of new data types, whereas the database administrator looks at the mapping of concrete data.

21 Intended use. Dodo is intended to be used for analytical queries over complex data structures, that is, queries which touch a lot of data and are often computation intensive.

It is possible to use Dodo without extending the system. In that case, the data is represented using the tuples, sets, lists and bags provided by the core system. The database administrator stores data in relational form and creates a "data dictionary" which assembles the relations into *frames* which present the user with bags of lists of tuples, or whatever is convenient for the application at hand. Notice that it is possible to provide several Dodo views of the same relational data.

Providing this nested view may be useful in itself, but is not the primary motivation behind the Dodo approach. The point of Dodo is to allow a programmer, the extension writer, to extend Dodo with new data types which are internally implemented using bulk operations. For example, the extension writer may define an application-specific data structure such as a document representation for Information Retrieval [dV99], or geometric structures for geographical applications. For instance, the extension writer may define a $docrep\langle\rangle$ frame (structure mapping) which specifies how to map a collection of documents onto a couple of relations in the underlying database. Using the framework presented in this chapter as a guide, the extension writer maps interesting operations on document representations to relational operations. Most of these are classical relational operators such as selections and joins, but interesting applications will also involve custom operations written in C and linked into the database kernel.

While the extension writer defines the mapping at the type level, the database administrator role (DBA) is concerned with actual data. Given $docrep\langle\rangle$ frames as defined above, the DBA may decide that this specific application calls for two document collections, and thus two instances of the $docrep\langle\rangle$ structure. Frames are built on relations in the underlying database, so the DBA also has to decide to which relations the instances map, how data is to be loaded into them, etcetera.

2.2 Nested data model and sublanguage

In this section we describe the upper part of diagram (1.1).

$$\begin{array}{ccc} N & \xrightarrow{q} & N \\ \uparrow & & \uparrow \\ F(N) & \xrightarrow{F(q)} & F(N) \end{array} \qquad (1.1)$$

We give a description of the kind of complex data model (N) we use in Dodo and the query language it presents to the user. The parts of the query language

related to the flattened layer are deferred to later sections. The type system for nested data developed in this chapter is based on a categorical notion of data types, similar to [Gru99] and [Fok92]. In general, the emphasis is more on the structure of functions between data types than the data types themselves.

2.2.1 Types

The type system is centered around the notion of a *type former*. Type formers construct a type (a set, basically). Some type formers take other types as parameters (List A), while others do not (Strings, or numbers).

22 Simple types. Simple types are String, Integer and other things that can be stored directly in the data model of the lower layer. These types are atomic in the sense that they have no visible internal type structure, as far as the type system in N is concerned. Generally, simple types map directly to scalar types provided by the underlying database.

- Unit type. The unit type $1 = \{\dagger\}$ contains precisely one value, written $\dagger$. This data type can be stored very efficiently because it requires zero bytes of storage.
- Primitive types. Primitive types are defined by the underlying system. In this report we assume *Int*, *Str* and *Bool*, sometimes written $\mathbb{Z}$, SS and $\mathbb{B}$. The **if then else** construct requires the boolean type. We often use the letters K, L and M to refer to primitive types, including the unit type.
- Key types. Key types are opaque values used to identify other things. They play an important role in the flattened layer and at most a minor role in the nested layer. Essentially, when a collection of Dodo values is stored as a table in the flattened layer, the primary key of that table is represented using a key type. Key types are not precisely atomic, sometimes several keys are combined into *product keys* or *sum keys*. We will get back to that in section 2.3. We will often use the greek letters α, β, γ, ... to refer to key types.

23 Sum and product types. Sum and product types are two common ways of combining Dodo types. Product types are cartesian products; the product type $A \times B$ denotes pairs (or more generally, tuples). Sum types are disjoint unions. The sum type $A + B$ denotes values that are either of type A *or* of type B. So, informally, elements of $A \times B$ contain an A *and* a B, whereas elements of the disjoint union $A + B$ contain *either* an A *or* a B. Notice that for finite sets A and B containing n and m elements, respectively, the product

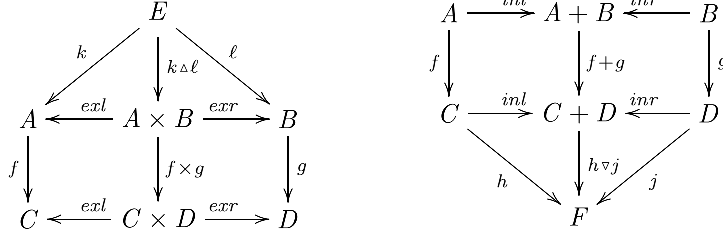


Figure 2.1: A summary of functions related to sum- and product types, given as a graph with types as nodes and maps as edges. A path through the graph is a concatenation of edges and corresponds to a composition of maps. This is a *commutative diagram,* which means that the existence of multiple paths 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 $exl \circ (f \times g) = f \circ exl$.

type $A \times B$ has $n \times m$ elements, whereas the sum type $A+B$ has $n+m$ elements. More formally,

- 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 right-hand parts. The notation $\times$ is extended from types to functions: for $f : A \to C$ and $g : B \to D$ we define $f \times g : A \times B \to C \times D$ by

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

  Products can be constructed using the $\vartriangle$ construct. For every $k : E \to A$ and $\ell : E \to B$ we define $k \vartriangle \ell : E \to A \times B$ by

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

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

- 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 dis j."

24 Function types. Function types $X \to Y$ denote mappings from one type to another. A function of type $X \to Y$ assigns to *every* element of X an element of Y. We often write type letters A and B for types which are known not to be function types. For instance, the type $\mathbb{Z} \to \mathbb{B}$ is an instance of the general type pattern $A \to B$, whereas $\mathbb{Z} \to (\mathbb{Z} \to \mathbb{B})$ is not.

Functions can be composed. Function composition means two or more functions are applied after each other. The function obtained by applying f after g is written $f \circ g$, e.g.,

$$(f \circ g)\ x = f(g\ x).$$

25 Functors. Functors build types out of other types. In fact, $\times$ and $+$ are functor instances, but were treated specially for clarity.

- 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.

- Bifunctors. It is possible to have functors that take more than one type as an argument. Such functors are called *bifunctors*. 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.

2.2.2 Query language

In this section we describe the Dodo query language, or, to be more precise, the nested part of the query language. It serves as an example of a query language dealing with complex types. Usually, the query language of a given system can be mapped onto something similar to the language described in this chapter. Minor differences do not matter much; we shall see in section 2.5 that what matters is that queries can eventually be rewritten into a composition of primitive operations.

26 Example. From the users point of view (as opposed to the extension writers view), Dodo implements a typed λ-calculus, extended with comprehension syntax. For instance, given a bag of numbers b (predefined by the database administrator), the following query computes the bag of squares of those numbers:

$$\mathsf{Bag}\ (\lambda n : \mathbb{Z} \bullet n^2)\ b.$$

Here, Bag is the bag-functor. Recall from paragraph 25 that functors are used both to form new types and to lift existing functions to those types. In this example, b has type $\mathsf{Bag}\mathbb{Z}$ and $\mathsf{Bag}\ (\lambda n : \mathbb{Z} \bullet n^2)$ has type $\mathsf{Bag}\mathbb{Z} \rightarrow \mathsf{Bag}\mathbb{Z}$ because the λ-term $(\lambda n : \mathbb{Z} \bullet n^2)$ has type $\mathbb{Z} \rightarrow \mathbb{Z}$. The λ-term denotes the function which, given an argument n, computes its square n^2. An alternative notation for the above query would be

$$Bag[n^2 \mid n \leftarrow b].$$

This is a *Bag*-comprehension. The first part, *Bag*, signifies the kind of value we are about to construct. The n^2 calculates the values to be put into the construction process, and after the $|$, there are zero or more generators which generate bindings for n. Comprehensions can be used to construct collections such as bags, but also to invoke aggregate functions. For instance, we could write $Sum[n^2 \mid n \leftarrow b]$ to calculate the sum of the squares of the numbers in the bag. Extension writers can easily define new kinds of comprehensions; it is simply a matter of defining a few functions with certain required properties.

27 Expressive power. Dodo implements a typed λ-calculus without a Y-operator or similar. This means that it cannot directly express recursive functions. The reason for the lack of recursion is that Dodo itself does not evaluate the queries, but it translates them to a relation query plan, and relational algebras typically do not support recursion. See, however, Chapter 6 for a sketch of how to deal with a specific type of recursion in Dodo. The query translation mechanism described in the current chapter can only deal with first-order λ-terms. By disallowing recursion, all higher-order λ-terms can be expanded as a preprocessing stage.

In a database context, the lack of recursion is not as much of a problem as it would seem on first sight. Most uses of recursion involve traversing a data structure and doing some processing along the way, and extension writers can endow their new data types with suitable higher-order functions to do so.

28 Language. In the following paragraphs we give a grammar and informal semantics for the Dodo query language. The formal semantics is expressed as rewrite rules in Section 2.5. The current section defines the grammar of the high-level, lambda like part dealing with complex data structures. In section 2.3 we define the part of the language dealing with flattened data. In sections 2.4 we introduce frames, the bridge between the two layers. The syntactical constructs are grouped according to the type of value they denote. Later, this will make it easier to verify the completeness of the rewrite rules.

Conceptually, everything in the following 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, but it makes the mixed trees that occur during the rewrite process very hard to describe. Therefore, we just call everything an *expression* and use the type system to impose more structure on it.

29 Simple 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	
let $x = e$ **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 $e' : B$
if b **then** e_1 **else** e_2	:	A	conditional	if $b : \mathbb{B}$ and $e_1 : A$ and $e_2 : A$

30 Monad comprehensions. Monad comprehensions allow the user to write expressions such as

$$List[(x, y) \mid x \leftarrow xs, y \leftarrow ys].$$

For the definition of a monad, see paragraph 130. For the moment, it is sufficient to know that the system associates the comprehension type M to a functor T and a so-called algebra τ_M. The functor encodes the type corresponding to the comprehension, for instance, the *Bag* comprehension is associated to the functor Bag because given values of type A, it constructs values of type $\mathsf{Bag}\ A$. Likewise, the Sum comprehension is associated to the identity functor Id because given numbers of type N, it calculates a result also of type $N = \mathsf{Id}\ N$.

$M[e \mid\,]$	:	$\mathsf{T}A$	monad-comprehension	if $e : A$
$M[e \mid x \leftarrow e']$	:	$\mathsf{T}B$	monad-comprehension	if $e_{x:A} : B$ and $e' : \mathsf{T}'A$ and $(\!\|\tau \to \tau_M\|\!)$ well-defined with τ init. algebra for T'
$M[e \mid b]$	:	$\mathsf{T}A$	monad-comprehension	if $e : A$ and $b : \mathbb{B}$
$M[e \mid q, qs]$	:	$\mathsf{T}A$	monad-comprehension	if $M[M[e \mid qs] \mid q] : \mathsf{TT}A$.

See paragraph 127 for an explanation of the sentence "$(\!\|\tau \to \tau_M\|\!)$ well-defined with τ init. algebra for T'". Informally, the idea is that it must be possible to convert values from type T' to T using *only* information from T'. For instance, implicit conversion from lists to bags is OK, because it simply drops information about the ordering of elements. Conversion from bags to lists is not OK, because the conversion needs to make up the order in which the elements are put in the list. Because there are multiple choices there, such a conversion must be done explicitly, perhaps using a function *sort*.

Extension writers define the semantics of a monad comprehension $M[\ldots \mid \ldots]$ by associating M with functions $zero_M$, denoting an empty collection or aggregation, $unit_M$ denoting a singleton, and $unnest_M$, which removes a degree of nesting. For more details, see the actual translation rules in figure 2.3, and Chapter 5.

31 Potentially complex function builders. By "potentially complex" we mean that the following constructs may have function types other than $A \to B$. The constructs themselves are easy to understand.

x	:	X	identifier	defined in schema or environment
$f\ e$	:	Y	function application	if $f : X \to Y$ and $e : X$
$f \circ g$	:	$X_1 \to X_3$	function composition	if $f : X_2 \to X_3$ and $g : X_1 \to X_2$

32 Simple function builders. We call the following construct simple because they always have first-order function type $A \to B$.

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

Lambda terms are well-known. The banana brackets $(\!\vert \cdot \vert\!)$ were already referred to in paragraph 30; their precise meaning is explained in paragraph 122. Constant functions are a useful tool in many circumstances. They are syntactic sugar for a lambda term: $\underline{v} = \mathit{const}\ v = (\lambda x \bullet v)$.

2.3 Flat data model and sublanguage

In this section we briefly describe the lower part of diagram (1.1).

$$\begin{array}{ccc} N & \xrightarrow{q} & N \\ \uparrow & & \uparrow \\ F(N) & \xrightarrow{F(q)} & F(N) \end{array} \qquad (1.1)$$

In the flat data model, we have the same primitive native types as in the nested layer, such as integers and strings. One might say that it is the other way around: the primitive types of the nested layer are those types which are in fact already defined in the flat layer. There is only one composite type, the relation. There is a set of operators to work on scalar values and relations. This "column algebra" includes all regular relational operations, such as joins, as well as some specialized operations for internal use by Dodo. The column algebra is a very simple language. There are no variable bindings, explicit loops or other language constructs. As far as Dodo is concerned, there are just column values and operators that combine them into other column values. In section 2.5 we describe how operations from the nested layer are translated to column expressions.

33 Binary model. We choose binary relations for, among others, the following reasons:

- Every n-ary relation can be decomposed into n binary relations. The binary relational model is in a sense the simplest, most Spartan version of the relational model.

- Existing research has shown [BK99] that the binary relational model performs well on analytical query workloads.
- Our approach revolves around functions, and functions are a special case of binary relations. In the flat layer, the binary relations are always between scalar types. The *frame* construct described in section 2.4 generalizes this to nested types.

Because of the binary model, we need no column labels and other complications. Every relation has a head-column and a tail-column. For every column operator it is implicitly defined on which side of the column it operates. For instance, the *plus* operator on relations always adds together the tails of its two arguments.

Every scalar operation (e.g., *plus*) is expected to be provided in two variations. One scalar variant, working on a single primitive value, and one vectorized variant which works on all values in a given column. For more detailed information about the column algebra, see our technical report [RFK04].

34 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; head distinctness is known as "functional"; tail distinctness is known as "injective";
- that they are *complete*, i.e., 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 $[K - L]$. Partial functions, i.e., columns where all heads are distinct, are written $[K \rightarrow L]$. Likewise, columns with distinct tails are written $[K \leftarrow L]$, and head-complete and tail-complete columns are written $[K \vdash L]$ and $[K \dashv L]$, respectively. The arrow ornaments can be combined, allowing us to write $[K \mapsto L]$ for total functions from K to L. Finally, the fact that all tails are known to be equal to the corresponding heads can be indicated by replacing the tail type with an exclamation mark: the column $idunit = \{(\dagger, \dagger)\}$ has type $[\mathbf{1} \mapsto\!\!\!\dashv !]$.

Important It is important to keep in mind that the arrow in a type $\alpha \rightarrow A$ represents a *total* function, whereas the arrow $\rightarrow$ in a column type $[\alpha \rightarrow \beta]$ represents a *partial* functions. We have chosen to use a systematic notation for column properties, but use the conventional function notation elsewhere.

35 Column expressions. Column expressions denote primitive values but most often columns. They occur only within frames expressions, see paragraph 38.

k	:	K	literal	
x	:	$[\alpha - \beta]$	column name	
$op(c_1, \ldots, c_n)$	:	X	prefix operator	if $\mathcal{C}_{op}(X, t_1, \ldots, t_n)$ holds and $c_1 : t_1, \ldots, c_n : t_n$
$c * c'$	:	$[\alpha - \gamma]$	semijoin	if $c : [\alpha - \beta]$ and $c' : [\beta - \gamma]$ note property propagation rules below
$c \cup c'$	:	$[\alpha - \beta]$	column union	if $c, c' : [\alpha - \beta]$

Literals are used in column expressions such as $settail(some_col, 3)$. 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 importantly head-distinct. The union operator $\sqcup$ preserves head-uniqueness if it exists, which is essential when combining, e.g. *bag* domains.

Prefix operations *op* have a type predicate $\mathcal{C}_{op}$ that gives the relation between permitted argument types and return type, including the properties described in paragraph 34. This is used to implement property propagation for column operators. For instance, the predicate $\mathcal{C}_{twin}$ for the operator $twin(r) = \{\!\!\{(x, x) \mid (x, y) \in r\}\!\!\}$ states that if r is head-complete, $twin(r)$ is both head-complete and tail-complete, because *twin* duplicates every head into the tail.

2.4 Flattening data

Frame expressions are the bridge between the nested and the flat layer. They express how to construct a collection of nested values out of scalars and relations from the flat layer. Generally, there is a frame type for every data type. The frame is interpreted as a collection of items of the given type, identified by distinct keys. The details of the interpretation depend on the frame type and are specified by the extension writer. In the examples in paragraph 7 we saw that a collection of primitive values is represented as an $atom\langle\rangle$ frame containing a single binary relation between keys and values. To represent bags and lists, we begin with a frame representation for the elements of the bags, and encapsulate it in a $bag\langle\rangle$ frame which uses a binary relation between bag-keys and element-keys to group the elements together in bags. Details of the representation of some specific data types can be found in chapter 3.

36 Interpretation function. The extension writer defines a new data type by giving a frame representation for it. Syntactically, this is simply a type signature for the components in the frame. Recall the $bag\langle\rangle$ frame mentioned already in paragraph 7. It has type signature

$$bag\langle[\alpha \mapsto\!| !], [\alpha - \beta], \beta \to A\rangle : \alpha \to \mathsf{Bag}\ A.$$

Here, α is the type of bag-identifiers, β is the type of element-identifiers, and A is the type of the elements themselves.

However, the most important job of the extension writer is to specify how the new frame should be interpreted: given a frame $foo\langle a, b, c\rangle$ of type $\alpha \to T$, how does one look up the value corresponding to a given key? We refer to this as the *interpretation function* for the frame. Interestingly, this interpretation function is not usually implemented within Dodo itself: it simply expresses the common understanding of the implementation of a given type between Dodo and a client application, see paragraph 37.

The interpretation function of a frame is important for two reasons. First, it is needed to prove the validity of rewrite rules within Dodo. Operations on nested types are implemented in Dodo as rewrite rules on frames. To check the validity of such a rule, the extension writer needs to show that the interpretation of the rewritten frame is indeed the same as what one would obtain by applying the operation to the interpretation of the original frame. In other words, the obligation of the extension writer is to prove the commutativity of the following diagram:

$$\begin{array}{ccc} N & \xrightarrow[\mathit{operation}]{} & N \\ {\scriptstyle \mathit{interpretation}}\uparrow & & \uparrow{\scriptstyle \mathit{interpretation}} \\ \mathit{frame} & \xrightarrow{\mathit{rewrite}} & \mathit{frame}' \end{array}$$

Second, the interpretation function is needed to interpret the end result of the query. A query begins as a sequence of operations applied to one or more frames containing the data. During rewriting, it is transformed into a frame expression containing column operations. The column operations are handed to the underlying relational database system and the result sets are pasted back into the frame structure. The end result is a frame which gives a relation between a singleton key and the nested value which is the answer to the query. This frame then has to be *interpreted* in order to obtain the actual answer.

37 Dodo Result language. In our work on Dodo, we focus on what happens 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:

$$\{\!|(\text{fido}, 3), (\text{spot}, 1), (\text{rex}, 9)|\!\}$$

rather than

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

The interpretation function is a function of type $Frame \to Key \to X$. It takes a frame and a key for which to retrieve the corresponding value. The form of the result is given here as X and the question is: what is X? There are many possibilities. The interpretation might be a string. It might be an XML document. It might be a Java object. 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 application in a suitable way. What matters however, is that whatever result language has been chosen, it has been specified with sufficient precision for defining unambiguous interpretation functions.

38 Frame expressions. Here we continue with the grammar of the Dodo language started in section 2.2.2.

$f\langle e_1, \ldots, e_n\rangle$	:	$\alpha \to A$	frame	if $e_1 : t_1, \ldots, e_n : t_n$,
				each t_i either $\alpha_i \to A_i$ or X_i,
				and $\mathcal{F}_f(\alpha \to A, t_1, \ldots, t_n)$
$empty$	:	$\emptyset \to A$	empty frame	constructs empty frame $f\langle\rangle$
				with f depending on type A.
$F \sqcup G$	:	$(\alpha_1 \cup \alpha_2) \to A$	frame union	if $F : \alpha_1 \to A$, $G : \alpha_2 \to A$
				and $\alpha_1 \cap \alpha_2 = \emptyset$.
$c * F$	:	$\alpha \to A$	frame translation	if $c : [\alpha \mapsto \beta]$ and $F : \beta \to A$
$F \circ atom\langle c\rangle$	:	$\alpha \to A$	frame translation	as above
dom F	:	$[\alpha \mapsto\!\!\!\rightarrow !]$	frame domain	if $F : \alpha \to A$

The first line of the above table gives the syntax for frame expressions. Every frame type f has a type predicate $\mathcal{F}_f$ which 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, for example, the *bag* frame from paragraph 36. The predicate $\mathcal{F}_{bag}$ for the *bag* frame reads

$$\mathcal{F}_{bag}(t, t_1, t_2, t_3) \iff t = \alpha \to A \ \wedge\ t_1 = [\alpha \mapsto\!\!\!\rightarrow !] \ \wedge\ t_2 = [\alpha - \beta] \ \wedge\ t_3 = \beta \to A,$$

which we abbreviate to

$$bag\langle[\alpha \mapsto\!\!\!\rightarrow !],\ [\alpha - \beta],\ \beta \to A\rangle : \alpha \to \mathsf{Bag}\,A.$$

The other items in the table are "frame valued expressions." Extension writers defining a new frame type are required to give rewrite rules which turn such expressions back into a frame. Semantically, a frame denotes a function whose domain is a finite set of primitive values, usually keys. The frame valued expressions, which in the sequel we call "required column operations," correspond to certain canonical operations on functions when regarded as sets of ordered pairs. For instance, the *empty* operator creates a new function $\{\} = \emptyset$, and the $\sqcup$ operator calculates the union of two frames $F : \alpha_1 \to X$ and $G : \alpha_2 \to X$. For this to be type safe, α_1 and α_2 have to be disjoint and they have to be subsets of the same primitive type α.

- The empty frame operator *empty* a new frame $f\langle \ldots \rangle : \alpha \to X$, where α and X are determined using type inference.
- The union operator $F_1 \sqcup F_2$ constructs a new frame F such that $F\ k = F_1\ k$ if $k \in \mathrm{dom}\, F_1$ and $F\ k = F_2\ k$ if $k \in \mathrm{dom}\, F_2$.
- The frame translation operator $c * F$ constructs the functional composition of a frame and a binary relation. For the result to denote a function again, c has to be functional, too. In formula, $(k, v) \in (c * F)$ if and only if there exists k' such that $(k, k') \in c$ and $(k', v) \in F$.
- In Dodo, the frame representation for functions yielding primitive values is usually $atom\langle c\rangle : K \to L$, with $c : [K \mapsto L]$. In that case, $c * F$ can also be written $F \circ atom\langle c\rangle$.
- The domain operator $\mathrm{dom}\ F$ is used to obtain the domain α of a frame $F : \alpha \to A$. The domain is delivered in the form of a binary identity relation containing precisely the keys in the domain.

Many examples can be found in Chapter 3.

2.5 Flattening queries

In the previous section we saw how frames are used as a flattened representation for nested data structures. Here we show how operations on nested data structures are translated to operations on flattened data. In particular, one frame represents a multitude of items at the same time, and our goal is to preserve the

d	
†	†

r	
†	1
†	7
†	8

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

g	
1	2005
7	1998
8	2001

$$(\lambda w \bullet dogs) = dogs' = bag\langle d, r, tuple\langle atom\langle f\rangle, atom\langle g\rangle\rangle\rangle$$

Figure 2.2: Example data. Recall that the key † is the single element of the unit type $1 = \{\dagger\}$.

bulk nature of frames during the translation of nested operations. The translation takes place in two phases. In the first phase, the query is brought in *point-free form.* In the second phase, the point-free representation is combined with the frames containing the data, and the combination is gradually rewritten into a frame representation of the query result. In section 2.5.1 we discuss the point-free query form. In section 2.5.2 we give the rewrite rules which bring Dodo queries in this point-free form. Most rules are trivial. The only non-trivial rule, which deals with the elimination of nested scopes, is discussed in section 2.5.3.

39 Example. To show how the functional data representation benefits query rewriting, we systematically rewrite a simple example query. Consider the following query, which computes the birth years of all dogs.

$$q \;=\; Bag[y \mid (x, y) \leftarrow dogs] : \mathsf{Bag}\, Dog.$$

In this query, the identifier *dogs* represents the collection of dogs as thought of by the user, i.e., $dogs : \mathsf{Bag}(Dog \times \mathbb{Z})$. Under water, however, Dodo works with functions, so the database administrator has defined a frame $dogs'$ using the equation

$$(\lambda w \bullet dogs) = dogs' = bag\langle d, r, tuple\langle atom\langle f\rangle, atom\langle g\rangle\rangle\rangle \tag{2.3}$$

with d, r, f and g as in figure 2.2. Hence, *dogs* as seen by the user is the value of the constant function $dogs' : 1 \to \mathsf{Bag}(Dog \times \mathbb{Z})$ defined by the database administrator.

Similarly, the query q is replaced by the constant function $q' = (\lambda w \bullet q)$. This is necessary because the end result is going to be the frame representation of the query result. A frame representation is of function type. Therefore, if we want to arrive at the query result using a series of rewrite steps, the query we start out with also needs to have function type.

As an illustration of the Dodo rewrite process, here are the steps used to flatten q'.

$$
\begin{array}{rl}
& \lambda w \bullet Bag[year \mid (name, year) \leftarrow dogs] \\
(E1) \;= & \quad \{ \text{ syntactic sugar, } exr : X \times Y \rightarrow Y \text{ is projection function } \} \\
& \lambda w \bullet Bag[exr\ z \mid z \leftarrow dogs] \\
(E2) \;= & \quad \{ \text{ definition Bag monad, paragraph 70 } \} \\
& \lambda w \bullet \mathsf{Bag}\ (\lambda z \bullet exr\ z)\ dogs \\
(E3) \;= & \quad \{ \text{ law: } (\lambda x \bullet f\ x) = f \text{, if } x \text{ is not free in } f \ \} \\
& \lambda w \bullet \mathsf{Bag}\ exr\ dogs \\
(E4) \;= & \quad \{ \text{ law: } (\lambda x \bullet f\ a) = f \circ (\lambda x \bullet a) \text{, if } x \text{ is not free in } f \ \} \\
& \mathsf{Bag}\ exr \circ (\lambda w \bullet dogs) \\
(E5) \;= & \quad \{ \text{ Equation (2.3) } \} \\
& \mathsf{Bag}\ exr \circ dogs' \\
(E6) \;= & \quad \{ \text{ Again, Equation (2.3) } \} \\
& \mathsf{Bag}\ exr \circ bag\langle d, r, pair\langle atom\langle f\rangle, atom\langle g\rangle\rangle\rangle \\
(E7) \;= & \quad \{ \text{ definition of } \mathsf{Bag} \text{ functor, paragraph 70 } \} \\
& bag\langle d, r, exr \circ pair\langle atom\langle f\rangle, atom\langle g\rangle\rangle\rangle \\
(E8) \;= & \quad \{ \text{ definition of } exr \ \} \\
& bag\langle d, r, atom\langle g\rangle\rangle \\
(E9) \;= & \quad \{ \text{ interpretation of } bag\langle\rangle \text{ frame, paragraph 70 } \} \\
& \dagger \mapsto \{\!|2005, 1998, 2001|\!\}
\end{array}
$$

In the above calculation, there are roughly two phases. The first five steps are primarily concerned with breaking down elements of the query language into simpler parts. This includes multiple binding, comprehension syntax and variable names in general. In the second phase, constant functions such as $dogs'$ are replaced by their frame definition. Functions such as Bag and exr are defined using a rewrite rule. Sometimes, the rewrite rule simply rearranges the query: exr drops part of the $pair\langle\rangle$ frame, and map inserts its argument in front of the inner frame of the $bag\langle\rangle$. Often, rewrite rules also introduce column operators. For instance, the translation of the operator $unnest : \mathsf{BagBag}\, X \rightarrow \mathsf{Bag}\, X$ involves a relational semijoin $r_1 * r_2$ between the relations r_1 and r_2, where r_1 is the relation between outer bags and inner bags, and r_2 the relation between inner bags and elements X.

For the slightly less trivial query $Sum[y \mid (x, y) \leftarrow dogs]$, the translation would be very similar to the above, but finish with a step where Dodo uses the built-in `sum()`-operator of the underlying database to calculate the sum of the numbers in g. The link between the comprehension type Sum and the column

operator `sum()` is given explicitly by the extension writer, see paragraph 73. Notice that using `sum()` avoids having to traverse the *bag*/*tuple*/*atom*-tree, which would have been necessary if the data were stored in a nested way.

40 Roles (again). It is instructive to consider the previous example with the roles defined in paragraph 20 in mind. Query q is posed by the user. Steps $(E1)$ up to $(E4)$ are either well-known identities from lambda calculus or part of the definition of the language. One might say that they are defined by the "language designer" or the "compiler writer." However, they make use of information provided by the extension writer. For instance, the *Bag* comprehension type has been declared by an extension writer to correspond to the Bag functor. The definition of *dogs* used in steps $(E5)$ and $(E6)$ is provided by the database administrator. The rewrite rule for the Bag functor and *exr* have again been defined by the extension writer.

2.5.1 Point-free form

41 Point-free form. The first several steps of the calculation in paragraph 39 bring the query in *point-free form.* By point-free, as opposed to *point-wise*, we mean that the expression is completely written as a composition of (higher-order) functions, without reference to individual elements or variable bindings. For instance, of the equivalent equations

$$\begin{aligned} h(x) &= f(g(x)), \\ h &= f \circ g, \end{aligned}$$

the first is point-wise, whereas the second is point-free. The expression in Par. 39 reaches point-free form after step $(E5)$. In point-free form, all variable bindings such as $name$, $year$, z and w have been eliminated, and function application is only used for higher-order functions such as map.

The practical significance of the point-free form is that it is a first step towards bulk operation. If we want to apply a function $f : A \to B$ to every item in a frame $F : \alpha \to A$, then the natural way to write this is $f \circ F : \alpha \to B$. Consider the implementation of a data type. For simplicity, we focus on pairs $X \times Y$. In a system designed in a point-wise style, the pair type is typically defined by stating that a pair is formed by somehow aggregating the representation of the elements of the pair. Notationally, this is often done using parentheses and comma, e.g.,

$$(a, b).$$

In terms of this representation, the projection function $exr : X \times Y \to Y$ is defined by the equation

$$exr\ (a, b) = b. \tag{2.4}$$

The *point-free* definition of pairs is stated as an equation at the function level rather than the element level. Notice that every function that produces pairs can be split into a part f that produces the first component, and a part g that produces the second. Recall from figure 2.1 the point-free definition of pairs as a data type whose values are constructed using the combinator $\vartriangle$ and deconstructed using two functions *exl* and *exr* satisfying

$$\begin{aligned} exl \circ (f \vartriangle g) &= f, \\ exr \circ (f \vartriangle g) &= g. \end{aligned}$$

This definition precisely captures the notion of a pair. This notion is shadowed in the frame-definitions of the above functions, which read

$$\begin{aligned} F \vartriangle G &= pair\langle F, G\rangle \\ exl \circ pair\langle F, G\rangle &= F \\ exr \circ pair\langle F, G\rangle &= G \end{aligned} \tag{2.5}$$

42 Benefit of the point-free representation. A minor benefit of the point-free definition is that it is more abstract. It states the interface a pair type must provide without prescribing a particular implementation. A more important benefit is that it suggests a possible bulk implementation. In the example, pairs are implemented using a $pair\langle\rangle$ frame, where $pair\langle F, G\rangle$ has type $\alpha \to X \times Y$ if $F : \alpha \to X$ and $g : \alpha \to Y$. This is completely analogous to the function combinator $\vartriangle$. Furthermore, rewrite step ($E8$) essentially applies the definition of *exr* given in equation 2.5. Notice again that this point-free implementation of projection takes constant time, as it just rearranges the frame structure without touching any data in the underlying database.

Point-free definitions describe what the composition of a function with another function is, rather than what the result is when the function is applied to a single argument. Therefore, if the newly defined function f is composed with another function g, where $\mathrm{ran}(g)$ consists of a million elements, then the point-free definition of f states how to apply f to a million elements at once.

Apart from the above mentioned two benefits, there is another aspect of point-freeness that might be beneficial as well. In a point-free expression of an algorithm the semantic constituents appear syntactically isolated as separate sub-expressions (much more than is generally the case in a point-wise expression). This is beneficial for algebraic manipulations: decomposing an expression in its semantic constituents and then recombining the parts in a different (but

equivalent) way. Transformational Programming exploits this aspect (as shown by Bird in his seminal work [Bir87] and his very advanced elaboration [BM96]); and we hope later on to exploit this possibility for automatic optimization.

43 Definition (Point-free form for Dodo). We call an identifier *predefined* if it is defined using a rewrite rule. For instance, *dogs* and *dogs′* from paragraph 39 have been predefined by the database administrator using equation (2.3), while *exr* and the Bag functor have been predefined by an extension writer. A Dodo expression is said to be in point-free form if it has first-order type $A \to B$ for and it is either

- a frame with all subexpressions either a column expression or also in point-free form,
- a predefined function, e.g., *exr*,
- a constant function $(\lambda x \bullet v) = const\ v$ with v predefined, e.g., $v = dogs$, or a primitive literal k.
- the composition of point-free Dodo expressions, e.g., $exl \circ exr$,
- or a predefined higher-order function operating on point-free Dodo expressions, e.g. Bag *exr*, or $f + g$.

Notice that expressions in point-free form no longer contain variables.

2.5.2 Translation to point-free form

In figure 2.3 we show how Dodo queries are translated to point-free form. We use the bracket notation $[\![E]\!]$ to denote the point-free form of E. The rewrite rules are grouped according to the paragraph in Section 2.2.2 they originate in. Notice that the case $[\![\lambda y \bullet f\ e]\!]$ with y occuring free in f is dealt with specially in section 2.5.3.

44 Assumed functions. The rewrite rules assume the existence of various predefined functions, such as $bool2sum : \mathbb{B} \to 1 + 1$, $\Delta : A \to A \times A$ and $\nabla : A + A \to A$. Definitions for those can be found in Chapter 3. Every comprehension type M requires a functor $\mathsf{T} = \mathsf{T}_M$ and monad functions $unit_M : A \to \mathsf{T}A$, $unnest_M : \mathsf{TT}A \to \mathsf{T}A$ and optionally $zero_M : X \to \mathsf{T}A$.

Moreover, the rewrite rules refer to a catamorphism (conversion function) $(\!|\tau_{\mathsf{T}'} \to \tau_M|\!)$. Again, we refer to paragraph 122 for a precise explanation, but informally, $\tau_{\mathsf{T}'}$ refers to the way the expression $(\mathsf{T}'\ (\lambda x \bullet e)\ xs)$ constructs a value of type $\mathsf{T}'A$, while τ_M refers to how the return value should be constructed

$$
\begin{array}{rcl}
\llbracket f \rrbracket & = & f \qquad \{\text{ if } f \text{ predefined }\} \\
\llbracket f\ e \rrbracket & = & f\ \llbracket e \rrbracket \qquad \{\ f \text{ higher-order due to type}\} \\
\llbracket f \circ g \rrbracket & = & \llbracket f \rrbracket \circ \llbracket g \rrbracket \\
\hline
\llbracket \lambda y \bullet k \rrbracket & = & \mathit{const}\ k \qquad \{\ k \text{ literal or a predefined frame }\} \\
\llbracket \lambda y \bullet \mathbf{let}\ x = e\ \mathbf{in}\ e' \rrbracket & = & \llbracket \lambda y \bullet e'_{x:=e} \rrbracket \\
\llbracket \lambda y \bullet (e, e') \rrbracket & = & \llbracket \lambda y \bullet e \rrbracket \times \llbracket \lambda y \bullet e' \rrbracket \circ \Delta \\
\llbracket \lambda y \bullet \mathbf{if}\ b\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\ \mathbf{fi} \rrbracket & = & \llbracket \lambda y \bullet \big(\nabla \circ ((\lambda_ \bullet e_1) + (\lambda_ \bullet e_2))\big)\ (\mathit{bool2sum}\ b) \rrbracket \\
\hline
\llbracket \lambda y \bullet y \rrbracket & = & \mathit{id} \\
\llbracket \lambda y \bullet f\ e \rrbracket & = & f \circ \llbracket \lambda y \bullet e \rrbracket \qquad \{\text{ if no } y \text{ in } f\ \} \\
\llbracket \lambda y \bullet f\ e \rrbracket & = & \{\text{ see section 2.5.3 if } y \text{ occurs free in } f\ \} \\
\llbracket \lambda y \bullet (f \circ g)\ e \rrbracket & = & \llbracket \lambda y \bullet f\ (g\ e) \rrbracket \\
\hline
\llbracket \lambda y \bullet M[e \mid\ ] \rrbracket & = & \llbracket \lambda y \bullet \mathit{unit}_M\ e \rrbracket \\
\llbracket \lambda y \bullet M[e \mid x \leftarrow e'] \rrbracket & = & \llbracket \lambda y \bullet \llparenthesis \tau_{\mathsf{T}'} \rightarrow \tau_M \rrparenthesis \big(\mathsf{T}'\ (\lambda x \bullet e)\ xs\big) \rrbracket \\
\llbracket \lambda y \bullet M[e \mid b] \rrbracket & = & \llbracket \lambda y \bullet M[e \mid _ \leftarrow \\
& & \quad \mathbf{if}\ b\ \mathbf{then}\ \mathit{unit}_M\ \dagger\ \mathbf{else}\ \mathit{zero}_M\ \dagger\ \mathbf{fi}] \rrbracket \\
\llbracket \lambda y \bullet M[e \mid qs, qs'] \rrbracket & = & \llbracket \mathit{unnest}_M M[M[e \mid qs'] \mid qs] \rrbracket \\
\hline
\llbracket \llparenthesis \alpha \rightarrow \beta \rrparenthesis \rrbracket & = & \llparenthesis \alpha \rightarrow \beta \rrparenthesis \\
\llbracket \mathit{const}\ v \rrbracket & = & \llbracket \lambda_ \bullet v \rrbracket
\end{array}
$$

Figure 2.3: Rewrite rules for translating Dodo expressions to point-free form.

according to comprehension type M. For example, if b is a bag, then $\mathit{Sum}[x \mid x \leftarrow b]$ is replaced by

$$\llparenthesis \tau_{\mathsf{Bag}} \rightarrow \tau_{\mathit{Sum}} \rrparenthesis (\mathsf{Bag}\ (\lambda x \bullet x)\ b).$$

In words, $(\mathsf{Bag}\ (\lambda x \bullet x)\ b) = b$ means "construct a bag out of the elements of b," τ_{Bag} means "construct a bag", τ_{Sum} means "add", and therefore the whole expressions means "add the elements of b."

45 Claim. Every Dodo expression of type $A \rightarrow B$ in which no free variables occur can be written in point-free form by repeated application of the rewrite rules in figure 2.3.

We give an outline of the proof. The first step is to verify that the right-hand sides of the equations in figure 2.3 are semantically equivalent to the left-hand sides. In most cases, this is trivial. For comprehensions, the prescribed semantics of the *unit*, *zero* and *unnest* functions are precisely that they should fulfill the equation. The extension writer should verify this for every new comprehension type.

The second step is to verify termination of the rewrite process. First observe that expressions which are point-free according to definition 43 do indeed reduce to themselves. Most rules in figure 2.3 either eliminate a syntactical construct which is never introduced (comprehension syntax, **if**, pair formation), or they yield an expression in which only strict subexpressions are enclosed in translation brackets. Such rewrite rules never contribute to nontermination.

One curious case involves the *const* v construct. The *const* operator is generated to alert subsequent rewriting stages to either insert a named frame containing data from the database, or generate a frame containing a primitive value k.

A λ-term $(\lambda x \bullet k)$, with k a *primitive* constant, is rewritten to *const* k, which is considered to be in normal form. Terms $[\![const\ v]\!]$ with v non-primitive are never generated by the system but may be written by users. In such cases, they are rewritten to $[\![\lambda x \bullet v]\!]$ and subjected to normal rewriting. Thus, although figure 2.3 contains both a rule mapping $[\![\lambda x \bullet _]\!]$ to *const* and one the other way around, this never leads to nontermination of the rewrite process.

46 Core predefs. In order to support the rewrite process described in section 2.5.2, 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 $\mathsf{1}$, $+$, $\times$ and Id, and predefs for id, Δ, $\times$, exl, exr, ∇, $+$ inl and inr as defined in section 2.2.1.

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 2.5.3 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 133, 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

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

so the rewrite process can continue with $(\lambda x \bullet e_1)$ and $(\lambda x \bullet e_2)$ separately.

2.5.3 Handling nested scopes

In section 2.5.2 we listed a set of rewrite rules for rewriting queries to point-free form. We skipped one case, that of $[\![\lambda y \bullet f\ e]\!]$ where y occurs in f. The problem here is that in point-free form, all variable bindings and references are eliminated, but as long as f depends on y we cannot eliminate y and $(\lambda y \bullet _)$. In this section we look at how to remove the dependence on y.

47 The easy cases. Let us enumerate all forms f can take. Because f has function type, all forms from paragraphs 29 and 30 do not apply. In paragraphs 31 and 32, we find five constructs which at the same time have type $X \to (A \to B)$ and may contain references to y:

- $[\![\lambda y \bullet (f'\ e')\ e]\!]$. See paragraph 48.
- $[\![\lambda y \bullet (f_1 \circ f_2)\ e]\!]$. Already covered by figure 2.3. Rewrites to $[\![\lambda y \bullet f_1\ (f_2\ e)]\!]$.
- $[\![\lambda y \bullet (\lambda x \bullet e')\ e]\!]$. This we can rewrite to $[\![\lambda y \bullet e'_{x:=e}]\!]$, eliminating the x.
- $[\![\lambda y \bullet (const\ v)\ e]\!]$. The easiest way to deal with this is to replace the *const*, yielding $[\![\lambda y \bullet (\lambda x \bullet v)\ e]\!]$. To this we can apply the previous rule.

The only nontrivial case is the case $f = (f'\ e')$, which we treat in the next paragraph.

48 The hard case. First we consider an example. Suppose we encounter $[\![\lambda y \bullet \mathsf{Bag}\ h\ e]\!]$ with $y : A$ and $e : \mathsf{Bag}\,B$. Given $y : A$, the function h has type $B \to C$. In other words,

$$y : A \vdash h : B \to C.$$

In a sense, h in $\lambda y \bullet \mathsf{Bag}\ h\ e$ is not applied to every element of the bag e, but to y and every element of e. We modify h to make this dependence on y explicit:

$$h' : A \times B \to C.$$

To obtain h', we introduce the parameter $z : A \times B$ representing both $y : A$ and an element $x : B$ of e. Our first attempt, $\big(\lambda z \bullet h(\mathit{exr}\ z)\big) : A \times B \to C$, still contains free occurrences of y in h. These must be replaced by references to the left-hand side of z:

$$h' := \big(\lambda z \bullet h_{y:=\mathit{exl}\ z}\ (\mathit{exr}\ z)\big) : A \times B \to C.$$

The function $\mathsf{Bag}\ h'$ has type $\mathsf{Bag}(A \times B) \to \mathsf{Bag}C$. The next question is: how to obtain a $\mathsf{Bag}(A \times B)$ given $y : A$ and $e : \mathsf{Bag}B$? To do this, we introduce the *distribution function*

$$D_{\mathsf{Bag}} : A \times \mathsf{Bag}B \to \mathsf{Bag}(A \times B)$$

which "distributes" the A over the Bag. For example,

$$D_{\mathsf{Bag}}\ (a, \{\!|1, 2|\!\}) = \{\!|(a, 1), (a, 2)|\!\}.$$

Using the distribution function we can complete the rewriting:

$$[\![\lambda y \bullet \mathsf{Bag}\ h\ e]\!] = \mathsf{Bag}\ [\![\lambda z \bullet h_{y:=\mathit{exl}\ z}\ (\mathit{exr}\ z)]\!]\ \circ D_{\mathsf{Bag}} \circ [\![\lambda y \bullet (y, e)]\!].$$

49 Scope unnesting over functor-like operators. The above example can be straightforwardly generalized to any operator Σ satisfying $\Sigma\ (f \circ g) = \Sigma f \circ \Sigma g$. For such Σ, the following equation can be used to eliminate the dependence of $\Sigma\, h$ on y:

$$[\![\lambda y \bullet \Sigma\ h\ e]\!] = \Sigma\ [\![\lambda z \bullet h_{y:=\mathit{exl}\ z}\ (\mathit{exr}\ z)]\!]\ \circ D_{\Sigma} \circ [\![\lambda y \bullet (y, e)]\!] \qquad (2.6)$$

where D_Σ is defined by the equation

$$\begin{aligned} D_\Sigma\ (y, xs) &= \Sigma\ (\lambda x \bullet (y, x))\ xs, && \text{(point-wise), or} \\ D_\Sigma \circ (y, _) &= \Sigma\ (y, _), && \text{(point-free)} \end{aligned} \qquad (2.7)$$

where we introduce the abbreviation $(y, _) = (\lambda x \bullet (y, x))$. Notice that for any new functor-like object Σ, an appropriate D_Σ must be defined at the frame level.

A proof for equation (2.6) is given by the following calculation:

$$
\begin{array}{ll}
 & \lambda y \bullet \Sigma\ h\ e \\
= & \quad \{\ h' = (\lambda z \bullet h_{y:=exl\ z}\ (exr\ z)) \text{ as in the previous paragraph } \} \\
 & \lambda y \bullet \Sigma\ (h' \circ (y, _))\ e \\
= & \quad \{\ \Sigma \text{ is functor-like } \} \\
 & \lambda y \bullet (\Sigma\, h' \circ \Sigma\, (y, _))\ e \\
= & \quad \{\ \text{Point-free definition of } D_\Sigma\ \} \\
 & \lambda y \bullet (\Sigma\, h' \circ D_\Sigma \circ\ (y, _))\ e \\
= & \\
 & \lambda y \bullet (\Sigma\, h' \circ D_\Sigma)\ (y, e) \\
= & \quad \{\ y \text{ is not free in } h' \text{ or } D_\Sigma\ \} \\
 & \Sigma\, h' \circ D_\Sigma \circ (\lambda y \bullet (y, e))
\end{array}
$$

For bifunctors, a similar result holds. Here are some examples of distribution functions:

$$
\begin{array}{rcl}
D_{\mathsf{Id}}\ (a, b) & = & (a, b) \\
D_\times\ (a, (b_1, b_2)) & = & ((a, b_1), (a, b_2)) \\
D_+\ (a, inl\ b) & = & inl\ (a, b) \\
D_+\ (a, inr\ b) & = & inr\ (a, b)
\end{array}
$$

If Σ is a composite operator, e.g., $\Sigma = FG$ or $\Sigma = F + G$, D_Σ can be constructed out of D_F and D_G. For instance, $D_{FG} = F\ D_G \circ D_F$:

$$FG\,(y, _) = F\,(G\,(y, _)) = F\,(D_G \circ (y, _)) = F\ D_G \circ F\ (y, _) = F\ D_G \circ D_F \circ (y, _).$$

50 Generalization to arbitrary higher-order functions. In the case of an arbitrary higher-order function Σ, the above trick does not work. In general, Σ has type $(X \to Y) \to F(X, Y) \to G(X, Y)$ for certain F and G. We can still construct $h' = (\lambda z \bullet h_{y:=exl\ z}\ (exr\ z))$, but the extension writer has to define a rewrite rule specifically for that Σ. The problem is that if $y : Z$, then $\Sigma\ h'$ has type $F(Z \times X, Y) \to G(Z \times X, Y)$, which means that it returns values of type $G(Z \times X, Y)$ rather than $G(X, Y)$. Of course, if X is not used in G, e.g., if $G(X, Y) = G'(Y)$ for some G', that does not matter.

51 SK combinators. It is interesting to compare the translation here to the theory of **S**, **K**, and **I** combinators [Tur79, PJ87]. With these combinators

it is possible to rewrite any λ-term to a point-free form. The combinators are given by the equations

$$\begin{aligned} (\lambda x \bullet x) &= \mathbf{I}, \\ (\lambda x \bullet c) &= \mathbf{K}\ c, \\ (\lambda x \bullet f\ e) &= \mathbf{S}\ (\lambda x \bullet f)\ (\lambda x \bullet e). \end{aligned} \tag{2.8}$$

Comparing these to Figure 2.3, it turns out that **K** and **I** are present there as *id* and *const*, respectively, but that there is no direct equivalent to **S**. The **S** operator performs scope unnesting, similar to what we do here in Section 2.5.3.

The reason for our lack of **S** operators is that a λ-term rewritten to **S**, **K**, **I** operators is still essentially a λ-term, but written in a form in which it can be more efficiently reduced. In our setting, however, we have as the back end a database kernel. That database kernel does not have sufficient expressive power to perform reductions, especially reductions on higher-order terms. Therefore, rewriting $(\lambda x \bullet \mathsf{Bag}\ f\ e)$ into $\mathbf{S}\ (\lambda x \bullet \mathsf{Bag}\ f)\ (\lambda x \bullet e)$ is not to our advantage, because it introduces a higher-order function $(\lambda x \bullet \mathsf{Bag}\ f)$, which we would rather avoid.

2.5.4 Further translation

Once the query has reached point-free form, we are not finished yet. Recall the example in paragraph 39. The first few steps, up to step $(E5)$, brought the query in point-free form. The later steps turn the query into a frame representation of the query result.

52 Pushing computation into the frames. After step $(E5)$ in paragraph 39, we have the query $\mathsf{Bag}\ \mathit{exr} \circ \mathit{dogs}' : \mathsf{1} \to \mathsf{Bag}(\mathit{SS} \times \mathbb{Z})$. The general form of the query is that it is a composition of functions, which are either operations from the nested level (Bag, exr) or frames containing data (dogs'). In the steps that follow, two things happen. Frame definitions such as equation 2.3 are expanded, and rewrite rules for complex operations are applied. The rewrite rules push the computation into the frames, either by rearranging the frames or by introducing new column operations. In the example, two such push-downs happen. The first is an application of the rule $\mathsf{Bag}\ f \circ \mathit{bag}\langle d, r, F\rangle = \mathit{bag}\langle d, r, f \circ F\rangle$. We will give a correctness argument for this rule later on. The second is $\mathit{exr} \circ \mathit{pair}\langle F, G\rangle = G$, which we already encountered as the point-free definition of exr in equation (2.5).

The general pattern for push-down rules is $f \circ F = F'$ with f an operation on nested data, F the frame representation of one or more values, and F' the frame representation of the values after f has been applied to them. In other

words, the rewrite rule constructs a new frame F' which incorporates the action of f into F. As mentioned in paragraph 42, the benefit of this is that such a rule is stated in bulk fashion. It specifies how to perform operation f on several values at the same time. In comparison to operations which work on an item at a time, such bulk operations can often be implemented more efficiently on the underlying platform.

53 Example. In chapter 3 we give example frame definitions for a number of types and rewrite rules for the operations on those types. We have already encountered the frame representation $bag\langle d, r, F\rangle$ of the Bag type given there. The interpretation function (paragraph 36) for $bag\langle\rangle$ frames is only defined for keys which occur in d. It specifies that when looking up a key in the frame, one should look up the matching element keys in r and then build a bag out of the elements obtained by looking up these keys in F. We will denote this as

$$bag\langle d, r, F\rangle\ x = \{\!|F\ y \mid (x, x) \in d \wedge (x, y) \in r|\!\}.$$

Notice that this is a point-wise definition of the semantics of the $bag\langle\rangle$-frame. To prove the correctness of the rewrite rule $\mathsf{Bag}\ f \circ bag\langle d, r, F\rangle = bag\langle d, r, f \circ F\rangle$, the extension writer can give an argument similar to the following:

$$
\begin{aligned}
& (\mathsf{Bag}\ f \circ bag\langle d, r, F\rangle)\ x \\
= & \quad \{ \text{ expand } \circ \} \\
& \mathsf{Bag}\ f\ (bag\langle d, r, F\rangle)\ x) \\
= & \quad \{ \text{ interpretation function } \} \\
& \mathsf{Bag}\ f\ \{\!|F\ y \mid (x, x) \in d \wedge (x, y) \in r|\!\} \\
= & \quad \{ \ \mathsf{Bag} \text{ is a functor } \} \\
& \{\!|f\ (F\ y) \mid (x, x) \in d \wedge (x, y) \in r|\!\} \\
= & \quad \{ \text{ introduce } \circ \} \\
& \{\!|(f \circ F)\ y \mid (x, x) \in d \wedge (x, y) \in r|\!\} \\
= & \quad \{ \text{ interpretation function } \} \\
& bag\langle d, r, f \circ F\rangle\ x
\end{aligned}
$$

Again, this is a point-wise argument for a point-free rule.

2.6 Extension writer obligations

When extending the system with a new data type, the extension writer has the following tasks:

- Add a new frame type. Choose a name for the new frame type and give a type signature relating the types of its components to the type of the frame. Example: $\mathit{pair}\langle\alpha \to A, \alpha \to B\rangle : \alpha \to (A \times B)$.

- Carefully define an interpretation function for the frame. This need not be code, but it must be clear and unambiguous. The database administrator needs to be able to use it while defining a mapping from nested data to flattened data, and the extension writer needs it to prove correctness of rewrite rules. If code is written, it either resides in Dodo itself or in the application built on top of Dodo. Example:

  $$\mathit{pair}\langle F, G\rangle\ k = \texttt{"("}\ \ (F\ k)\ \ \texttt{","}\ \ (G\ k)\ \ \texttt{")"}.$$

- Give implementations for the frame domain, frame translation and frame union operations. Example:

  $$\text{dom}\ \mathit{pair}\langle F, G\rangle = \text{dom}\ F = \text{dom}\ G, \quad r * \mathit{pair}\langle F, G\rangle = \mathit{pair}\langle r * F, r * G\rangle.$$

- If the new type can be regarded as a functor, declare so and give a rewrite rule for its map functionality. Example:

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

- Give implementations for type-specific operations by giving a type signature and rewrite rules. Example: $\mathit{mul} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, defined by $\mathit{mul} \circ \mathit{pair}\langle \mathit{atom}\langle f\rangle, \mathit{atom}\langle g\rangle\rangle = \mathit{atom}\langle \mathit{mul}(f, g)\rangle$.

- Implement any flat operators needed by the above rewrite rules, such as *mul* above. Give their type signature for use in the flat layer, and provide an implementation in the underlying database system. Such an implementation will typically be in written in C. Example:

  $$\begin{aligned}&\mathit{mul}(f : [\alpha \to \mathbb{Z}], g : [\alpha \to \mathbb{Z}]) = \\ &\qquad \{\!|(k, xy) \mid (k, x) \in f \wedge (k, y) \in g|\!\} : [\alpha \to \mathbb{Z}].\end{aligned}$$

- For any functor or higher-order function Σ, give a rewrite rule for D_Σ satisfying $D_\Sigma \circ (y, _) = \Sigma(y, _)$.

- If possible, designate an initial algebra for the type. (See chapter 5.) Also designate algebras for interesting operations and catamorphisms between them. (Again, see chapter 5.) Example: List has initial algebra τ_{List}. The function *sumlist* is the catamorphism $(\!|\tau_{\mathsf{List}} \to \underline{0} \triangledown (+)|\!)$.

- When possible, add a monad declaration to enable comprehension syntax for the new type, and maybe for some of its operations. This is only possible for operations which have an algebra (see above) declared for them.

2.7 Summary

Dodo is an extensible database with support for complex nested data types. Internally, the nested data types are stored in a flattened, non nested form to improve performance. The Dodo data model does not hide the mapping from nested structures to flattened structures, but exposes it to the extension writer. At the nested level, Dodo can be extended with new data types and operations. At the flattened level, with primitive (scalar) types and operations on (binary) relations over those types. Throughout Dodo, the notion of functions and function composition is used to guide the design of new data types and operations.

In the Dodo data model, *frames* are the bridge between the nested and the flat data model. They describe how to combine flattened information back into nested form. A frame always stands for a key→value mapping. The extension writer expresses the semantics of a frame type using an *interpretation function*, which is used to prove the correctness of rewrite rules. Nested operations are implemented by giving a rule for *composition*. For every operation f, rules are given which prescribe how to replace the composition $f \circ F$ of the operation with a frame by a new frame F' which denotes the values in F but with f applied to them.

Before these rules can be applied, the query first needs to be brought in "point-free form." This means that it is written as the composition of a sequence of nested operations. Most rules for bringing the query in point-free form are trivial; a few are not. Notable among the non-trivial rules are those dealing with nested scopes. These rules require additional helper functions to be defined by the extension writer for each new higher-order function.

Chapter 3

Realization

In the previous chapter we described the "Theory of Dodo." In this chapter we consider implementation aspects. We give concrete frame definitions and rewrite rules for a number of data types and explore the role of the type system in Dodo. At the flattened layer, we do not directly use SQL or MIL [BK99] the query language of MonetDB. Instead, we target an intermediate "column algebra." The column algebra is at approximately the same level of abstraction as MIL, but abstracts away from the physical representation of keys, see Section 3.2. Apart from allowing a more natural representation of composite key types (multi-attribute primary keys) in SQL, this also improves readability because it hides naming differences and bookkeeping chores. In general, we strife for every column operation to be easily implementable in MIL, possibly with the use of user defined functions.

In Section 3.1, we discuss some elementary and often used column operations, giving a taste of the column algebra and clearing the way for later sections. In Section 3.2 we consider *sum* and *product* key types, which often arise during query processing in Dodo. We explain when they arise, and introduce abstract functions *mksum* and *mkprod* to hide their implementation details. We also discuss how to implement them on MonetDB and SQL. In Section 3.3 we describe a number of data types as Dodo extensions. This serves both as an illustration of what Dodo extensions look like to the programmer, and to provide a baseline Dodo implementation for use in later chapters. In Section 3.4 we briefly discuss Dodo's type system and possible improvements to it. Improving Dodo's type inference and propagation mechanism directly leads to opportunities for Dodo to emit better flat query plans.

54 Bag columns semantics. The columns (binary relations) 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 that whenever we take the union of two sets, the sets are disjoint. It does mean, however, that the semantics of the column operations are defined in terms of bag-braces $\{\!\!\{\ \}\!\!\}$ rather than set-braces $\{\ \}$. For instance, we write

$$twin(c : [\alpha - \beta]) = \{\!\!\{(x, x) \mid (x, y) \in c\}\!\!\} : [\alpha - \alpha] \tag{3.1}$$

rather than

$$twin(c : [\alpha - \beta]) = \{(x, x) \mid (x, y) \in c\} : [\alpha - \alpha].$$

55 Type notation and propagation of properties. Recall from paragraph 34 the notation $[K - L]$ for a binary relation ("column") with heads in K and tails in L. The dash in the middle can be adorned with markers for special properties. For instance, $[K \rightarrow L]$ denotes a many-to-one relation, and $[K \vdash L]$ means that all elements in K are present as a head. Moreover, we write $[K -!]$ for columns with the property that every head is equal to its tail.

Many operators propagate column properties of their arguments. For instance, for general columns $c : [\alpha - b]$, the result of $twin(c)$ has type $[\alpha -!]$. However, if c is head-distinct or head-complete, then so is the result of *twin*. In paragraph 35, we introduced the type predicate $\mathcal{C}_{op}$ to embody this information. It is tempting to try to incorporate it also in type signatures such as (3.1), but this quickly becomes unreadable. Therefore we choose to give only the minimal set of properties in these signatures. If a column operator is given as taking arguments with properties, this means that the properties *must* be present for the operation to be well defined. If the return value of an operator has properties, these are known to be always present, regardless of the argument properties. Other information in C_{op} has no special notation, but will sometimes be noted in the surrounding text.

3.1 Basic column operators

The choice of the operations here is based on the ideas in Section 2.3. We keep the number of data types in the flat layer as small as possible, allowing only scalar types and binary relations between scalar types. In particular, we use binary identity relations to implement sets (unary relations). This makes it possible to implement things such as domain restriction of tables using the normal join operators. However, practical considerations may make it more

attractive to have a column algebra in which such operations are present as as explicit operators. Fortunately, the line of thinking underlying the translation scheme of Chapter 2 does not depend on the particular form of the core algebra.

56 Column creation. The schema defined by the DBA (paragraph 20) provides access to the existing data in the database through named columns. In this paragraph we introduce some operators useful for constructing new columns during query evaluation. The first operator is *emptycol*. It creates an empty column, the type of which is to be determined through type inference.

$$emptycol() = \{\!|\,|\!\} : [K \leftrightarrow L].$$

The second is *colpair*, which takes a pair of primitive values and returns a column containing just this pair:

$$colpair(x : K, y : L) = \{\!|(x, y)|\!\} : [K \leftrightarrow L].$$

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

$$\begin{aligned}
converse(c : [K - L]) &= \{\!|(y, x) \mid (x, y) \leftarrow c|\!\} : [L - K] \\
sethead(c : [K_1 - K_2], v : L) &= \{\!|(v, y) \mid (x, y) \leftarrow c|\!\} : [L - K_2] \\
settail(c : [K_1 - K_2], v : L) &= \{\!|(x, v) \mid (x, y) \leftarrow c|\!\} : [K_1 - L] \\
twin(c : [K - L]) &= \{\!|(x, x) \mid (x, y) \leftarrow c|\!\} : [K -!] \\
rtwin(c : [K - L]) &= \{\!|(y, y) \mid (x, y) \leftarrow c|\!\} : [L -!]
\end{aligned}$$

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 and assert head and tail completeness properties:

$$\begin{aligned}
twinu(c : [K - L]) &= distinct\ \{\!|(x, x) \mid (x, y) \leftarrow c|\!\} : [K \leftrightarrow!], \\
rtwinu(c : [K - L]) &= distinct\ \{\!|(y, y) \mid (x, y) \leftarrow c|\!\} : [L \leftrightarrow!].
\end{aligned}$$

For readability, we will often abbreviate $converse(r)$ to $r^{\cup}$.

58 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.

$$\begin{aligned}
op_{not}(b : [\alpha \to \mathbb{B}]) &= \{\!|(h, \neg v) \mid (h, v) \leftarrow b|\!\} : [\alpha \to \mathbb{B}] \\
op_{and}(b_1 : [\alpha \to \mathbb{B}], b_2 : [\alpha \to \mathbb{B}]) &= \{\!|(h, b_1(h) \wedge b_2(h)) \mid (h, x) \leftarrow b_1|\!\} : [\alpha \to \mathbb{B}] \\
op_{or}(b_1 : [\alpha \to \mathbb{B}], b_2 : [\alpha \to \mathbb{B}]) &= \{\!|(h, b_1(h) \vee b_2(h)) \mid (h, x) \leftarrow b_1|\!\} : [\alpha \to \mathbb{B}]
\end{aligned}$$

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

$$op_{eq}(f_1 : [\alpha \to K], f_2 : [\alpha \to K]) \quad = \quad \{\!|(h, f_1(h) = f_2(h)) \mid (h, x) \leftarrow f_1|\!\} : [\alpha \to \mathbb{B}].$$

In paragraph 76 we implement the function $bool2sum : \mathbb{B} \to 1 + 1$ using two column operators *selecttrue* and *selectfalse* defined such that for every f, there is a partitioning (α_1, α_2) of the heads of f such that

$$\begin{aligned} selecttrue(f : [\alpha \to \mathbb{B}]) &= \{\!|(h, \dagger) \mid (h, x) \leftarrow f,\ x \text{ true}|\!\} : [\alpha_1 \mapsto 1] \\ selectfalse(f : [\alpha \to \mathbb{B}]) &= \{\!|(h, \dagger) \mid (h, x) \leftarrow f,\ x \text{ false}|\!\} : [\alpha_2 \mapsto 1] \end{aligned}$$

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

3.2 Key representation

59 Total functions. Key types are used as the domain type for frames. Frames are *total functions*, i.e., they assign a value to *every* key in their domain. In some respects, it would have been easier to allow frames to be *partial* functions, which do not have to assign a value to every key. However, after experimenting with a theory of Dodo based on partial functions, we have come to the conclusion that using partial functions makes it much harder to reason about frame expressions, because there is no simple way anymore to guarantee that frames actually contain certain data. Attempts to improvise such ways boil down to reinventing total functions.

Therefore, in Dodo frames are total functions and thus require special types to serve as their domain. For instance, if we have a frame describing dogs and a frame describing people, the dog frame has as its domain a special "dog identifier" type containing precisely as many keys as there are dogs in the database. Likewise, the people frame has a special-purpose "people identifier" type.

The special purpose types described here are *static*, in the sense that they denote sets of identifiers present in the current database. It is of course possible that the database changes, but for the duration of the query evaluation process, it can be considered immutable. During query evaluation, however, frames are constructed representing partial query results. Such frames also need special purpose types to denote their domain. In the rest of this section, we examine how such *dynamic* special purpose types occur, and how they are handled by Dodo's type system.

60 Dynamic special-purpose types. Given the special-purpose types representing database data, the question arises which types are used as the domain of the frames used as intermediate stages during rewriting and evaluation. For example, in a query such as

$$Bag[(personname\ p, dogname\ d) \mid p \leftarrow persons, d \leftarrow dogs],$$

the resulting $bag\langle d, r, G\rangle$ frame contains one pair for every person-dog combination. Therefore, the $pair\langle\rangle$ frame G enclosed by the $bag\langle\rangle$ frame must have $|persons|$ times $|dogs|$ keys in its domain. As another example, suppose we have two frames $F_1 = bag\langle d_1, r_1, G_1\rangle : \alpha_1 \rightarrow \mathsf{Bag}\, A$ and $F_2 = bag\langle d_2, r_2, G_2\rangle : \alpha_2 \rightarrow \mathsf{Bag}\, A$ with α_1 and α_2 disjoint. According to Section 2.6 and paragraph 38, the $bag\langle\rangle$ frame must have a rewrite rule which can combine F_1 and F_2 into a frame $F_1 \sqcup F_2 = bag\langle d, r, G\rangle : (\alpha_1 \cup \alpha_2) \rightarrow \mathsf{Bag}\, A$. However, if G_1 has type $\beta \rightarrow A$ and G_2 has type $\gamma \rightarrow A$, in general we cannot simply throw the elements in G_1 and G_2 in one heap: $G_1 \sqcup G_2$ is only defined if β and γ are disjoint subsets of a common base type, which need not be the case. And even if it were the case that β and γ are both subsets of, say, the integers, then we have no way to make sure they are disjoint.

61 Making up new types. In situations such as the above, the Dodo formalism allows to simply *assume* the existence of a suitable type, and use helper operations to obtain relations between this type and known keys. For instance, in the situation where we have a column $r : [\alpha - \beta]$ of person-dog pairs, we create a special purpose type γ to represent the pairs. We use operations $mkprod()$, $prodleft()$ and $prodright()$ to obtain the relationship between γ, α and β:

$$\begin{aligned} r &: [\alpha - \beta] \\ mkprod(r) &: [\alpha \leftarrowtail \gamma] \\ prodleft(r) &: [\gamma \mapsto \alpha] \\ prodright(r) &: [\gamma \mapsto \beta] \end{aligned}$$

How γ, $mkprod()$, $prodleft()$ and $prodright()$ are implemented, is up to the underlying database. On MonetDB, with its machine oriented binary model, α, β and γ are most likely all subsets of a variant of the integer type. In that case, $mkprod()$ can be implemented using a simple row-numbering operation. For an example, see paragraph 62.

In the other example, where $G_1 : \beta \rightarrow A$ and $G_2 : \gamma \rightarrow A$ had to be combined into a single frame, $G : ? \rightarrow A$, we first obtain $g_1 = \mathrm{dom}\ G_1 : [\beta \leftrightarrow !]$ and $g_2 = \mathrm{dom}\ G_2 : [\gamma \leftrightarrow !]$. These relations represent the domains of G_1 and G_2.

Then, we posit a type δ containing precisely $|g_1| + |g_2|$ elements, and obtain the relationship between β, γ and δ:

$$\begin{array}{rcll} g_1 & : & [\alpha_1 - \beta] & \\ g_2 & : & [\alpha_2 - \gamma] & \text{with } \alpha_1 \cap \alpha_2 = \emptyset \\ \mathit{mksum}(g_1, g_2) & : & [(\alpha_1 \cup \alpha_2) \hookleftarrow \delta] & \\ \mathit{sumleft}(g_1, g_2) & : & [\delta \rightarrow \beta] & \\ \mathit{sumright}(g_1, g_2) & : & [\delta \rightarrow \gamma] & \end{array}$$

This is elaborated in paragraph 63.

62 mkprod. Together with the type signatures

$$\begin{array}{rcl} \mathit{mkprod}([\alpha - \beta]) & : & [\alpha \hookleftarrow \gamma], \\ \mathit{prodleft}([\alpha - \beta]) & : & [\gamma \mapsto \alpha], \\ \mathit{prodright}([\alpha - \beta]) & : & [\gamma \mapsto \beta], \end{array}$$

the defining characteristic of *mkprod*(), *prodleft*() and *prodright*() is

$$\mathit{prodleft}(r)^{\cup} * \mathit{prodright}(r) = r. \tag{3.2}$$

Essentially, this means that the operations neither lose information, because r can still be reconstructed, nor make up new information, because the type signature states that γ is complete. Thus, $\mathit{mkprod}(r)$ constructs γ such that there is a one-to-one mapping between γ and the rows of r. In an integer-based key representation, this can easily be implemented using a row numbering operator. In the following example, row numbering arbitrarily starts at 100:

r		$\mathit{mkprod}(r)$		$\mathit{prodleft}(r)$		$\mathit{prodright}(r)$	
1	10	1	100	100	1	100	10
1	20	1	101	101	1	101	20
2	30	2	102	102	2	102	30
3	40	3	103	103	3	103	40
4	40	4	104	104	4	104	40

Notice that *prodleft*() is simply the converse of *mkprod*(). We include it anyway because of symmetry with *mksum*().

63 mksum. Similarly, with $\alpha_1 \cap \alpha_2 = \emptyset$, *mksum*() is characterized by the signatures

$$\begin{array}{rcl} \mathit{mksum}([\alpha_1 - \beta], [\alpha_2 - \gamma]) & : & [(\alpha_1 \cup \alpha_2) \hookleftarrow \delta] \\ \mathit{sumleft}([\alpha_1 - \beta], [\alpha_2 - \gamma]) & : & [\delta \rightarrow \beta] \\ \mathit{sumright}([\alpha_1 - \beta], [\alpha_2 - \gamma]) & : & [\delta \rightarrow \gamma] \end{array}$$

together with the equations

$$\begin{aligned} mksum(r_1, r_2) * sumleft(r_1, r_2) &= r_1, \\ mksum(r_1, r_2) * sumright(r_1, r_2) &= r_2. \end{aligned}$$

As with *mkprod*, these equations ensure no information is lost, while the type signature ensures no extra information is contributed. Here is an example of how *mksum* can implemented using unions and row numbering operations:

r_1	
1	a
2	b
2	c

r_2	
4	P
5	Q
6	Q

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

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

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

64 Order-preserving versions. We assume our primitive types, or at least the ones used as index types, to be equipped with an ordering. The ordering of the element-keys is used in the *list*⟨⟩ frame to represent the order of the elements in the list. This 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:

$$\begin{gathered} x < y \\ \Longleftrightarrow \\ (prodleft_ord(r)(x),\ prodright_ord(r)(x)) \\ < (prodleft_ord(r)(y),\ prodright_ord(r)(y)). \end{gathered}$$

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

$$\begin{aligned} \forall (d, b), (d', b') \in sumleft_ord(r_1, r_2) \bullet \quad & d < d' \iff b < b' \\ \forall (d, c), (d', c') \in sumright_ord(r_1, r_2) \bullet \quad & d < d' \iff c < c' \end{aligned}$$

Here is an example for *mksum_ord*, consider in particular p and q.

r_1	
1	a
3	b
5	c

r_2	
2	q
4	p

sumleft_ord	
10	a
11	b
12	c

sumright_ord	
20	p
21	q

mksum_ord	
1	10
2	21
3	11
4	20
5	12

Notice 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.

3.3 Data type definitions

The operations defined here are the bare minimum needed to demonstrate interesting cases. For types, that means $\mathbb{B}$, $\mathbb{Z}$ and SS 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, e.g., $decimal : \mathbb{Z} \to SS$. We have an $lb : \mathsf{List}A \to \mathsf{Bag}A$ operation and a $sum : \mathsf{Bag}A \to \mathbb{Z}$ operation, both catamorphisms. 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. 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.

65 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:

$$id \circ F = F.$$

66 Implementing constant functions. The constant function constructor *const* from paragraph 32 is used for two purposes. The first is to introduce literals into the point-free form, e.g., the simple query 3 has point-free form *const* 3, which is translated using the rule

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

The other use is to introduce named values defined in the Dodo schema. Recall Equation (2.3), where $dogs' = (\lambda x \bullet dogs)$ is defined as

$$(\lambda w \bullet dogs) = dogs' = bag\langle d, r, tuple\langle atom\langle f\rangle, atom\langle g\rangle\rangle\rangle$$

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

$$const\ dogs \circ G = settail(\text{dom}\ \ G, \dagger) * F.$$

What happens here is that $settail(\text{dom}\ \ G, \dagger)$ creates a column that maps every key in α (the domain of G) to $\dagger$. This column is then used to transform F from domain $\mathbf{1}$ to domain α.

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

$$atom\langle[\alpha \mapsto K]\rangle : \alpha \to K$$

To look up a value in an atom frame, look it up in the binary relation it wraps:

$$atom\langle f\rangle\ x = f(x).$$

Frame rules:

$$\begin{array}{rcl} empty & = & atom\langle emptycol()\rangle \\ dom\ atom\langle f\rangle & = & twin(f) \\ r * atom\langle f\rangle & = & atom\langle r * f\rangle \\ atom\langle f_1\rangle \sqcup atom\langle f_2\rangle & = & atom\langle f_1 \cup f_2\rangle \end{array}$$

Because the $\sqcup$ operator forbids overlapping domains, the frame union of two $atom\langle\rangle$ frames is simply the column union of its columns.

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

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

To look up a key in a pair frame, look it up in its two components and combine the results into a pair:

$$pair\langle F, G\rangle\ x\ \ =\ \ \texttt{"("}\ \ F\ x\ \ \texttt{","}\ \ G\ x\ \ \texttt{")"}$$

Frame rules:

$$\begin{array}{rcl} empty & = & pair\langle empty, empty\rangle \\ \text{dom}\ \ pair\langle F, G\rangle & = & \text{dom}\, F = \text{dom}\, G \\ r * pair\langle F, G\rangle & = & pair\langle r * F, r * G\rangle \\ pair\langle F_1, G_1\rangle \sqcup pair\langle F_2, G_2\rangle & = & pair\langle F_1 \sqcup F_2, G_1 \sqcup G_2\rangle \end{array}$$

Recall from paragraph 23 that pairs are constructed using the $\vartriangle$ operator, transformed using $\times$ and destructed using *exl* and *exr*. We give rules for these operations, plus the distribution function $D_\times$.

$$
\begin{aligned}
(f \vartriangle g) \circ F &= \mathit{pair}\langle f \circ F, g \circ F\rangle \\
(f \times g) \circ \mathit{pair}\langle F, G\rangle &= \mathit{pair}\langle f \circ F, g \circ G\rangle \\
\mathit{exl} \circ \mathit{pair}\langle F, G\rangle &= F \\
\mathit{exr} \circ \mathit{pair}\langle F, G\rangle &= G \\
D_\times \circ \mathit{pair}\langle H, \mathit{pair}\langle F, G\rangle\rangle &= \mathit{pair}\langle \mathit{pair}\langle H, F\rangle, \mathit{pair}\langle H, G\rangle\rangle
\end{aligned}
$$

69 The *either* frame. The $\mathit{either}\langle F, G\rangle$ frame denotes a sum type. The idea is that the key exists either in F or in G. The type rule is

$$\mathit{either}\langle \alpha_1 \to A, \alpha_2 \to B\rangle : \alpha \to (A + B),$$

with (α_1, α_2) a partitioning of α, i.e., $\alpha_1 \cup \alpha_2 = \alpha$ and $\alpha_1 \cap \alpha_2 = \emptyset$. To look up a key k in a frame $\mathit{either}\langle F, G\rangle$, check in which domain it occurs and return the appropriate value:

$$
\begin{aligned}
\mathit{either}\langle F, G\rangle\ x &= \mathit{inl}\ (F\ x) \quad \text{if } x \text{ occurs in dom } F, \\
\mathit{either}\langle F, G\rangle\ x &= \mathit{inr}\ (G\ x) \quad \text{if } x \text{ occurs in dom } G,
\end{aligned}
$$

The required frame rules and construction, destruction and distribution functions:

$$
\begin{aligned}
\mathit{empty} &= \mathit{either}\langle \mathit{empty}, \mathit{empty}\rangle \\
\text{dom } \mathit{either}\langle F, G\rangle &= \mathit{domF} \sqcup \mathit{domG} \\
r * \mathit{either}\langle F, G\rangle &= \mathit{either}\langle r * F, r * G\rangle \\
\mathit{either}\langle F_1, G_1\rangle \sqcup \mathit{either}\langle F_2, G_2\rangle &= \mathit{either}\langle F_1 \sqcup F_2, G_1 \sqcup G_2\rangle \\
\mathit{inl} \circ F &= \mathit{either}\langle F, \mathit{empty}\rangle \\
\mathit{inr} \circ F &= \mathit{either}\langle \mathit{empty}, F\rangle \\
(f + g) \circ \mathit{either}\langle F, G\rangle &= \mathit{either}\langle f \circ F, g \circ G\rangle \\
(f \triangledown g) \circ \mathit{either}\langle F, G\rangle &= (f \circ F) \sqcup (g \circ G) \\
D_+ \circ \mathit{pair}\langle H, \mathit{either}\langle F, G\rangle\rangle &= \mathit{either}\langle \mathit{pair}\langle \text{dom } F * H, F\rangle, \mathit{pair}\langle \text{dom } G * H, G\rangle\rangle
\end{aligned}
$$

In the rule for $(f \triangledown g)$, we first calculate $f \circ F$ and $g \circ G$ which have type $\alpha_1 \to A$ and $\alpha_2 \to A$, respectively. Because the $\mathit{either}\langle\rangle$ frame guarantees that α_1 and

α_2 are a partition of α, we construct the result by simply combining $f \circ F$ and $g \circ G$ using $\sqcup$. Notice that in the rule for D_+, we have to write dom $_ * H$ in order to get the domains in the $pair\langle\rangle$ frames right: F is only defined on α_1 while H is defined on the whole α.

70 The *bag* frame. The $bag\langle d, r, F\rangle$ frame is used to represent bags. Most items from the shopping list in Section 2.6 are present here. Type rule:

$$bag\langle[\alpha \mapsto\!\!\!\!\rightarrow !], [\alpha - \beta], \beta \to A\rangle : \alpha \to \mathsf{Bag} A$$

In a frame $bag\langle d, r, F\rangle$, 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.

Here are the required frame operations for bags:

$$\begin{aligned}
\operatorname{dom}\ bag\langle d, r, F\rangle &= d \\
r' * bag\langle d, r, F\rangle &= bag\langle twin(r' * d), r' * r, F\rangle \\
bag\langle d_1, r_1, F_1\rangle \sqcup bag\langle d_2, r_2, F_2\rangle &= bag\langle d_1 \sqcup d_2, mksum(r_1, r_2), \\
&\qquad sumleft(r_1, r_2) * F_1 \\
&\qquad \sqcup sumright(r_1, r_2) * F_2\rangle
\end{aligned}$$

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.

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

$$\begin{aligned}
zero_{\mathsf{Bag}} \circ F &= bag\langle \operatorname{dom} F, emptycol(), empty\rangle \\
unit_{\mathsf{Bag}} \circ F &= bag\langle \operatorname{dom} F, \operatorname{dom} F, F\rangle \\
unnest_{\mathsf{Bag}} \circ bag\langle d_1, r_1, bag\langle d_2, r_2, F\rangle\rangle &= bag\langle d_1, r_1 * r_2, F\rangle \\
\mathsf{Bag}\ f \circ bag\langle d, r, F\rangle &= bag\langle d, r, f \circ F\rangle \\
D_{\mathsf{Bag}} \circ pair\langle F, bag\langle d, r, G\rangle\rangle &= bag\langle d, mkprod(r), \\
&\qquad pair\langle prodleft(r) * F, \\
&\qquad\qquad prodright(r) * G\rangle\rangle \\
\text{Comprehension type } Bag &: (bag, \mathsf{Bag}, (zero_{\mathsf{Bag}}, unit_{\mathsf{Bag}}, unnest_{\mathsf{Bag}}))
\end{aligned}$$

The *unit* rule takes the frame F and uses it three times in the result. Because $unit_{Bag} \circ F$ has the same domain as F, the result frame gets $\operatorname{dom} F$ as its first component. It gets $\operatorname{dom} F$ as the outer-inner relation because every bag

it constructs (outer) contains precisely one element (inner). We use here that 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.

71 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 $list\langle d, r, F\rangle$ with $r : [\alpha - \beta]$, the ordering of the β keys determines the order of the elements in the list. An alternative implementation would be not to use ordering of the element keys, but simply store an additional column containing position numbers (ordinals). This approach is explored in Section 4.2.1.

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

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

Assume the types $r_1 : [\alpha - \beta]$ and $r_2 : [\beta - \gamma]$. 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

$$\begin{array}{ll} & unnest_{List} \circ list\langle d_1, r_1, list\langle d_2, r_2, F\rangle\rangle \\ = & list\langle d_1, r_1 * mkprod_ord(r_2), prodright_ord(r_2) * F\rangle, \end{array}$$

the expression $mkprod_ord(r_2)$ gives a relation between the β-keys and new identifiers for the (β, γ)-pairs in r_2, so $r_1 * mkprod_ord(r_2)$ 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 (see paragraph 64). In the third component, we translate the elements in F to the new properly ordered key space.

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

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

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

73 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

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

Assuming types $d : [\alpha \mapsto\!\!\!\rightarrow !]$, $r : [\alpha - \beta]$ and $f : [\beta \mapsto \mathbb{Z}]$, the mystery column XX must have type $\alpha \to \mathbb{Z}$ and maps each $a \in \alpha$ to the sum of the numbers in the relational image $(r * f)(a)$. We take $XX = op_{sum}(d, r * f)$ where op_{sum} is the column operator which sums the tails of its second argument while grouping the heads by its first argument. Due to the possibility that $(r * f)(a) = emptyset$, the translation not only depends on $r * f$ but also on d. Keys which occur in d but not in $r * f$ are assigned sum 0. Because in SQL, empty groups are assigned `NULL`, 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. Of course, if the head-type is 1, the grouping could be omitted.

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 $Sum[x^2 \mid x \leftarrow some_bag_or_list]$.

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

74 The functions *any* and *all*. The functions *any* and *all* are entirely similar to *sum*, except that they calculate the logical disjunction ($\vee$) with unit element *false* and the logical conjunction ($\wedge$) 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*.

$$any \quad : \quad \mathsf{Bag}\mathbb{B} \to \mathbb{B}$$

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

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

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

75 The *decimal* function. The function *decimal* maps integers to their decimal string representation.

$$
\begin{aligned}
decimal &: \mathbb{Z} \rightarrow SS \\
decimal \circ atom\langle f\rangle &= atom\langle op_{decimal}(f)\rangle
\end{aligned}
$$

76 Boolean operations. Booleans can very well be implemented as sum types $1 + 1$, but in the prototype we 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.

$$
\begin{aligned}
not &: \mathbb{B} \rightarrow \mathbb{B} \\
and &: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B} \\
or &: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B} \\
bool2sum &: \mathbb{B} \rightarrow 1 + 1 \\
not \circ atom\langle b\rangle &= atom\langle op_{not}(b)\rangle \\
and \circ pair\langle atom\langle b\rangle, atom\langle b'\rangle\rangle &= atom\langle op_{and}(b, b')\rangle
\end{aligned}
$$

$$\begin{aligned} or \circ pair\langle atom\langle b\rangle, atom\langle b'\rangle\rangle &= atom\langle op_{or}(b, b')\rangle \\ bool2sum \circ atom\langle b\rangle &= either\langle selecttrue(b), selectfalse(b)\rangle \end{aligned}$$

77 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.

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

78 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:

$$\begin{aligned} & eq \circ pair\langle pair\langle F_1, G_1\rangle, pair\langle F_2, G_2\rangle\rangle \\ = \; & and \circ pair\langle eq \circ pair\langle F_1, F_2\rangle, eq \circ pair\langle G_1, G_2\rangle\rangle. \end{aligned}$$

79 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.

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

where

$$\begin{aligned} LL &= eq \circ pair\langle(\mathrm{dom}\, F_1 \cap \mathrm{dom}\, F_2) * F_1, (\mathrm{dom}\, F_1 \cap \mathrm{dom}\, F_2) * F_2\rangle \\ LR &= (\mathrm{dom}\, F_1 \cap \mathrm{dom}\, G_2) * const\ false \\ RL &= (\mathrm{dom}\, G_1 \cap \mathrm{dom}\, F_2) * const\ false \\ RR &= eq \circ pair\langle(\mathrm{dom}\, G_1 \cap \mathrm{dom}\, G_2) * G_1, (\mathrm{dom}\, G_1 \cap \mathrm{dom}\, G_2) * G_2\rangle \end{aligned}$$

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. An alternative implementation would be a higher-order function eq' that calculates equality only on the intersection of its arguments domains, together with a separate operator *override* that helps insert the default value *false* where the domains to not overlap. This yields

$$\begin{aligned} & eq \circ pair\langle either\langle F_1, G_1\rangle, either\langle F_2, G_2\rangle\rangle \\ = \; & override(const\ false, eq'(F_1, F_2) \sqcup eq'(G_1, G_2)) \end{aligned}$$

For primitive types, the eq' operator maps nicely to operations supported by most platforms:

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

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

3.4 The Dodo type system

81 Type inference. Dodo uses a type inference system. The goal of type inference is to determine the type of the query result and of the subexpressions of the query automatically, with as little user input as possible. We begin with a small example of why this is useful. Recall the *empty* frame operator from paragraph 38. It is used by rewrite rules when they need the frame representation for an an empty frame, i.e., a frame $\emptyset : \emptyset \to A$. An example of such a situation is the rewrite rule for the sum type constructor $\mathit{inl} : A \to A + B$:

$$\mathit{inl} \circ F = \mathit{either}\langle F, \mathit{empty}\rangle.$$

Here, all data (F) is pushed to the left side of the frame, but something must be put on the right side. The idea is that the the rule for *inl* simply writes *empty*, and that for every type B, there is an instance $\mathit{empty}_B : \emptyset \to B$ which constructs a frame of the proper type.

Type inference makes it possible for the rewrite rules to simply write *empty* and let other parts of the system (other extensions) worry about which frame structure should be put there.

82 Unification. Type inference takes place by assigning each subexpression a *type template* containing both concrete types (known) and *type variables* (not yet known). The semantics of the language give rise to constraints on the type variables. For instance, the expression $(f\ \ e)$ can only have type B if $e : A$ and $f : A \to B$ for certain A. These constraints can be expressed as a set of equations on the type variables, and solved in a process called *unification*, which yields the most general assignment to the type variables which satisfies all constraints [Pie02]. See figure 3.1 for an example of this. In the example, the most general type assignment for $\nabla \circ \mathit{either}\langle F, \mathit{empty}\rangle$ is calculated. After

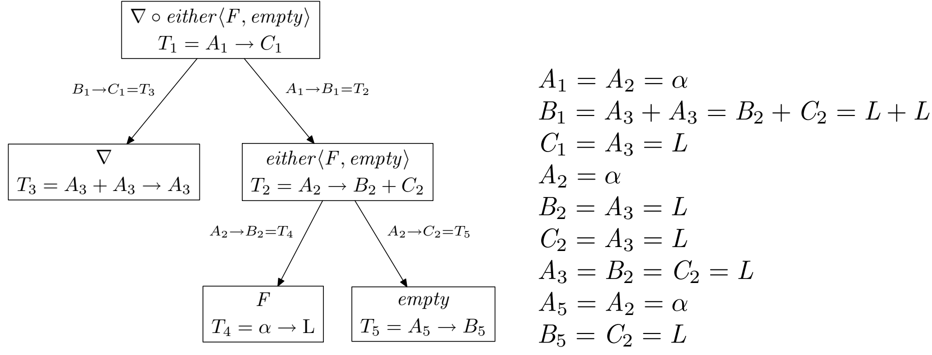


Figure 3.1: Every node in the abstract syntax tree (AST) is assigned a type template, and every edge in the AST contributes equations on the type variables $A_1, \ldots, B_5$. The type templates of $\nabla = id \triangledown id$ and $F : \alpha \to L$ are given in the data dictionary. Solving the equations allows us to conclude that *empty* is used here with type $A_5 \to B_5 = \alpha \to L$.

unification, it is known that *empty* is used here with type $\alpha \to L$. If L is a primitive type, it should be implemented as an $atom\langle\rangle$ frame.

Referring back to paragraph 67, where the $atom\langle\rangle$ frame is defined, we see that the next step in the translation process is the replacement of *empty* by $atom\langle emptycol()\rangle$, yielding

$$\nabla \circ either\langle F, atom\langle emptycol()\rangle\rangle.$$

Thanks to the derived type of *empty*, we now know its replacement is an $atom\langle\rangle$ frame.

83 Error checking. The most obvious use for type inference is, of course, error checking. If unification is unable to find a solution, the constraints on the type variables must be contradictory, which indicates a mistake in the query or rewrite rule.

In a system based on rewrite rules, type checking is also useful for intermediate steps. If the initial query is well-typed, the intermediate stages should also be well-typed. The Dodo prototype has a switch which turns on type checking for every intermediate step, thus pinpointing the rewrite rule that introduced the mistake.

In many cases, it is possible to verify the types not when the rewrite rule is applied, but when it is defined. If the rewrite rule can expressed purely using pattern matching, for instance,

$$(\lambda \mathtt{x} \bullet (\mathtt{e}_1, \mathtt{e}_2)) = (\lambda \mathtt{x} \bullet \mathtt{e}_1) \vartriangle (\lambda \mathtt{x} \bullet \mathtt{e}_2),$$

the type checking can even be performed at rule compile time.

84 Key space management. In Section 3.2, we discussed key spaces, that is, special purpose types which represent collections of certain objects. In paragraph 61 we stated that "the Dodo formalism allows to simply *assume* the existence of a suitable type, and use helper operations to obtain relations between this type and known keys." By now, we can be more explicit over what happens: during type checking, the column expression $mkprod(r)$ gets assigned a template type variable as usual. During unification, this type variable is then adorned with annotations about the key spaces it is a product of, and also a reference to the expression out of which the new type is created, in this case r. A similar thing happens with $mksum(r_1, r_2)$ and both associated $*left()$ and $*right()$ functions.

The annotations about the key spaces are used by the back end to choose the representation: in MonetDB, the representation is simply another object-id counter, but in a SQL system, product keys could be implemented using multi-attribute primary keys. The reference to the underlying relations is used to detect mismatches between the use of $mkprod()$ and $prodright()$ and $prodleft()$, and also for readability and optimization purposes.

85 Abstract prodleft and prodright. As stated above, product and sum key spaces created using $mkprod()$ and $mksum()$ carry a reference to the arguments of the keyspace operator which created them. This is used for error detection, but it also has other uses. These uses are related to issues which arise when a composite key space is used in more then one place in a frame, which is often the case. Consider the frame

$$bag\langle d, \mathcal{F}(mkprod(r)), \mathcal{G}(prodright(r))\rangle$$

The $\mathcal{F}$ signifies that the relation part of the $bag\langle\rangle$ frame somehow uses $mkprod(r)$ to construct a composite keyspace γ out of $r : [\alpha - \beta]$, and $\mathcal{G}$ that the item part of the frame uses $prodright()$, and possibly $prodleft()$ to take it apart again and use α and β in a certain way.

Dodo contains many rewrite rules which optimize column expressions. Some of these might affect $mkprod(r)$. For instance, in some contexts, we might wish to replace $r' * mkprod(r)$ by $mkprod(r' * r)$. This is not in general correct, but

it might be correct in certain contexts. However, if we do this replacement in $\mathcal{F}$, the type γ is replaced by some γ' and expression $\mathcal{G}(prodright(r))$ becomes incorrect, because it uses the wrong relation as its input. In summary, it can be hard to *locally* optimize expressions which use $mkprod()$, because modifications can have consequences in remote parts of the query.

Another, related, problem is that in practice, the r in $mkprod(r)$ can become quite a large expression, which makes intermediate results hard to read and hinders experimentation with rewrite rules. The solution we have chosen in Dodo is that we introduced *abstract* versions of $mkprod()$ and friends, where r is given only as an argument to $mkprod$, now called $amkprod(r)$. The abstract destructors $aprodleft()$ and $aprodright()$ have the same meaning as their concrete counterparts, but do not repeat r. The reason this works is that although r is relevant to the back end, where a concrete representation of the composite key must be chosen, but it is irrelevant during query translation at the Dodo level. At the Dodo level, the only thing that matters is that keys from γ can be mapped back to α and β, which is the function of $aprodleft()$ and $aprodright()$. See figure 3.2 for a graphical comparison of parse trees with and without abstract composites.

86 Using column properties for optimization. Careful propagation of column properties leads to many local optimization opportunities. Frame-level rewrite rules are generally written under the assumption that a later step will, for instance, eliminate common subexpressions. Moreover, for simplicity, many rules generate subexpressions which can be proven to be unnecessary if column properties are taken into account.

Consider for instance, the operators $mkprod(r)$, $prodleft(r)$ and $prodright(r)$. Given a relation $r : [\alpha - \beta]$, these construct a new keyspace γ identifying the rows of table r. However, if r is one-to-many, then every row can be identified by its tail and no expensive renumbering operations are necessary: if r has type $[\alpha \leftarrow \beta]$, we can take $\gamma \subset \beta$ and put

$$\begin{aligned} mkprod(r) &= r, \\ prodleft(r) &= converse(r), \\ prodright(r) &= rtwin(r). \end{aligned}$$

For example,

r	
1	10
1	20
2	30

$mkprod(r)$	
1	10
1	20
2	30

$prodleft(r)$	
10	1
20	1
30	2

$prodright(r)$	
10	10
20	20
30	30

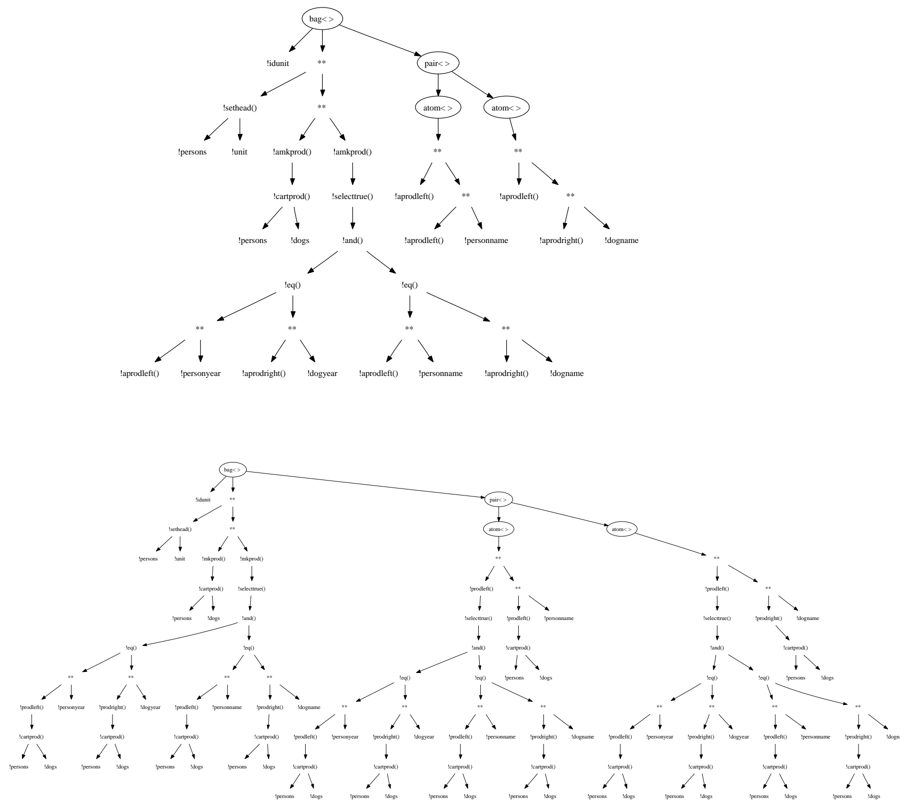


Figure 3.2: Parse trees of two rewritten Dodo expressions, one with abstract composite keys, and one without. It is not hard to guess which one is harder to debug.

In typical relational implementations, both *converse* and *rtwin* can be implemented purely as table header manipulations and thus be evaluated in constant time.

Notice, however, that the above is not completely type-safe. The definition of *mkprod* in paragraph 61 requires the new type γ to contain precisely one key for each row: $mkprod(r) : [\alpha \hookleftarrow \gamma]$. If we take $\gamma = \beta$ as above, this is no longer necessarily the case. From the perspective of the type system, the above optimizations are only applicable if $r : [\alpha \hookleftarrow \beta]$. Obviously, it would be very beneficial if the type system could be relaxed in such a way that it can handle the above optimization even if $r : [\alpha \leftarrow \beta]$. That is the subject of the next paragraph.

87 Subset typing. This and the next paragraph describe future work — already partially realized — that elaborates some ideas described up to now.

To make better use of the optimization opportunities mentioned in the previous paragraph, it would be desirable to have a more fine-grained notion of completeness than the one provided by the head- and tail completeness properties. We can attempt to better estimate which subset of a key type occurs in a relation. For instance, the `if then else` construct, or more precisely, the function $bool2sum : \mathbb{B} \to \mathbf{1} + \mathbf{1}$, uses two column operations *selecttrue*() and *selectfalse*() which split a key set α into two sets α_1 and α_2 for which a predicate is true or false, respectively. Often, after some computation, the two key spaces are later combined again into α. This is not hard to see: the keys are split in order to compute the **then**-clause on α_1 and the **else**-clause on α_2, but afterwards, they are recombined into a frame which contains all results of the **if then else** expression. It would be very nice if Dodo could statically determine that all keys in α occur again in this frame.

It is possible to do so if we introduce predicate annotations on key types. Every time a choice is made out of the keys in a key space, a corresponding predicate is assigned, which yields true for one set, and false for the other. Key spaces can now be decomposed into subsets on which predicates or known to true or false. For instance, if we denote the negation of p as $\overline{p}$, we have

$$\alpha = \alpha_p \cup \alpha_{\overline{p}}, \quad \alpha_p \cup \alpha_p = \alpha_p, \quad \alpha_p \cap \alpha_p = \alpha_p, \text{ and } \alpha_p \cap \alpha_{\overline{p}} = \emptyset.$$

Moreover, we have distributivity:

$$\alpha_p \cap (\alpha_{q_1} \cup \alpha_{q_2}) = \alpha_{pq_1} \cup \alpha_{pq_2}.$$

88 Unification of subset types. Until now, unification of types always yielded a set of type equations. However, for subset types, it yields a set of subset nullifications. Suppose we have α_1 and α_2 subsets of α, and at some

point, these types must be unified. If we can write α_1 and α_2 as unions of disjoint subsets

$$\begin{aligned}\alpha_1 &= \alpha_{x_1} \cup \ldots \cup \alpha_{x_n} \cup \alpha_{y_1} \cup \ldots \cup \alpha_{y_k},\\ \alpha_2 &= \alpha_{x_1} \cup \ldots \cup \alpha_{x_n} \cup \alpha_{z_1} \cup \ldots \cup \alpha_{z_k},\end{aligned}$$

where the x_i, y_i and z_i are different combinations of predicates p_1, $\overline{p}_1$, p_2, $\overline{p}_2$, etcetera, and where the α_{x_i} are in common between α_1 and α_2 while α_{y_i} and α_{z_i} are not, then

$$\alpha_1 = \alpha_{x_1} \cup \ldots \cup \alpha_{x_n} \cup \alpha_{y_1} \cup \ldots \cup \alpha_{y_k} = \alpha_2 = \alpha_{x_1} \cup \ldots \cup \alpha_{x_n} \cup \alpha_{z_1} \cup \ldots \cup \alpha_{z_k}$$

implies

$$\alpha_1 = \alpha_{x_1} \cup \ldots \cup \alpha_{x_n} = \alpha_2$$

and thus, $\alpha_{y_i} = \alpha_{z_i} = \emptyset$ for all i. In other words, the relationship between subsets can completely be characterized by listing sets of predicates of which the conjunction is known to be always false.

Implementation-wise, keeping track of these sets can be done fairly straightforward using a tree structure on predicates. We expect this method to open the door to much stronger optimizations than are possible using simple completeness-flags.

3.5 The prototype

We have developed a prototype Dodo system to test our ideas on query flattening. The prototype consists of a type checker, a rewrite engine, a collection of type definitions and rewrite rules similar to Section 3.3, and an optional graphical user interface (GUI).

89 Rewriting in the prototype. Syntactically, the prototype follows the language described in Chapter 2. The user enters a query expression, either fully at the nested level, e.g., *Bag*[...], or already partially rewritten towards column expressions. The first step is the type inference step, in which type information is attached to the abstract syntax tree (AST) of the expression. If the type of the expression is not a function type, the query is first wrapped into a constant lambda term, as explained in paragraph 39.

Then follows a series of iterations in which parts of the query are offered to a list of rewrite rules. One of the matching rules is asked to perform a substitution on the tree. If more than one rule matches, the user can interactively select in

the GUI which of the rules is chosen. This facilitates experimentation. In non-interactive mode, the leftmost substitution is chosen.

After each rewrite step, the intermediate result is (optionally) presented to the user and (optionally) type checked again. Intermediate results can presented textually, but also in graph format, as depicted in Figure 3.2. In the graph format, it is possible to perform common subexpression elimination to improve readability.

90 Rule verification for extension writers. A Dodo extension consists of frame definitions, including rewrite rules to implement standard frame functionality, declarations of operations, with rewrite rules implementing them in a point-free manner, and generally also extensions at the column level, in the form of new primitive data types and column operations. The Dodo framework specifies relations between all these elements, but in practice, it may be hard for the extension writer to keep track of it all.

In order to assist the extension writer in writing correct rewrite rules, our prototype provides a mode in which the type inference mechanism verifies the type after every rewrite step. A chang in type would indicate an incorrect rewrite rule. This has been very useful in tracking down problems. A similar idea could be applied at the column level, where the type system predicts properties such as head-completeness or injectivity, which can be checked at run-time.

Most rewrite rules purely operate at the syntactical level. For example, the rule $exl \circ pair\langle F, G\rangle = F$ does not use type information. In the current prototype such a rule is implemented simply by stating the two patterns, e.g., `"exl .. pair<$f, $g>"` and `"$f"`. Though not currently implemented, it is straightforward to type check such rules at rule insertion time, rather than when the rule is applied. This, too, should aid the extension writer in building a correct extension.

3.6 Summary

In this chapter we have examined the role of type information in Dodo. In particular, we examined key space management. Key spaces are used in Dodo to denote the domains of data functions. If a data function stores k values, its domain should consist of k keys. This gives rise to a multitude of different *key spaces*, which should be handled sufficiently abstract to allow for multiple implementations, but expressive enough to handle the requirements of extension writers. In Section 3.2 we introduced column operators *mkprod*, *prodleft*, *prodright*, *mksum*, *sumleft* and *sumright* to deal with this.

We also gave concrete Dodo implementations of several common data structures used in the prototype. For rewriting, among others, the empty frame

operator, it is necessary to derive type information from the surrounding query. We developed a type system for Dodo, and discussed how it could be improved to allow more extensive optimization.

Chapter 4

Pathfinder

91 Pathfinder. Pathfinder [GST04, BGvK$^+$06], now part of MonetDB/XQuery, is a relational approach to XQuery processing. Using a clever relational encoding of XML documents, it compiles XQuery queries into efficient relational algebra expressions, which it executes on a relational back-end, currently MonetDB. Boncz et al. [BGvK$^+$06] report high performance and scalability using this approach, querying documents up to 11 GB in size with time scaling almost linearly with document size. The key to Pathfinder's excellent performance lies in the combination of its document representation, which allows efficient, sort-free evaluation of XPath axis steps, and a technique called "loop lifting" [GT04, GST04] which eliminates nested loop evaluation of for-let-where-return (FLWOR) expressions in favor of efficient table-based bulk operations.

92 Query flattening. Being a compiler for queries over a nested model (XML) to a flat model (a relational algebra), the Pathfinder approach is an instance of query flattening (Chapter 1, paragraph 2). Therefore, as an example of an independently developed full-blown real-world system, it is suitable for validation of our approach. Validation questions are

1. Can Dodo's core principles be observed in Pathfinder?
2. Can the Pathfinder approach be implemented as Dodo extensions?
3. With hindsight, would Dodo and its core principles been helpful in the development of Pathfinder would they have been known at the time?

It turns out that several essential Pathfinder algorithms have equivalent counterparts in Dodo, in particular loop lifting, which turns out to be the Pathfinder

equivalent of how nested scopes are eliminated in Dodo using distribution functions D_* (Section 2.5.3). Moreover, we show how, once the Pathfinder-specific data types are implemented as Dodo extensions, we get loop lifting and the associated book-keeping for free.

4.1 Pathfinder in a nutshell

93 Essence. The essence of Pathfinder lies in its encoding of XML trees and item sequences. The tree encoding enables efficient XPath axis traversal, and the sequence encoding enables efficient processing of nested XQuery expressions. Following Grust, Sakr and Teubner [GST04], we begin with the data structures and see how the algorithms follow naturally from them.

94 Tree encoding. The tree encoding used in Pathfinder [GvKT03] is based on the pre- and post-order rank of the nodes in the tree. In summary, this means that we record the order in which a linear scan through an XML document encounters the open and closing tag of each element. Storing nodes by pre-rank, post-rank and level allows efficient positional comparisons between nodes. For instance, if node n_2 is a *descendant* of node n_1, that means that when serialized, n_2 *starts later* and *ends earlier*, i.e.,

$$n_1.pre < n_2.pre \;\wedge\; n_2.post < n_1.post. \tag{4.1}$$

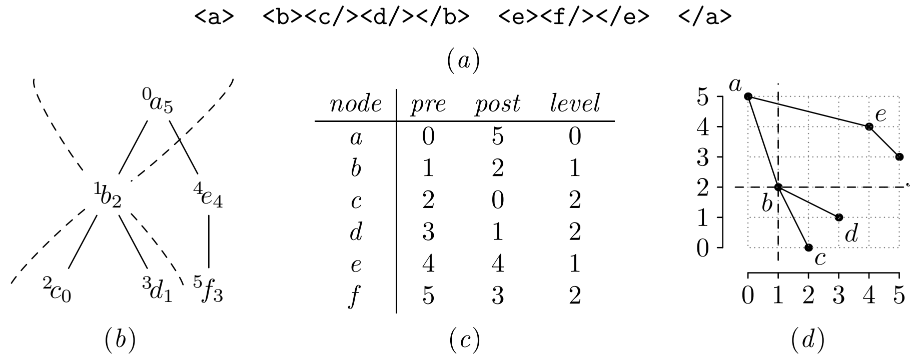
See figure 4.1 for an example. The actual MonetDB/XQuery implementation uses a scheme based on pre-rank, size of subtree, and level, which is equivalent due to the equation

$$n.pre + n.size = n.post + n.level,$$

but has several implementation advantages. For presentation purposes, however, we often use the pre/post scheme.

Using conditions such as (4.1) as join conditions, one can do XPath axis traversal in bulk. In the XML tree navigation language XPath, the tree is traversed along XPath *axes* such as *descendant*, *child*, *parent* or *following*. Assuming the document is represented as a pre/post table T, and that we have a table t containing the pre-numbers of a set of nodes, called the *context node set*, the set of nodes reachable from nodes in t along axis s can be computed using the join $t \bowtie_s T$, where $\bowtie_s$ uses an equation such as (4.1) as a selection criterion.

Notice that the result can be used as the input to another join step: to find the nodes reachable through a path $./step_1/\ldots/step_n$ from an initial context node set t, one simply computes $t \bowtie_{step_1} \cdots \bowtie_{step_n} T$.



node	*pre*	*post*	*level*
a	0	5	0
b	1	2	1
c	2	0	2
d	3	1	2
e	4	4	1
f	5	3	2

Figure 4.1: (a) an XML document; (b) the document, drawn as a tree, with pre-rank indicated by superscripts and post-rank by subscripts; (c) tabular representation of the pre- and post-ranks; (d) ranks used as coordinates in the pre/post plane. The dashed lines indicate how comparisons on pre/post rank distinguish between ancestors, descendants, preceding and following nodes of node b.

95 Staircase join. Axis steps can be regarded as region queries in the pre/post plane, which can efficiently be computed on standard relational databases using a B-tree index [Gru02]. It is, however, possible to do better if we extend the database with tree-aware join algorithms. The staircase join [GvKT03] is a family of specialized join algorithms for join conditions similar to (4.1), which take the tree shape of the underlying data into account in order to avoid the costly sorting and duplicate removal phases required by XPath semantics [FMM$^+$05].

The staircase join takes two arguments. The first is a table t of pre-numbers denoting the context node set. The other is the document table T consisting of pre/post/level tuples and other information. Both tables are assumed to be ordered in document order, i.e., on the pre-attribute. The output of the staircase join operation for an axis s is a new table of pre-numbers, listing the nodes in T reachable from t along axis s.

How staircase joins take the tree shape into account is illustrated in figure 4.2. In case (a), context node p completely dominates q, that is, $q/descendant$ ::⊂ $p/descendant$. Thus, q can safely be omitted (pruned) from the context. Omitting dominated nodes gives rise to the characteristic staircase shape from which the algorithms derive their name. In case (b), no node can simultaneously have both p and q as an ancestor. Thus, during a simultaneous sequential scan of both the context set and the node table (going from left to right in the pre/post

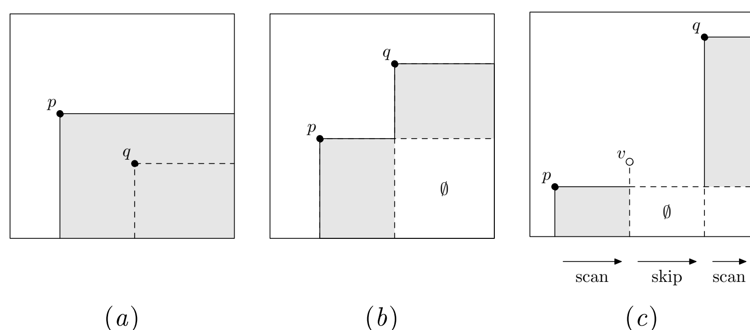


Figure 4.2: Tree awareness in the staircase join for the *descendant* axis step: (a) pruning, (b) partitioning and (c) skipping. Nodes in the context set are drawn closed, other nodes, e.g., v in case (c) are drawn open.

plane), only a single context node needs to be considered at a time. Clearly, the end result is again sorted in document order and without duplicates.

Evaluating condition (4.1) using a "normal" join would in the *descendant* case not generate duplicates either, but might unnecessarily touch nodes in the document table because it is not aware that the descendants of p and q never overlap. Along other axes, partitioning does help eliminate duplicates.

Finally, in case (c) we see that irrelevant parts of the document table can be skipped. On encountering p in the context, the nodes following p in the document table are copied to the output table until the first node with a higher post-rank than p is found. In the diagram, this is v.

As v is a follower node of p, it has no common descendants with p. All other nodes between v and q are guaranteed to also follow p. Therefore, on encountering v, all tuples in the document table up to the next context node q can be skipped, and because the document table is ordered on pre-rank, finding q can be done very efficiently.

The use of staircase joins leads to impressive performance improvements. For instance, the number of nodes touched by a descendant step is $|context|+|result|$, which is the absolute minimum: In order to generate the result, all nodes in it have to be touched, and in order to determine the result, the whole input context sequence has to be processed.

In this example, we concentrated on the *descendant* step, but for the other axis steps, similar properties can be derived. Implementations of staircase joins exist in MonetDB [BGvK+06], currently the preferred Pathfinder back end, but also for the open source RDBMS PostgreSQL [MGvKT04].

96 Loop lifting. The XQuery language is fully composable. That is, every expression may be used as a building block for a larger expression. Pathfinder is an XQuery compiler. It takes an XQuery query and compiles it to a relational algebra query. In doing so, it takes full advantage of composability. Translation of a subexpression E takes place in an "iteration context" determined by the enclosing *for* expressions (scope). By iteration context we mean the sequence of all variable bindings for which the expression is to be evaluated. For example, in the query

```
for $x in (10,20)   for $y in (100,200)   return $x+$y,
```

which yields the sequence $(110, 210, 120, 220)$, the outer `for` is evaluated once, the inner `for` is evaluated twice, and the `return` clause is evaluated four times, each time with other bindings for $\$x$ and $\$y$. The foundation of Pathfinder's compilation strategy is that every language construct which affects control flow, such as `for` and `if`, does so by manipulating a tabular representation of the iteration context. Before we make that more precise, we first consider *item sequences.*

97 Item sequences. In the XQuery 1.0 and XPath 2.0 data model (XDM) [FMM+05], every expression evaluates to a sequence of items. Such an item is either an atomic value, such as a string or a number, or it is a node in an XML tree. The semantics of XDM are such that item sequences are never nested: expressions are considered to yield a *stream* $(x_1, x_2, \ldots)$ of items. As a consequence of this stream-based world view, an individual item x is always considered equivalent to the singleton stream (x).

In Pathfinder, every item sequence is represented as a table consisting of ordinal/item pairs. The ordinals are simply numbers indicating the position of the corresponding item in the stream. The items are strings and numbers in the case of atomic items, or pre-rank numbers in the case of an XML node. Thus, in the example query above, the result of evaluating the expression `(10,20)` is a table

(10,20)	
0	10
1	20

98 Loop lifting, cont'd. The point of loop lifting is that when Pathfinder generates code, it does not generate code which performs a single evaluation of that expression, it generates code which performs the evaluations for all iterations at once, in parallel. The compiled version of the expression takes as an argument a table of all circumstances in which it will be evaluated, and returns a table containing all results. For item sequences, these tables consist

not of pairs, but of triples (*iter*, *ordinal*, *value*). The *iter* column identifies the iteration context in which a tuple was generated. We explain loop lifting in more detail by means of a few examples.

99 Example 1, for ... in. We will construct tabular representations for the intermediate stages in the evaluation of the query

`for $x in (10,20)   for $y in (100,200)   return $x+$y`. (4.2)

The tabular representations consists of tuples ($iter_i, pos, item$). The *pos* and *item* fields are as described in the previous paragraph. The *iter* fields are used to tell apart the different parallel evaluations of the expression. The first step is to evaluate the subexpression `(10,20)`. In this step, there is only a single iteration (parallel evaluation) going on, numbered 0:

	`(10,20)`	
$iter_0$	*pos*	*item*
0	0	10
0	1	20

(4.3)

The `for $x in` construct takes this table and constructs a table with all bindings for `$x` for each iteration. This can be accomplished using renumbering, which turns the single iteration 0 containing the sequence $(10, 20)$ into two iterations, containing the singleton sequences (10) and (20). At the same time, it constructs a mapping table between $iter_1$ and $iter_0$ for later reference.

$iter_1$	$iter_0$
0	0
1	0

	`$x`	
$iter_1$	*pos*	*item*
0	0	10
1	0	20

(4.4)

We see that there are now two iterations, with different bindings for `$x`. In the next step, the subexpression `(100,200)` is evaluated in both iterations at the same time, resulting in:

$iter_1$	$iter_0$
0	0
1	0

$\bowtie$

	`(100,200)`	
$iter_0$	*pos*	*item*
0	0	100
0	1	200

$=$

	`(100,200)`	
$iter_1$	*pos*	*item*
0	0	100
0	1	200
1	0	100
1	1	200

(4.5)

We clearly see how $iter_1$ distinguishes between the two sequences, while the (unlabelled) *pos* column distinguishes items within sequences. The `for $y in`

construct uses this to construct $iter_2$ and a new mapping table, which we use to *translate* `$x` to $iter_2$:

$iter_2$	$iter_1$
0	0
1	0
2	1
3	1

	`$y`	
$iter_2$	*pos*	*item*
0	0	100
1	0	200
2	0	100
3	0	200

	`$x`	
$iter_2$	*pos*	*item*
0	0	10
1	0	10
2	0	20
3	0	20

(4.6)

Finally, `$x+$y` can now be computed using an efficient loop lifted version of the addition operator. The result can be brought back to $iter_0$ by joining with the mapping tables and renumbering the *pos* column:

	`$x+$y`	
$iter_2$	*pos*	*item*
0	0	110
1	0	210
2	0	120
3	0	220

	`for $x in ... return $x+$y`	
$iter_0$	*pos*	*item*
0	0	110
0	1	210
0	2	120
0	3	220

(4.7)

This yields the required sequence $(110, 210, 120, 220)$.

100 Example 2, if . . . then. Other control flow, such as `if then else`, is implemented similarly. Consider the query

```
for $x in (10,20)
  for $y in (100,200)
  let $sum := $x+$y
    return
      if ($sum mod 3 = 0)
      then ($sum, " is a triple. ")
      else ($sum, " is no triple. ")
```

This query yields the result "110 is no triple. 210 is a triple. 120 is a triple. 220 is no triple." The first part of the evaluation goes exactly the same as in paragraph 99. Then, the branch condition is evaluated, and the context table is split in two: $iter_{2a}$, where the condition is true, and $iter_{2b}$, where it is false. Now, the `then`-clause can be evaluated in the true context, and the `else`-clause

in the false context:

	`$sum mod 3=0`			`then ...`			`else ...`	
$iter_2$	*pos*	*item*	$iter_{2a}$	*pos*	*item*	$iter_{2b}$	*pos*	*item*
0	0	*false*	1	0	210	0	0	110
1	0	*true*	1	1	`"is a"`	0	1	`"is no"`
2	0	*true*	2	0	120	3	0	220
3	0	*false*	2	1	`"is a"`	3	1	`"is no"`

(4.8)

The final result is obtained by merging the tables and translating back from $iter_2$ to $iter_0$.

101 Loop lifted staircase join. Axis steps often occur inside loops and benefit similar to the addition operator from implementing a special loop lifted version of the staircase join. MonetDB/XQuery's loop lifted staircase join [BGvK+06] applies the axis step not to a single context node sequence, but to all context node sequences of all iterations simultaneously. It retains the important property that it needs at most a single sequential scan over the document table to do so. Hence, tree navigation is effectively transformed into a join.

102 Performance benefits. Especially in the presence of large volumes of data, the loop lifted version of the addition operator can be orders of magnitude faster than repeated application of the normal operator. Loop lifted operators are bulk operators. Rather than being invoked again and again for every argument, the loop lifted addition operator is invoked once, with two streams of numbers as arguments, and produces another stream of numbers as the result. Streaming access means predictable memory access patterns and fewer branch mispredictions, resulting in a huge performance boost. The same holds for all loop lifted operations.

This performance boost is very noticable even in comparisons between relational databases. Profiling experiments [BZN05] with MySQL show that even during a simple sequential table scan, the database kernel spends only about 10% of its time in the functions which do the actual work, in this experiment, multiplication integers. The rest of the time is spent on looking up fields in records and other overhead. MonetDB, on the other hand, stores its data in simple, array-like structures, requiring nothing more than a pointer increment to look up the next number. The result is that on average, MonetDB spends 3 CPU cycles per number pair, compared to 49 for MySQL. Obviously, for systems supporting more complex data structures than tables, the difference will be even larger.

Loop lifting allows Pathfinder to evaluate subexpressions in bulk, replacing tree navigation by table manipulation and joins. In fact, some of the most frequently used table manipulations are (*i*) replacing a column, filling it with a densely increasing sequence of integers, (*ii*) replacing all values in a column with a constant, usually 0. It is interesting to notice that in MonetDB, such operations can be done in constant time, simply by manipulating the table header. Many SQL based platforms offer similar functionality in the form of the OLAP extension `RANK/DENSE_RANK` [GST04] which, though probably not as cheap as MonetDB, can still be evaluated quite efficiently.

103 Conclusion. Pathfinder is an example of a system where choosing a suitable bulk oriented data representation, in this case the pre/post plane, makes a huge difference in the efficiency of algorithms. The classic staircase join takes advantage of the fact that we often wish to traverse an XPath axis not from one context node, but from a set of context nodes. The loop lifted staircase join generalizes this further, traversing the axis for many context node sets at once. Allowing clever extension writers to pick such a data representation while still allowing convenient item-at-a-time style querying at the top level is exactly the motivation behind Dodo.

The most important parallel between Pathfinder and Dodo is loop lifting. The core principle of Dodo is to decompose a data type into a flattened representation of a collection, and to enable vectorized execution of query operators. Dodo achieves this by rewriting the query in point-free form, which introduces bulk operations on complex data, and mapping the complex bulk data to flattened storage structures to enable vectorization. Loop lifting in Pathfinder is very similar to the transformation of queries to point-free form and the decomposition into flat operators proposed in Dodo for general data types.

4.2 Pathfinder extension to Dodo

Notice that the above way of storing XML trees does not involve nested data types per se, which are the focus of Dodo. It is, however, a clever way of storing nested data in a flat way and translating queries to the flattened representation. One of the major points of the multi-model architecture and Dodo is that it explicitly puts the mapping from nested data to flat data in the hands of the extension writer, thus enabling the programmer to use a flattened representation which is very different from the conceptual data model. If the mapping would have been automatic, one would most likely end up with XML trees represented as a bunch of linked objects, with all the performance problems that entails.

104 New frames. In this section we consider how Pathfinder data structures

can be implemented in Dodo, and we look at how Dodo flattens XQuery queries. We introduce two new data types with their associated frames:

$$\mathit{xmlnode}\langle\rangle : \nu \to \mathit{NODE} \quad \text{and} \quad \mathit{seq}\langle\rangle : \alpha \to \mathit{Seq}\, X\,.$$

The *NODE* type is an abstract type encoding all information about a single XML tree node, including its relationship with other nodes. Notice that a *NODE* is a specific node in the XML tree graph, rather than a whole subtree. To clarify the distinction, a node has parents and children, whereas a tree has subtrees and enclosing trees. The $\mathit{xmlnode}\langle\rangle$ frame maps keys to nodes. We usually use the type letter ν ("nu") for node keys. All nodes a node is related to in any way are understood to live within the same $\mathit{xmlnode}\langle\rangle$ frame.

The $\mathit{seq}\langle\rangle$ frame represents a mapping from keys to sequences of values. It is similar to the $\mathit{list}\langle\rangle$ and $\mathit{bag}\langle\rangle$ frames of Chapter 3, but has explicit support for the $\mathit{pos}()$ position operator.

105 Iteration context. XQuery expressions are defined [BCF+05] to evaluate to a sequence of items, where items are either atomic items or nodes. The iteration context tables we encountered in the section about loop lifting are in Dodo represented as frames of type $\alpha \to \mathsf{Seq}(\mathit{SS} + \mathit{NODE})$. In this type, the keys α function as the iteration counter in the context table. To every iteration (α) corresponds a sequence (Seq) of items ($\mathit{SS} + \mathit{NODE}$), which are either atomic values, here assumed to be strings (SS), or nodes (NODE).

As a consequence of the type structure $\alpha \to \mathsf{Seq}(\mathit{SS} + \mathit{NODE})$, the result of evaluating an XQuery query will have the frame structure

$$\mathit{seq}\langle \ldots, \mathit{either}\langle \mathit{atom}\langle\ldots\rangle, \mathit{xmlnode}\langle\ldots\rangle\rangle\rangle,$$

where $\mathit{seq}\langle\rangle$ represents Seq, $\mathit{atom}\langle\rangle$ denotes the atomic items and $\mathit{xmlnode}\langle\rangle$ denotes the XML nodes in the sequence.

4.2.1 The $\mathit{seq}\langle\rangle$ frame

The $\mathit{seq}\langle\rangle$ frame is similar to the $\mathit{list}\langle\rangle$ frame of Chapter 3, except that our implementation carries an extra binary relation to store ordinals in.

106 Frame definition. The $\mathit{seq}\langle\rangle$ frame has type rule

$$\mathit{seq}\langle[\alpha \mapsto\!\!\!\rightarrow !], [\alpha \leftarrow \beta], [\beta \to \mathbb{N}], \beta \to X\rangle : \alpha \to \mathsf{Seq} X\,.$$

As with lists, in $\mathit{seq}\langle d, r, p, F\rangle$, d gives the domain of the frame, r gives the relation between sequence identifiers and items, p gives the position of every item in the sequence, and F gives the items themselves. The result of looking up

key k in a $seq\langle d, r, p, F\rangle$ is the sequence of items such that for every $(k, k') \in r$, item $F(k')$ occurs on position $p(k')$. The required rewrite rules (Section 2.6) are as follows:

$$\begin{aligned}
empty &= seq\langle emptycol, emptycol, \\
&\qquad emptycol, empty\rangle, \\
dom\ seq\langle d, r, p, F\rangle &= d, \\
r' * seq\langle d, r, p, F\rangle &= seq\langle twin(r' * d), mkprod(r' * r), \\
&\qquad prodright() * p, F\rangle, \\
seq\langle d_1, r_1, p+1, F_1\rangle \sqcup seq\langle d_2, r_2, p_2, F_2\rangle &= seq\langle d_1 \cup d_2, mksum(r_1, r_2), \\
&\quad sumleft() * p_1 \cup sumright() * p_2, \\
&\quad sumleft() * F_1 \cup sumright() * F_2\rangle \\
\mathsf{Seq}\ f \circ seq\langle d, r, p, F\rangle &= seq\langle d, r, p, f \circ F\rangle \\
D_{\mathsf{Seq}} \circ pair\langle H, seq\langle d, r, p, F\rangle\rangle &= seq\langle d, r, p, pair\langle r^{\cup} * H, F\rangle\rangle \\
zero_{\mathsf{Seq}} \circ F &= seq\langle \mathrm{dom}\, F, emptycol, \\
&\qquad emptycol, empty\rangle \\
unit_{\mathsf{Seq}} \circ F &= seq\langle \mathrm{dom}\, F, \mathrm{dom}\, F, \\
&\qquad settail(\mathrm{dom}\, F, 1), F\rangle \\
unnest \circ seq\langle d_1, r_1, p_1, seq\langle d_2, r_2, p_2, F\rangle\rangle &= seq\langle d_1, r_1 * r_2, \\
&\quad newpos(r_1, r_2, p_1, p_2), F\rangle
\end{aligned}$$

The standard frame operations and the implementations of the monad operators are fairly standard. In the type signature, we have chosen r to be injective (tail-distinct), and therefore the item identifiers suffice as row identifiers for r. Thus we do not need *mkprod*() operations in the rules for D_{Seq} and *unnest*. However, in $r * seq\langle\rangle$ we need to create new inner keys to make sure they are unique (see the example later on). In $unit_{\mathsf{Seq}}$, which constructs singleton sequences, we simply use sequence identifiers as item identifiers.

107 The *newpos*() operator. The only new column operator needed is the *newpos* operation, which makes up new ordinals. This essentially entails clustering r_2 on α (through r_1) and then sorting r_2 first by the ordinals corresponding to β (through p_1) and then by those of γ (through p_2). Then, the new ordinals can be assigned using a simple counter which is reset at every group boundary.

At first sight, *newpos* sounds like a very expensive operation. However, in practice the tuples in the binary relations are stored in a certain order, and we can exploit that to make *newpos* cheaper. As is clearly visible in the loop lifting examples in paragraph 99 and further, r_1 and p_1 are already ordered according

to β, and the ordering of the ordinals agrees with that ordering. Likewise, r_2 is already ordered by β *and* γ, and p_2 is ordered by γ with the order of the ordinals again agreeing. Thus, in practice the group and order phases will be unnecessary, and *newpos* will amount to a simple grouped dense ranking operation, exactly corresponding with the one already used in MonetDB/XQuery, which has acceptable cost.

108 Example rewriting. Plugging the definition of $seq\langle\rangle$ into the rules for scope elimination in Section 2.5.3 give something remarkably close to the loop lifting demonstrated in Section 4.1. It is not hard to imagine how XQuery query (4.2) maps to the Dodo query

$$\lambda w \bullet Seq[x + y \mid x \leftarrow tens, y \leftarrow hundreds],$$

where *Seq* is the monad comprehension belonging to Seq. Then,

$$
\begin{aligned}
& [\![\lambda w \bullet Seq[x + y \mid x \leftarrow tens, y \leftarrow hundreds]]\!] \\
= & \quad \{ \text{ comprehension syntax } \} \\
& [\![\lambda w \bullet unnest\ (\mathsf{Seq}\ (\lambda x \bullet \mathsf{Seq}\ (\lambda y \bullet x + y)\ hundreds)\ tens)]\!] \\
= & \quad \{ \ w \text{ does not occur free in the expression } \} \\
& unnest \circ \mathsf{Seq}\ [\![\lambda x \bullet \mathsf{Seq}\ (\lambda y \bullet x + y)\ hundreds]\!] \circ [\![\lambda w \bullet tens]\!] \\
= & \quad \{ \text{ Scope unnesting (Equation 2.6), set } tens' = (\lambda_ \bullet tens) \ \} \\
& unnest \\
& \circ \mathsf{Seq}\ \big(\mathsf{Seq}\ [\![\lambda z \bullet (\lambda y \bullet exl\ z + y)\ (exr\ z)]\!] \circ D_{\mathsf{Seq}} \circ [\![\lambda x \bullet (x, hundreds)]\!]\big) \\
& \circ nest' \\
= & \quad \{ \ hundreds' = (\lambda_ \bullet hundreds), \text{ simplify } \} \\
& unnest \circ \mathsf{Seq}\ \big(\mathsf{Seq}\ (+) \circ D_{\mathsf{Seq}} \circ (id \triangledown hundreds')\big) \circ nest' \\
= & \quad \{ \text{ expand } \mathsf{Seq} \text{ functor } \} \\
& unnest \circ \mathsf{Seq}\ \mathsf{Seq}\ (+) \circ \mathsf{Seq}\ D_{\mathsf{Seq}} \circ \mathsf{Seq}\ (id \triangledown hundreds') \circ nest'.
\end{aligned}
$$

For comparison with the example in Section 4.1, we now substitute $seq\langle\rangle$ frames for $tens'$ and $hundreds'$, and evaluate the expressions

$$
\begin{aligned}
A_1 &= tens', \\
A_2 &= \mathsf{Seq}\ (id \triangledown hundreds') \circ A_1, \\
A_3 &= \mathsf{Seq} D_{\mathsf{Seq}} \circ A_2, \\
A_4 &= \mathsf{Seq}\ \mathsf{Seq}\ (+) \circ A_3, \\
A_5 &= unnest \circ A_4.
\end{aligned}
$$

We start with the frames *tens'* and *hundreds'*, where *hundreds'* is very similar to *tens'*, except that it uses internal key space β rather than α.

$$A_1 = tens' = seq\langle \begin{array}{c} d_1 \\ \hline \begin{array}{c|c} \dagger & \dagger \end{array} \end{array}, \begin{array}{c} r_1 \\ \hline \begin{array}{c|c} \dagger & \alpha_1 \\ \dagger & \alpha_2 \end{array} \end{array}, \begin{array}{c} p_1 \\ \hline \begin{array}{c|c} \alpha_1 & 1 \\ \alpha_2 & 2 \end{array} \end{array}, atom\langle \begin{array}{c} f_1 \\ \hline \begin{array}{c|c} \alpha_1 & 10 \\ \alpha_2 & 20 \end{array} \end{array} \rangle\rangle. \quad (4.9)$$

In the first step, every number in the sequence is transformed into a number/sequence pair. The *hundreds'* frame has domain $\mathbf{1} = \{\dagger\}$, so we use frame translation $c * F$ to translate it to the domain $\{\alpha_1, \alpha_2\}$.

$$\begin{array}{rcl} A_2 & = & seq\langle d_1, r_1, p_1, pair\langle atom\langle f_1\rangle, settail(f_1, \dagger) * hundreds'\rangle\rangle \\ & = & seq\langle d_1, r_1, p_1, pair\langle atom\langle f_1\rangle, \\ & & \begin{array}{c} \\ \hline \begin{array}{c|c} \alpha_1 & \dagger \\ \alpha_2 & \dagger \end{array} \end{array} * seq\langle \begin{array}{c} d_2 \\ \hline \begin{array}{c|c} \dagger & \dagger \end{array} \end{array}, \begin{array}{c} r_2 \\ \hline \begin{array}{c|c} \dagger & \beta_1 \\ \dagger & \beta_2 \end{array} \end{array}, \begin{array}{c} p_2 \\ \hline \begin{array}{c|c} \beta_1 & 1 \\ \beta_2 & 2 \end{array} \end{array}, atom\langle \begin{array}{c} f_2 \\ \hline \begin{array}{c|c} \beta_1 & 100 \\ \beta_2 & 200 \end{array} \end{array} \rangle\rangle\rangle\rangle \\ & = & seq\langle d_1, r_1, p_1, pair\langle atom\langle f_1\rangle, \\ & & seq\langle \begin{array}{c} d_2' \\ \hline \begin{array}{c|c} \alpha_1 & \alpha_1 \\ \alpha_2 & \alpha_2 \end{array} \end{array}, \begin{array}{c} r_2' \\ \hline \begin{array}{c|c} \alpha_1 & \gamma_1 \\ \alpha_2 & \gamma_2 \\ \alpha_2 & \gamma_3 \\ \alpha_2 & \gamma_4 \end{array} \end{array}, \begin{array}{c} p_2' \\ \hline \begin{array}{c|c} \gamma_1 & 1 \\ \gamma_2 & 2 \\ \gamma_3 & 1 \\ \gamma_4 & 2 \end{array} \end{array}, atom\langle \begin{array}{c} f_2' \\ \hline \begin{array}{c|c} \gamma_1 & 100 \\ \gamma_2 & 200 \\ \gamma_3 & 100 \\ \gamma_4 & 200 \end{array} \end{array} \rangle\rangle\rangle\rangle \end{array}$$

Then, the $seq\langle\rangle$ and the $pair\langle\rangle$ frame are exchanged using the D_{Seq} operator. It translates f_1 to the γ keyspace as $f_1' = r_2'^{\cup} * f_1$.

$$\begin{array}{rcl} A_3 & = & seq\langle d_1, r_1, p_1, seq\langle \begin{array}{c} d_2' \\ \hline \begin{array}{c|c} \alpha_1 & \alpha_1 \\ \alpha_2 & \alpha_2 \end{array} \end{array}, \begin{array}{c} r_2' \\ \hline \begin{array}{c|c} \alpha_1 & \gamma_1 \\ \alpha_2 & \gamma_2 \\ \alpha_2 & \gamma_3 \\ \alpha_2 & \gamma_4 \end{array} \end{array}, \begin{array}{c} p_2' \\ \hline \begin{array}{c|c} \gamma_1 & 1 \\ \gamma_2 & 2 \\ \gamma_3 & 1 \\ \gamma_4 & 2 \end{array} \end{array}, \\ & & pair\langle atom\langle \begin{array}{c} f_1' \\ \hline \begin{array}{c|c} \gamma_1 & 10 \\ \gamma_2 & 10 \\ \gamma_3 & 20 \\ \gamma_4 & 20 \end{array} \end{array} \rangle, atom\langle \begin{array}{c} f_2' \\ \hline \begin{array}{c|c} \gamma_1 & 100 \\ \gamma_2 & 200 \\ \gamma_3 & 100 \\ \gamma_4 & 200 \end{array} \end{array} \rangle\rangle\rangle\rangle \end{array}$$

Now, $\mathsf{Seq}\ \mathsf{Seq}$ $(+)$ uses a loop lifted addition operator to add the numbers.

$$A_4 = seq\langle d_1, r_1, p_1, seq\langle \begin{array}{c} d_2' \\ \hline \begin{array}{c|c} \alpha_1 & \alpha_1 \\ \alpha_2 & \alpha_2 \end{array} \end{array}, \begin{array}{c} r_2' \\ \hline \begin{array}{c|c} \alpha_1 & \gamma_1 \\ \alpha_2 & \gamma_2 \\ \alpha_2 & \gamma_3 \\ \alpha_2 & \gamma_4 \end{array} \end{array}, \begin{array}{c} p_2' \\ \hline \begin{array}{c|c} \gamma_1 & 1 \\ \gamma_2 & 2 \\ \gamma_3 & 1 \\ \gamma_4 & 2 \end{array} \end{array}, atom\langle \begin{array}{c} f_1' \oplus f_2' \\ \hline \begin{array}{c|c} \gamma_1 & 110 \\ \gamma_2 & 210 \\ \gamma_3 & 120 \\ \gamma_4 & 220 \end{array} \end{array} \rangle\rangle\rangle$$

And finally, *unnest* removes the nested $seq\langle\rangle$ frames and introduces a new numbering.

$$\begin{aligned} A_5 &= \mathit{seq}\langle d_1, \frac{r_1}{\begin{array}{c|l} \dagger & \alpha_1 \\ \dagger & \alpha_2 \end{array}} * \frac{r'_2}{\begin{array}{c|l} \alpha_1 & \gamma_1 \\ \alpha_2 & \gamma_2 \\ \alpha_2 & \gamma_3 \\ \alpha_2 & \gamma_4 \end{array}}, \frac{\mathit{newpos}}{\begin{array}{c|l} \gamma_1 & 1 \\ \gamma_2 & 2 \\ \gamma_3 & 1 \\ \gamma_4 & 2 \end{array}}, \mathit{atom}\langle \frac{f'_1 \oplus f'_2}{\begin{array}{c|l} \gamma_1 & 110 \\ \gamma_2 & 210 \\ \gamma_3 & 120 \\ \gamma_4 & 220 \end{array}} \rangle\rangle \\ &= \mathit{seq}\langle \frac{d_1}{\begin{array}{c|c} \dagger & \dagger \end{array}}, \frac{r_1 * r'_2}{\begin{array}{c|l} \dagger & \gamma_1 \\ \dagger & \gamma_2 \\ \dagger & \gamma_3 \\ \dagger & \gamma_4 \end{array}}, \frac{\mathit{newpos}}{\begin{array}{c|l} \gamma_1 & 1 \\ \gamma_2 & 2 \\ \gamma_3 & 3 \\ \gamma_4 & 4 \end{array}}, \mathit{atom}\langle \frac{f'_1 \oplus f'_2}{\begin{array}{c|l} \gamma_1 & 110 \\ \gamma_2 & 210 \\ \gamma_3 & 120 \\ \gamma_4 & 220 \end{array}} \rangle\rangle \end{aligned}$$

109 Recognizing loop lifting. If we now compare the expressions above to those in Section 4.1, they turn out to be very similar. For instance, the frame tens' corresponds to the table representation of $(\mathtt{10}, \mathtt{20})$ in equation (4.3), except for the use of more fancy key notation such as α_1 and †. In the next step, A_2, we introduce the frame $\mathit{hundreds}'$, but translate it to the domain $\{\alpha_1, \alpha_2\}$, which is exactly what we see happen in equation (4.5). Then, in A_3, f_1 is translated to the scope of f_2. The frame definition of $r * seq\langle\rangle$ states that this is done by translating with $r_2^{\cup}$. Thus, the frame structure of the $seq\langle\rangle$ frame helps keep track of the mapping between scopes, which we explicitly kept track of using the $(\mathit{iter}_2, \mathit{iter}_1)$ tables in the loop lifting example. In step A_4, the rule for the Seq functor makes it clear that $(+)$ should be applied to the $pair\langle\rangle$ frame inside the $seq\langle\rangle$ frames. Rewrite rules for $+$ on atom frames introduce the loop lifted addition operator $\oplus$. At this point, we have calculated the required numbers, but they are still inside a nested sequence. In equation (4.7), this is visible because they are in scope iter_2. The *unnest* operator joins r_1 and r_2 to bring them to the outer scope. It also introduces new position numbers using *newpos*.

In figure 4.3, we show A_5 both in frame-form and in a tabular representation similar to equation (4.7). It turns out that the keys γ remaining in the frame representation correspond to the implicit row-ids in the tabular form.

4.2.2 The *xmlnode*$\langle\rangle$ frame

110 Decomposition in Pathfinder. MonetDB/XQuery currently decomposes its XML document collection into 12 binary relations [BGvK+06]:

$$A_5 \quad = \quad seq\langle \begin{array}{c|c} \multicolumn{2}{c}{d_1} \\ \hline \dagger & \dagger \end{array}, \begin{array}{c|l} \multicolumn{2}{c}{r_1 * r'_2} \\ \hline \dagger & \gamma_1 \\ \dagger & \gamma_2 \\ \dagger & \gamma_3 \\ \dagger & \gamma_4 \end{array}, \begin{array}{c|l} \multicolumn{2}{c}{newpos} \\ \hline \gamma_1 & 1 \\ \gamma_2 & 2 \\ \gamma_3 & 3 \\ \gamma_4 & 4 \end{array}, atom\langle \begin{array}{c|l} \multicolumn{2}{c}{f'_1 \oplus f'_2} \\ \hline \gamma_1 & 110 \\ \gamma_2 & 210 \\ \gamma_3 & 120 \\ \gamma_4 & 220 \end{array} \rangle\rangle$$

		$x+$y	
(*rownum*)	$iter_0$	*pos*	*item*
(γ_1)	†	1	110
(γ_2)	†	2	210
(γ_3)	†	3	120
(γ_4)	†	4	120

Figure 4.3: Comparison between the final *seq*⟨⟩ frame of paragraph 108 and a tabular represention similar to (4.7).

positional info	node info	attribute info
pre_size	*pre_prop*	*attr_own*
pre_level	*pre_kind*	*attr_qn*
pre_frag	*qn_ns*	*attr_prop*
	qn_loc	
	prop_text	
	prop_com	
	prop_ins	
	prop_tgt	

The positional information pertains to the pre/post plane. Node information means information about the kind of node, e.g., element or comment, its name spaces, its name or text content, etc. Attributes are stored separately in the *attr_∗* tables.

The *pre_frag* relation is used to separate the node collection into *fragments*. Fragments are connected components of the XML node graph. There are two kinds of fragments: named fragments and anonymous fragments. Every document in the store is a distinct fragment, because nodes within a document are connected through axis steps, whereas nodes from different documents are not so connected. We call stored documents *named fragments* because they can be reached through their name: `doc("banana.xml")`. *Anonymous fragments* are fragments which do not have an external origin. They are created during query

evaluation as a result of element construction, e.g.,

```
<a>{ for $x in (10,20) return <b> {$x} </b> }</a>
```

111 The *xmlstore*$\langle\rangle$ **frame.** In Dodo, we use frames rather than a naming convention to organize the relations of the decomposed document collection. For convenience, we introduce *xmlstore*$\langle\rangle$, which simply groups together the twelve relations that make up the document store. It is not a proper frame because it does not provide rewrite rules for frame translation and other required operations.

For simplicity, we ignore name spaces and the special encoding of attributes; we simply regard attributes as nodes with a special name. In Pathfinder, the *_kind* and *_prop* columns are used to distinguish several types of nodes: text nodes, element nodes, comment nodes, etc. In Dodo, we use an *either*$\langle\rangle$ frame (sum type) to encode this distinction, but also ignore the matter occasionally for simplicity.

112 The *xmlnode*$\langle\rangle$ **frame.** Our *xmlnode*$\langle\rangle$ frame contains two *xmlstore*$\langle\rangle$ frames, one for the named and one for the anonymous fragments. The separation of named and anonymous fragments is important in the implementation of the frame union operator $\sqcup$.

This is the structure of the *xmlnode*$\langle\rangle$ frame:

$$xmlnode\langle pre^a, pre^n, store^a, store^n, meta\rangle.$$

Here, $store^a$ and $store^n$ are *xmlstore*$\langle\rangle$ structures containing the anonymous and the named document stores. Although semantically, map keys to individual nodes, the *xmlnode*$\langle\rangle$ frame carries around the whole document store within its frame structure. The reason is that in paragraph 104, we stated that

> *The NODE type is an abstract type encoding all information about a single XML tree node, including its relationship with other nodes.*

The relationship of a node with other nodes is encoded as pre/post information in the *xmlstore*$\langle\rangle$ structures.

The pre^a and pre^n columns provide the mapping from external node identifiers to pre-numbers. We will generally assume that the pre-numbers in $store^a$ are different from those in $store^n$. As the query navigates the documents, pre^a and pre^n vary; the *pre* relations point out the "active" nodes in the store, while $store^a$ and $store^n$ contain all nodes and remain unchanged during navigation. The *meta* component is used to record meta-information such as the names of the fragments in $store^n$. It is used, among others, by the `doc()` function but will be ignored from now on.

It is important to understand the distinction between $xmlnode\langle\rangle$ and $xmlstore\langle\rangle$. The $xmlstore\langle\rangle$ structure contains information about a complete collection of XML trees: it contains positional and node information about *all* nodes in *all* trees. On the other hand, an $xmlnode\langle\rangle$ frame represents a selection of nodes out of an $xmlstore\langle\rangle$. For instance, consider a frame of type $\alpha \to \mathsf{Seq}(SS + NODE)$ happening to contain only strings and no nodes. Because it contains no nodes, the domain of the $xmlnode\langle\rangle$ frame inside the $seq\langle\rangle$ is empty: there are no nodes in the sequences, only strings. However, the $xmlstore\langle\rangle$ frame $store^n$ still contains the whole tree collection, because in a later step, the query might invoke the `doc()` operator to retrieve a document from it.

The frame structure of the type $\alpha \to \mathsf{Seq}(SS + NODE)$ resulting from XQuery expressions was already sketched at the end of paragraph 105, but can now be given in more detail:

$$seq\langle \ldots, either\langle atom\langle \ldots\rangle, xmlnode\langle \ldots,\ xmlstore\langle \ldots\rangle, xmlstore\langle \ldots\rangle\rangle\rangle\rangle,$$

The $seq\langle\rangle$ frame contains item keys, which the $either\langle\rangle$ frame maps tot the $atom\langle\rangle$ frame if they are strings, and to the $xmlnode\langle\rangle$ frame if they are nodes. The *pre* columns in the $xmlnode\langle\rangle$ frame are used to map the item keys to a pre-number, which can be used to look up information about the node in the $xmlstore\langle\rangle$ structure.

113 Operations on $xmlnode\langle\rangle$**.** Here we sketch the implementation of the required frame operations for the $xmlnode\langle\rangle$ frame:

$$\begin{aligned}
\text{dom}\ xmlnode\langle pre^a, pre^n, \ldots\rangle &= \ !twin(pre^a \cup pre^n),\\
r * xmlnode\langle pre^a, pre^n, \ldots\rangle &= xmlnode\langle r * pre^a, r * pre^n, \ldots\rangle,\\
empty &= xmlnode\langle \emptyset, \emptyset, empty, store^n, meta\rangle,\\
xmlnode\langle p_1^a, p_1^n, st_1^a, st^n, meta\rangle \sqcup &\\
xmlnode\langle p_2^a, p_2^n, st_2^a, st^n, meta\rangle &= xmlnode\langle p_3^a, p_1^n \cup p_2^n, st_3^a, st^n, meta\rangle,
\end{aligned}$$

where in the last rule we abbreviated *pre* to p and *store* to st, and we take pre_3^a and $store_3^a$ as described below.

In the first two rules, we see how dom and $r*$ only influence *pre* and do not reach the $xmlstore\langle\rangle$ structures, hence no expensive renumbering of the whole collection. In the union rule, we still need to describe pre_3^a and $store_3^a$. We do not give formulas because in this case, they obscure more than they clarify.

A typical use of the $\sqcup$ operator is to implement **if then else**. After evaluation of the **then** and the **else** branch, the results are merged back into a single frame using $\sqcup$. We know that that $store^n$ is identical in both $xmlnode\langle\rangle$ frames, because it is not updated during query evaluation. If neither of the branches

has created new nodes, $store^a$ will also be identical. In case node creation does occur, the $store^a$ structures will be different and need to be merged. When the pre numbers used in $store^a_1$ and $store^a_2$ overlap, this may involve renumbering.

114 Implementing doc(). The `doc()` function maps document names to the root node of the corresponding fragment. It takes one argument, which is expected to be singleton sequence containing a string. We rewrite

$$doc \circ seq\langle \ldots, either\langle atom\langle dname\rangle, xmlnode\langle \ldots store^n, meta \ldots\rangle\rangle\rangle,$$

where $meta$ contains the column $frag_name$, into

$$seq\langle \ldots, either\langle empty, xmlnode\langle \emptyset, dname * frag_name, \ldots, store^n, meta, \ldots\rangle\rangle\rangle.$$

In other words, we drop the names and create a new pre^n column out of $dname$ and $frag_name$. It is not necessary to actually copy the documents out of $store^n$, we just insert references to them into the pre^n column. Any anonymous fragments mentioned in the argument sequence are now unreachable, so $store^a$ is dropped.

115 Implementing axis steps. XPath axis steps navigate from nodes to other nodes. Thus, they affect only pre^n and pre^a. Because XPath expressions never leave the fragment, the path expression can be evaluated in parallel for pre^a against $store^a$ and pre^n against $store^n$. Thus, in the rest of this paragraph we simply refer to pre and $store$.

An axis step maps a node to a node set. After evaluating the first step in an expression consisting of several steps, the rest of the path is evaluated for every node in the result, and the result of the path expression is the union of the resulting node sets. In an XQuery context, path expressions yield *node sequences*, consisting of the items in the node set, ordered in document order. The final step in a path expression may produce non-node results, e.g., `fn:string(n)`, in that case, the items are ordered according to the order of the corresponding node.

We implement path expressions as a chain of item sequence transformations.

$$e/step_1::test_1/\ldots/step_n::test_n$$

simply as a chain of operations:

$$FILTER_{\mathsf{Seq}}\ (test'_n) \circ step_n \circ \cdots \circ$$
$$FILTER_{\mathsf{Seq}}\ (test'_1) \circ step_1 : \mathsf{Seq}(SS + NODE) \rightarrow \mathsf{Seq}(SS + NODE)$$

applied to expression e. In this expression,

$$FILTER_{\mathsf{Seq}}\ p = unnest_{\mathsf{Seq}} \circ \mathsf{Seq}\ (\lambda n \bullet \mathbf{if}\ (p\ n)\ \mathbf{then}\ unit_{\mathsf{Seq}}\ n\ \mathbf{else}\ zero_{\mathsf{Seq}})$$

is a function which eliminates items which do not match predicate p from a sequence, and the $test'_i$ are Dodo translations of the XPath tests expressions $test_i$.

The steps are implemented using loop lifted stair case joins, using the *pre* column as the context node set and *store* as the document table. The *iter* argument to the loop lifted staircase join is provided using the sequence to item relation r of the $seq\langle\rangle$ frame. Notice that the arguments to the filters are lambda terms which can be loop lifted according to the normal rewrite rules.

4.3 Conclusion

In this chapter, we have looked at XQuery/MonetDB (Pathfinder) as an illustration of query flattening, and as a specialization of the Dodo approach.

Pathfinder uses a well-chosen mapping from nested (XML) data model to flat relational model, and compiles XQuery queries into equivalent relational algebra queries at the flattened level. In doing so, it benefits greatly from loop lifting. Loop lifting is a translation technique replacing nested-loop query plans by table manipulations. Compiled expressions work on a iteration context table which contains all variable bindings the expression is evaluate with. The result is a table containing all answers, which can be fed into subsequent expressions.

Experimental results with Pathfinder demonstrate that many-at-a-time processing such as introduced by loop lifting can improve efficiency by orders of magnitude. One explanation for this can be found in the properties of modern computer architectures. Physical operators which process multitudes of operands in a single operation often display significantly better cache behaviour and suffer less branch misprediction penalties than nested-loop iteration. The other is that many-at-a-time operations have the opportunity to exploit relations between its arguments. For instance, the staircase join saves considerable work by pruning the context node sequence (figure 4.2).

These concerns are precisely the driving motivation behind Dodo. Regarding Question 1 of paragraph 92, we observe very similar query rewriting strategies in Pathfinder and Dodo, but where Pathfinder focusses on two fundamental data structures, item sequence and pre/post representation, Dodo is concerned with flattening complex data structures *in general*.

In this chapter we have shown how Pathfinder's fundamental data structures can be implemented as Dodo extensions (Question 2), and how, given an implementation for the sequence type, the loop lifting in Pathfinder arises automatically as a consequence of Dodo's general rules for dealing with nested scopes. This in itself may be of interest to seasoned Pathfinder developers, but it also serves as a validation of the Dodo approach. The success of loop lifting in Pathfinder demonstrates that Dodo's scope elimination rules indeed give rise

to well performing systems.

With regard to the staircase join, Dodo does not help developers invent the staircase join. Inventing a smart mapping of complex data to flattened data requires a degree of creativity one cannot ask of computer program. On the other hand, given the idea of the pre/post plane and the staircase join algorithms, Dodo does help design a system around them (Question 3). The multi-model approach generally encourages developers to design data representations which are more creative than simply mapping object fields to table attributes, and gives them the tools to implement every part at the right level of abstraction.

Chapter 5

Categorical background

In this chapter, we take a closer look at the category theoretical background of Dodo, in particular at monads, which are the theoretical foundation of the comprehension syntax, and at catamorphisms, which are used to model aggregate functions and conversions, and in Chapter 6 also to implement a restricted class of recursive functions. The theory given here has been foreshadowed already in a some cryptic remarks near paragraph 30 of Chapter 2. In this chapter, we give a more formal treatment.

In Section 5.1 we show how algebras and catamorphisms are a powerful and concise method of reasoning about data structures and programs which traverse those data structures. In Section 5.2, we examine the monad concept, and its use in monad comprehensions. These sections are written purely from the point of view of nested data types, and do not consider flattening. In the third section, and in Chapter 6, we consider how to apply the theory to the situation in Dodo, where the nested data model is layered over a flattened representation.

5.1 Algebras, monads and comprehensions

116 Abstract data types. At the conceptual level, lists of integers, with type denoted L, can be thought of to be generated using two operations

1. $nil_L : \mathbf{1} \to L$, which returns the empty list;
2. $cons_L : \mathbb{Z} \times L \to L$, which 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 $\oplus$. For instance, to calculate the sum of the numbers in the list, one could take $e = 0$ and $\oplus = +$:

$$\begin{aligned} f\ nil(\dagger) &= sum\ [\,] = e = 0, \\ f\ cons(x,\ell) &= x + (f\ \ell). \end{aligned}$$

It can be shown with induction to the argument of f that f calculates the sum the numbers of the list. We say that $f = sum$ is defined *by induction to the structure of its argument.*

As another example of a data structure, bags of integers, denoted B, are defined using the same two operations *nil* and *cons*, but with in addition the equation

$$cons(x, cons(y,\ell)) = cons(y, cons(x,\ell)), \tag{5.1}$$

which expresses the indifference of bags towards the order in which elements are inserted: in the LHS, y is added first, then x, but in the RHS it is the other way around.

In contrast to lists, where there was a one-to-one relation between the elements of L and the *nil*/*cons*-trees, with bags, there is only a one-to-one relation between elements of B and *equivalence classes* induced on B by equation (5.1).

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. Notice that in a sense, adding equations makes the type "smaller", because the equations make formerly different elements indistinguishable from each other.

117 Combining operators into one. In the previous paragraph, we wrote that an algebra is a base type with a collection of operators defined on it. Using sum types, it is possible to wrap up all operators into a single super-operator. Defining an algebra as a base type abstracts further from day to day use, but is more convenient from a theoretical point of view. As an example of a super-operator, it is possible to combine *nil* and *cons* into a single operator

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

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 usefulness of this will soon become apparent.

118 Definition (Algebra). *Given a functor* F*, an* F-algebra *is a function*

$$\tau : \mathsf{F}A \to A.$$

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* $\mathcal{A}lg(\mathsf{F}, E)$*.*

In this definition, F represents the "signature" of the algebra. We give two examples:

- The list algebra has two operators: a constant function *nil*, and *cons* which takes 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

$$\mathsf{INS}\ X \;=\; 1 + \mathbb{Z} \times X,$$
$$\mathsf{INS}\ f = id_1 + id_{\mathbb{Z}} \times f.$$

- 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 $(\lambda z \bullet 0)$.

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 to use $nil : 1 \to L$, $tip : \mathbb{Z} \to L$, $concat : L \times L \to L$ and an equation expressing associativity of *concat*. This is called a *union algebra* because now the fundamental operation is concatenation (union). Union algebras have signature functor $\mathsf{UN}\ X = 1 + \mathbb{Z} + X \times X$.

119 Algebras with equations. Informally, an algebra $a \triangledown op$ satisfies, say, equation (5.1) if substituting $nil = a$ and $cons = op$ yields truth for any instantiation of x, y and ℓ. 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 may skip the rest of this paragraph. For a full treatment of algebras with equations, see Fokkinga [Fok92].

We write equation (5.1) 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

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

In these intimidating looking definitions, the *inr* serves to select the proper sub-operation of φ, i.e., *cons* if $\varphi = nil \triangledown cons$. 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.

120 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 $\alpha \to_{\mathsf{F}} \gamma$.

121 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:

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

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 $(nil_L \triangledown cons_L) \to_{\mathsf{INS}} (\underline{0} \triangledown (+))$ with INS as in paragraph 118. The homomorphism equation for *sum*

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

can be written as two equations

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

The following diagrams illustrate its behaviour:

$$
\begin{array}{ccc}
\mathit{inl}\ \dagger & \xmapsto{\mathit{nil} \triangledown \mathit{cons}} & [\,] \\
\downarrow{\scriptstyle id_1 + id_{\mathbb{Z}} \times sum} & & \downarrow{\scriptstyle sum} \\
\mathit{inl}\ \dagger & \xmapsto[\underline{0} \triangledown (+)]{} & 0
\end{array}
\qquad
\begin{array}{ccc}
\mathit{inr}(2,[3,5]) & \xmapsto{\mathit{nil} \triangledown \mathit{cons}} & [2,3,5] \\
\downarrow{\scriptstyle id_1 + id_{\mathbb{Z}} \times sum} & & \downarrow{\scriptstyle sum} \\
\mathit{inr}(2,8) & \xmapsto[\underline{0} \triangledown (+)]{} & 2+8
\end{array}
\tag{5.2}
$$

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

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

Lemma 124 provides a perhaps more intuitive interpretation: $\tau : \mathsf{F}A \rightarrow A$ is initial if and only if it is a bijection, up to E-equivalence, between $\mathsf{F}A$ and A. We shall see in paragraph 126 that this makes catamorphisms functions which are expressed by induction to the structure of their arguments.

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

124 Lambeks lemma. An algebra $\tau : \mathsf{F}A \rightarrow A$ 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 116 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 $[\mathit{cons}(x, \mathit{cons}(y, \ell)) = \mathit{cons}(y, \mathit{cons}(x, \ell))]$-equivalence class. In contrast, the algebra $(\underline{0} \triangledown (+)) : \mathsf{F}\mathbb{Z} \rightarrow \mathbb{Z}$ is not initial in $\mathcal{A}lg(\mathsf{F}, \text{eqn. (5.1)})$: the number 5 can be written in different ways that are not (5.1)-equivalent:

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

125 Obtaining initiality. When a functor F is polynomial, i.e., when it can be expressed completely as a composition of sum type, product type, identity functor, constant functor, and type 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 (5.1).

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 [Fok92], paragraph 39 on page 60. We are not aware of a proof for the general case. We use this in paragraph 138.

126 Initiality in operation. The pictures in paragraph 121 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.

127 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 indispensable 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 $(\!|\varphi|\!)_{\mathsf{F},E}$ 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 135.

128 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 2.2.1 we introduced the List functor which could construct list types over arbitrary types, not just over integers. To extend the algebra concept to parameterized 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 $\mathsf{INS}(A, X)$.

5.2 Monads and monad comprehensions

129 Comprehension examples. Comprehension notation provides a convenient way to write down collection values and aggregations. A comprehension expression $M[\ \mid \ldots]$ 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 $[\ \mid\ ]$ has no meaning in itself. Examples of comprehensions with $xs = [1, 2, 3]$ are

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

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

$$\begin{array}{rl}
 & \mathit{List}[\mathit{List}[x + y \mid y \leftarrow xs] \mid x \leftarrow xs] \\
= & [\mathit{List}[1 + y \mid y \leftarrow xs], \mathit{List}[2 + y \mid y \leftarrow xs], \mathit{List}[3 + y \mid y \leftarrow xs]] \\
= & [[2, 3, 4],\ [3, 4, 5],\ [4, 5, 6]]
\end{array} \tag{5.3}$$

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.

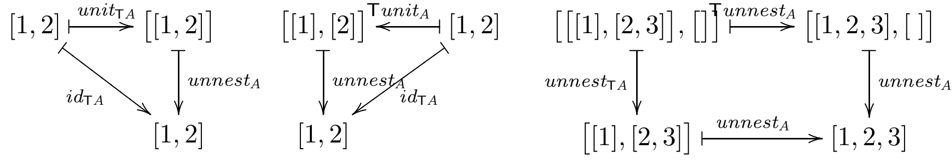
130 Definition (Monad). A T-monad is a triple $(\mathsf{T}, \mathit{unit}, \mathit{unnest})$ with T a functor, *unit* and *unnest* families of functions $\mathit{unit}_A : A \to \mathsf{T}A$ and $\mathit{unnest}_A : \mathsf{TT}A \to \mathsf{T}A$ that satisfy the equations in the following commutative diagrams:

$$\begin{array}{ccccc}
\mathsf{T}A & \xrightarrow{\mathit{unit}_{\mathsf{T}A}} & \mathsf{TT}A & \xleftarrow{\mathsf{T}\mathit{unit}_A} & \mathsf{T}A \\
 & \searrow^{\mathit{id}_{\mathsf{T}A}} & \downarrow{\scriptstyle \mathit{unnest}_A} & \swarrow_{\mathit{id}_{\mathsf{T}A}} & \\
 & & \mathsf{T}A & &
\end{array}
\qquad
\begin{array}{ccc}
\mathsf{TTT}A & \xrightarrow{\mathsf{T}\mathit{unnest}_A} & \mathsf{TT}A \\
\downarrow{\scriptstyle \mathit{unnest}_{\mathsf{T}A}} & & \downarrow{\scriptstyle \mathit{unnest}_A} \\
\mathsf{TT}A & \xrightarrow{\mathit{unnest}_A} & \mathsf{T}A
\end{array} \tag{5.4}$$

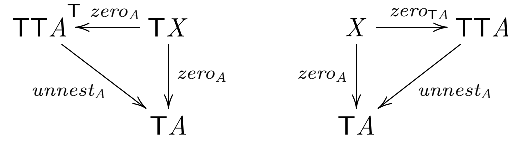
The intuition here is that unit_A adds an extra level of nesting while *unnest* removes one. As an example, consider

$$\begin{array}{rcl}
\mathsf{T} & = & \mathsf{List}, \\
\mathit{unit}_A\ a & = & [a], \\
\mathit{unnest}_A\ [[a_1, \ldots, a_n], \ldots, [z_1, \ldots, z_m]] & = & [a_1, \ldots, a_n, \ldots, z_1, \ldots, z_m]
\end{array}$$

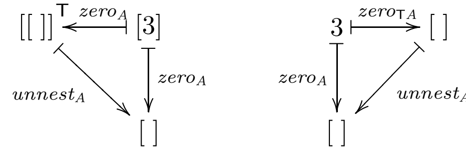
It is interesting to try out the equalities from (5.4) on this monad:



131 Definition (Monad with Zero). A monad with zero (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



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



132 Comprehension syntax. The Dodo comprehension syntax demonstrated in paragraph 129 consists of three parts. First, the name of the comprehension type, e.g, List. Second, the head of the comprehension, e.g., 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 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 translation to comprehensionless syntax according to the following rules:

- A comprehension $M[e\ |\ ]$ is rewritten to $(unit_M\ e)$. So, $List[3\ |\ ] = unit_{List}\ 3 = [3]$. Likewise, referring to paragraph 133, $Sum[3\ |\ ] = unit_{Sum}\ 3 = id\ 3 = 3$.

- 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,

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

- As hinted at in paragraph 129, 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 (5.3),

$$\begin{aligned} & List[x+y \mid x \leftarrow xs,\ y \leftarrow xs] \\ = \ & unnest_{\mathsf{List}}\ List[List[x+y \mid y \leftarrow xs] \mid x \leftarrow xs] \\ = \ & unnest_{\mathsf{List}}\ [[2,3,4],\ [3,4,5],\ [4,5,6]] \\ = \ & [2,3,4,\ 3,4,5,\ 4,5,6] \end{aligned} \tag{5.5}$$

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

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

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

133 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 $\mathsf{INS}\ \mathsf{Id}A \rightarrow \mathsf{Id}A$. In such a case we define the *Sum* comprehension type as

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

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

5.3 Application in Dodo

134 Mapping to storage layout. In the nested data model, data types are defined in terms of constructor algebras like $\tau = \mathit{nil} \triangledown \mathit{cons}$. 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.

135 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, 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 127 we mentioned theorems that can be used to combine adjacent catamorphisms and homomorphisms, eliminating the materialization 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 137 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.

136 Homomorphism graph. In order to express catamorphisms ($\llparenthesis\cdot\rrparenthesis$, see 122) 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 knows 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 Dodo's storing algebras simply as a name, and sophisticated Dodo's 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 $\llparenthesis\tau \circ \mathsf{F}(f, id)\rrparenthesis$. Using fusion theorems (paragraph 127) 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 $\varphi \circ \mathsf{F}(f, id)$ will make the homomorphism graph much more useful.

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

$$
\begin{array}{ccccc}
\rho & \xleftarrow{\;rev\;} & \tau & \xrightarrow{\;\mathsf{T}\,f\;} & \tau \circ \mathsf{F}(f, id) \\
 & \searrow^{lb}\,(\text{dashed}) & \downarrow{\scriptstyle lb} & & \downarrow{\scriptstyle lb}\,(\text{dashed}) \\
 & & \beta & \xrightarrow{\;\mathsf{B}\,f\;} & \beta \circ \mathsf{F}(f, id) \\
 & & \downarrow{\scriptstyle sum} & & \\
 & & \sigma & &
\end{array}
\qquad (5.6)
$$

with τ declared the initial F-algebra and β the initial (F, E)-algebra. Recall that $E = \{(5.1)\}$ 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, $sum = \llparenthesis\sigma\rrparenthesis_{\mathsf{F},E}$. Likewise, we have the list reversal function $rev = \llparenthesis\rho\rrparenthesis_{\mathsf{F}} : \mathsf{T}A \rightarrow \mathsf{T}A$ and the conversion function $lb = \llparenthesis\beta\rrparenthesis_{\mathsf{F}} : \mathsf{T}A \rightarrow \mathsf{B}A$. 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 19) 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.

138 No Bag type needed. In example 137, 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 137 we picked the algebra $\beta : \mathsf{INS}\mathsf{B}A \rightarrow \mathsf{B}A$ as a convenient initial object of $\mathcal{A}lg(\mathsf{INS}, E)$, where $E = \{(5.1)\}$ represents indifference to order. If there is no bag type available, we can just use another initial (INS, E)-algebra.

Paragraph 125 promises the existence of an algebra τ' that constructs lists just as τ does, but with the added assumption that the order of the elements in the list shall never be considered. The catamorphism $(\!|\tau \rightarrow \tau'|\!)$ can be implemented as id because the underlying implementation remains the same.

The updated homomorphism graph becomes

$$
\begin{array}{ccccc}
\rho & \xleftarrow{rev} & \tau & \xrightarrow{\mathsf{T}\, f} & \tau \circ \mathsf{F}(f, id) \\
 & \overset{id}{\searrow} & \big\downarrow id & & \big\downarrow id \\
 & & \tau' & \xrightarrow{\mathsf{B}\, f} & \tau' \circ \mathsf{F}(f, id) \\
 & & \big\downarrow sum & & \\
 & & \sigma & &
\end{array}
\qquad (5.7)
$$

139 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.

Chapter 6

Inductively defined types

In Chapter 2 we described the Dodo data model with its nested and flattened layer. New types at the nested layer are defined by defining a frame representation. The framework described in Chapter 2 allows for types with parameters, such as `List`, but not for inductive types. An inductively defined type is a type which is defined in terms of itself, such as the classical list type in functional programming:

$$\texttt{data ZList = Nil | Cons } \mathbb{Z} \texttt{ ZList} \tag{6.1}$$

Notice how `ZList` occurs on the right hand side, while being defined on the left hand side. In paragraph 126 of Chapter 5, catamorphisms were introduced as *functions which are expressed by induction to the construction of their argument.* Catamorphisms play an important role when dealing with comprehension syntax (figure 2.3) and when converting and aggregating over collection types. However, in Chapter 2, inductively defined types were not supported as such. Instead, in paragraph 27 we recommended to define such types opaquely and endow them with higher-order functions suitable for processing and iterating over the type. For instance, in Chapter 3 we defined a `List` type with the functor-property (`map`) and `sum` and `count` functions. To implement a given catamorphism $(\!|\alpha|\!)$, Dodo attempts to find a path in the catamorphism graph which fully consists of "canned" catamorphisms such as *list*2*bag* and *sum* (paragraph 137).

The recommendation still stands. For maximum performance, processing should be done using operations defined at the frame level, translated to frame transformations and efficient flat operators. In the current chapter, however, we consider how to extend the Dodo data model with inductively defined types and how to deal with functions which recursively traverse values of such types. Apart from this being an interesting topic by itself, it is also a useful foundation for the theory described in paragraph 137, since it allows the system to evaluate

catamorphisms by itself if no suitable predefined implementation can be found. In such cases, implementing the catamorphism directly may be useful as a last resort.

140 This chapter. In Section 6.2 we define a frame representation for inductive types. We

- specify the structure of the $mu\langle\cdots\rangle$ frame and its semantics (its interpretation function);
- give rewrite rules for the standard frame operations $dom\ mu\langle\cdots\rangle$, $r * mu\langle\cdots\rangle$ and $mu\langle\cdots\rangle \sqcup mu\langle\cdots\rangle$;

This allows us to build inductively defined data types by placing suitably formatted data in the underlying database and declaring a $mu\langle\cdots\rangle$ frame. It also allows Dodo to handle such data when it occurs in queries. In Section 6.3 and 6.4, we consider how to construct such values within queries (initial algebras) and how to consume values (catamorphims), respectively. We

- show how to generate a rewrite rule which implements the initial algebra for an inductively defined type, based on its structure (signature);
- show how to implement catamorphisms using a generic bulk oriented iteration at the column level.

6.1 Inductively defined types and catamorphisms

141 Inductively defined types. Above we presented the type of lists of integers as it might be represented in the functional language Haskell: `data ZList = Nil | Cons ℤ ZList`. This equation states that a value of type `ZList` is either the empty list (`Nil`) or an element ($\mathbb{Z}$) prepended (`Cons`) to another list (`ZList`). In the notation of Chapter 2, we could say

$$ZList \text{ “=” } 1 + \mathbb{Z} \times ZList. \tag{6.2}$$

More precisely, there is a bijection τ with inverse $\tau^{\cup}$ between $ZList$ and $1 + \mathbb{Z} \times ZList$, the partial decomposition of its tree structure. In fact, this bijection is precisely the initial algebra (paragraph 122) of $ZList$.

$$1 + \mathbb{Z} \times ZList \underset{\tau^{\cup}}{\overset{\tau}{\rightleftarrows}} ZList. \tag{6.3}$$

Generalizing to arbitrary *signature* functor S, with in the above case $\mathsf{S}X = 1 + \mathbb{Z} \times X$, Equations (6.2) and (6.3) take the form

$$ZList \text{ ``}{=}\text{''} \ \mathsf{S}ZList; \qquad \mathsf{S}ZList \underset{\tau^{\cup}}{\overset{\tau}{\rightleftarrows}} ZList.$$

The equations then read "*ZList* is a fixpoint type of S," for which, as all such types are isomorphic, we introduce the notation $\mu\mathsf{S}$. So, $ZList = \mu\mathsf{S}$. In the examples in this chapter, we usually take $\mathsf{S}X = 1 + \mathbb{Z} \times X$, but the theory is independent of the choice of S.

142 Catamorphisms. We have seen in Chapter 5 that initiality of $\tau : \mathsf{S}T \to T$ means that for every S-algebra $\alpha : \mathsf{S}A \to A$, there is a *unique* homomorphism $(\!|\alpha|\!) : T \to A$ which makes the following diagram commute:

$$\begin{array}{ccc} \mathsf{S}T & \xrightarrow{\tau} & T \\ {\scriptstyle \mathsf{S}(\!|\alpha|\!)}\downarrow & & \downarrow{\scriptstyle (\!|\alpha|\!)} \\ \mathsf{S}A & \xrightarrow{\alpha} & A \end{array} \tag{6.4}$$

For instance, if we put $\tau = nil \triangledown cons$ and choose $\alpha = \underline{0} \triangledown (+)$, it is easy to derive

$$\begin{aligned} (\!|\alpha|\!)\ [] &= (\alpha \circ \mathsf{S}(\!|\alpha|\!) \circ \tau^{\cup})\ [\,] = (\alpha \circ \mathsf{S}(\!|\alpha|\!))\ (inl\ \dagger) = \alpha\ (inl\ \dagger) = \underline{0}\ \dagger = 0, \\ (\!|\alpha|\!)\ [z, zs \ldots] &= (\alpha \circ \mathsf{S}(\!|\alpha|\!) \circ \tau^{\cup})\ [z, zs \ldots] = \cdots = \alpha\ (inr\ (z, (\!|\alpha|\!) zs)) = z + (\!|\alpha|\!) zs \end{aligned}$$

which is the definition of the well-known function *sum* on lists:

$$\begin{aligned} sum\ [\,] &= 0, \\ sum\ [z, zs \ldots] &= z + sum\ zs. \end{aligned}$$

Thus the statement that catamorphisms are functions defined by induction to the structure of their argument.

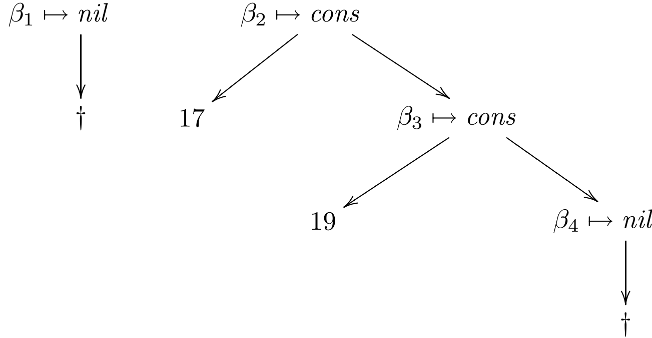
6.2 Frame representation for fixpoint types

143 Example. In this section we define a frame structure for inductively defined types. Inductively defined types can be characterized as types of which the values may "contain" other values of the same type. For instance, if we write the list $[17, 19]$ in *cons*/*nil*-notation, we see that $[17, 19] = cons(17, [19])$. In our frame representation, we break up this containment relation and instead "prepend" 17 not to the list [19] itself, but to a key identifying the list [19]. Before we state this formally, we begin with an example.

144 Example. In this example we consider a frame representation for the mapping

$$F = \{\alpha_1 \mapsto [\,], \alpha_2 \mapsto [17, 19]\} : \alpha \to \mathit{ZList}.$$

A natural starting point for a frame representation of F is to first consider all "list nodes" used in F, and identify each of them with a key:



Notice that we use keys α_1 and α_2 to identify the lists as seen from outside the frame, and that we use keys β_1, ..., β_4 for the "internal view." In the frame representation, there is a column $r : [\alpha \mapsto \beta]$ to make the relation between the two explicit. This relation identifies which list node β_i is the root of the tree identified by α_j. Now, we can remove the nested structure by replacing all references to sublists by explicit keys:

$$G = \left\{ \begin{array}{c} \beta_1 \mapsto nil \\ \downarrow \\ \dagger \end{array}, \quad \begin{array}{c} \beta_2 \mapsto cons \\ \swarrow \quad \searrow \\ 17 \qquad \beta_3 \end{array}, \quad \begin{array}{c} \beta_3 \mapsto cons \\ \swarrow \quad \searrow \\ 19 \qquad \beta_4 \end{array}, \quad \begin{array}{c} \beta_4 \mapsto nil \\ \downarrow \\ \dagger \end{array} \right\}.$$

In this form, the data can easily be encoded in a regular frame $G : \beta \to 1 + \mathbb{Z} \times \beta$. Together, r and G contain all information we need to represent F. All we have to do is wrap $G : \beta \to \mathsf{S}\beta$ and $r : [\alpha \mapsto \beta]$ into a frame for the fixpoint type, which we call $mu\langle\rangle$. In practice, we also add the domain $d = \operatorname{dom} G$ to the

frame. This is redundant, but simplifies the rewrite rules. The result:

$$
\begin{aligned}
F &= mu\langle \begin{array}{c|c} \multicolumn{2}{c}{r} \\ \hline \alpha_1 & \beta_1 \\ \alpha_2 & \beta_2 \end{array}, \begin{array}{c|c} \multicolumn{2}{c}{d} \\ \hline \beta_1 & \beta_1 \\ \beta_2 & \beta_2 \\ \beta_3 & \beta_3 \\ \beta_4 & \beta_4 \end{array}, G\rangle \\
G &= either\langle atom\langle \begin{array}{c|c} \multicolumn{2}{c}{e} \\ \hline \beta_1 & \beta_1 \\ \beta_4 & \beta_4 \end{array}\rangle, pair\langle atom\langle \begin{array}{c|c} \multicolumn{2}{c}{f} \\ \hline \beta_2 & 17 \\ \beta_3 & 19 \end{array}\rangle, atom\langle \begin{array}{c|c} \multicolumn{2}{c}{s} \\ \hline \beta_2 & \beta_3 \\ \beta_3 & \beta_4 \end{array}\rangle\rangle\rangle.
\end{aligned}
\tag{6.5}
$$

The column e lists the *nil*-nodes, f encodes the data in the *cons*-nodes, and s lists the successor nodes of the *cons*-nodes.

Notice that in this example, the encoding is redundant: keys β_1 and β_4 both identify identical *nil*-nodes. It is possible to remove this redundancy by replacing every β_1 by β_4, or vice versa, without affecting the theory in this chapter.

145 Semantics of the mu-frame. The relationship between $mu\langle r, d, G\rangle$ and r, d and G is defined succinctly by the following diagram:

$$
\begin{array}{ccccc}
\mathsf{S}\beta & \xleftarrow{\ G\ } & \beta & \xleftarrow{\ r\ } & \alpha \\
\downarrow{\scriptstyle \mathsf{S}mu\langle d,d,G\rangle} & & \downarrow{\scriptstyle mu\langle d,d,G\rangle} & \swarrow{\scriptstyle mu\langle r,d,G\rangle} & \\
\mathsf{S}(\mu\mathsf{S}) & \underset{\tau^{\cup}}{\overset{\tau}{\rightleftarrows}} & \mu\mathsf{S} & &
\end{array}
\tag{6.6}
$$

When speaking of the semantics of the $mu\langle\rangle$-frame, the question is: given a frame $mu\langle r, d, G\rangle$, such as the one in (6.5), how does one look up a key in it? In the diagram, we start with a key in α, e.g., in the upper right of the diagram, and we wish to obtain the corresponding value of type $\mu\mathsf{S}$ in the lower right. The first step is to use r to turn the zlist key α into a internal node key β. The question of the semantics of $mu\langle r, d, G\rangle$ is now reduced to the semantics of $mu\langle d, d, G\rangle$, which is a $mu\langle\rangle$-frame which does not distinguish between internal and external keys: every node is visible from the outside. In the sequel, it will often be convenient to focus on how to handle frames $mu\langle d, d, G\rangle$. Any such discussion can easily be generalized to $\langle r, d, G\rangle$ using the law

$$
mu\langle r, d, G\rangle = r * mu\langle d, d, G\rangle \tag{6.7}
$$

and thus,

$$
f \circ mu\langle r, d, G\rangle = f \circ (r * mu\langle d, d, G\rangle) = r * (f \circ mu\langle d, d, G\rangle).
$$

The rest of the diagram is concerned with how to look up a key $k \in \beta$ in $mu\langle d, d, G\rangle$: first use G to determine the structure of the corresponding node, e.g., "empty" (inl †), or "17 prepended to ..." (inr $(17, \ldots)$). However, instead of an actual zlist, the dots now contain a key which identifies the zlist. The step $\mathsf{S}mu\langle d, d, G\rangle$ is the recursion step, which replaces this key by the actual value. Notice that the functor S, in this case $\mathsf{S}f = id + id \times f$, "knows" precisely where in the node, keys have to be replaced by actual zlists. Thus, in the lower left corner of the diagram, we find values such as inl † and $inr(17, [19])$. We finish by applying τ, the initial algebra for $\mu\mathsf{S}$, which in our example replaces inl by nil and inr by $cons$. Expressed as a formula, diagram (6.6) becomes

$$\begin{array}{rcll} mu\langle r, d, G\rangle\ x & = & (\tau \circ \mathsf{S}mu\langle d, d, G\rangle \circ G)\ r(x), & \text{(general)} \\ mu\langle d, d, G\rangle & = & \tau \circ \mathsf{S}mu\langle d, d, G\rangle \circ G. & (\text{with } r = d) \end{array} \tag{6.8}$$

See figure 6.1 for an extended example.
Reminder: the procedure described here is not how inductive types are treated during processing. We are simply defining the semantics of the $mu\langle\rangle$ frame by stating how one extracts nested data from it. In the next section, we give a definition of τ at the frame level, to be used during query processing.

146 Frame equivalence. In the sequel we often need to compare $mu\langle\rangle$ frames for equality. Without proof we claim that

$$\begin{array}{c} mu\langle r, d, G\rangle = mu\langle r * \varphi,\ \varphi^{\cup} * d * \varphi,\ \mathsf{S}atom\langle\varphi\rangle \circ G' \circ atom\langle\varphi^{\cup}\rangle\rangle \\ \varphi \text{ is one-to-one} \end{array} \tag{6.9}$$

The intuition behind φ is that it maps the old internal keys to the new internal keys. Thus, if α are tree identifiers, β old node identifiers, and γ new node identifiers, then $r{*}\varphi : [\alpha \mapsto \gamma]$ and $\varphi^{\cup}{*}d{*}\varphi : [\gamma \mapsto !]$. Similarly, with $G : \beta \to \mathsf{S}\beta$,

$$\mathsf{S}atom\langle\varphi\rangle \circ G' \circ atom\langle\varphi^{\cup}\rangle : \gamma \to \mathsf{S}\gamma.$$

Again, the signature functor S knows precisely in which positions β needs to be replaced by γ using $atom\langle\varphi\rangle$.

147 Standard frame operations. We implement the fundamental frame operations as follows:

$$\begin{array}{rcl} \text{dom}\ mu\langle r, d, G\rangle & = & twin(r) \\ r' * mu\langle r, d, G\rangle & = & mu\langle r' * r, d, G\rangle \\ mu\langle r_1, d_1, G_1\rangle \sqcup mu\langle r_2, d_2, G_2\rangle & = & mu\langle \\ & & \quad (r_1 * sl^{\cup}) \cup (r_2 * sr^{\cup}), \\ & & \quad rtwin(ms), \\ & & \quad (\mathsf{S}atom\langle sl^{\cup}\rangle \circ G_1 \circ atom\langle sl\rangle) \sqcup \\ & & \quad (\mathsf{S}atom\langle sr^{\cup}\rangle \circ G_2 \circ atom\langle sr\rangle)\rangle \end{array}$$

$$
\begin{array}{ll}
 & F\ \alpha_2 \\
= & \\
 & mu\langle r, d, G\rangle\ \alpha_2 \\
= & \quad \{\ \text{equation (6.8)}\ \} \\
 & (\tau \circ \mathsf{S}mu\langle d, d, G\rangle \circ G)\ r(\alpha_2) \\
= & \quad \{\ \text{evaluate } r\ \} \\
 & (\tau \circ \mathsf{S}mu\langle d, d, G\rangle \circ G)\ \beta_2 \\
= & \quad \{\ \text{evaluate } G\ \} \\
 & (\tau \circ \mathsf{S}mu\langle d, d, G\rangle)\ (inr\ (17, \beta_3)) \\
= & \quad \{\ \text{apply } \mathsf{S},\ \tau = nil \triangledown cons\ \} \\
 & cons\ (17, mu\langle d, d, G\rangle\ \beta_3) \\
= & \quad \{\ \text{equation (6.8)}\ \} \\
 & cons\ (17, (\tau \circ \mathsf{S}mu\langle d, d, G\rangle \circ G)\ \beta_3) \\
= & \quad \{\ \text{evaluate } G\ \} \\
 & cons\ (17, (\tau \circ \mathsf{S}mu\langle d, d, G\rangle)\ (inr\ (19, \beta_4))) \\
= & \quad \{\ \text{apply } \mathsf{S},\ \tau = nil \triangledown cons\ \} \\
 & cons\ (17, (cons\ (19, mu\langle d, d, G\rangle\ \beta_4))) \\
= & \quad \{\ \text{equation (6.8)}\ \} \\
 & cons\ (17, (cons\ (19, (\tau \circ \mathsf{S}mu\langle d, d, G\rangle \circ G)\ \beta_4))) \\
= & \quad \{\ \text{evaluate } G\ \} \\
 & cons\ (17, (cons\ (19, (\tau \circ \mathsf{S}mu\langle d, d, G\rangle)\ (inl\ \dagger)))) \\
= & \quad \{\ \text{apply } \mathsf{S},\ \tau = nil \triangledown cons\ \} \\
 & cons\ (17, (cons\ (19, (nil\ \dagger)))) \\
= & \quad \{\ \text{graphical representation}\ \}
\end{array}
$$

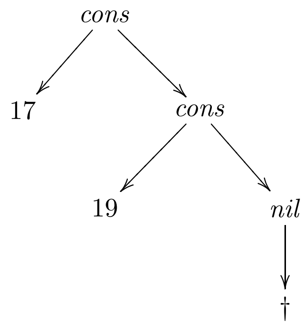


Figure 6.1: The interpretation rule for $mu\langle\rangle$-frames in action. Three times, equation (6.8) is used to expand the $mu\langle\rangle$-frame, each time followed by an application of G to obtain the node structure. Applying $\mathsf{S}mu\langle d, d, G\rangle$ means looking up the node keys yielded by G and recursively interpreting the result. The first two times, this yields a new node. The final time, for the empty list, it leaves the node unchanged.

$$
\begin{aligned}
ms &= mksum(sethead(d_1, \dagger), sethead(d_2, \dagger)) \\
&\qquad\qquad : [1 \leftrightarrow \beta + \gamma] \\
sl &= sumleft(\ldots) : [\beta + \gamma \leftrightarrow \beta] \\
sr &= sumright(\ldots) : [\beta + \gamma \leftrightarrow \gamma]
\end{aligned}
$$

The essential step in the rule for $\sqcup$ is that we construct a new keyspace which encompasses all nodes in both $mu\langle\rangle$-frames. Suppose the nodes of $mu\langle r_1, d_1, G_1\rangle$ are identified by keys in β and those in $mu\langle r_2, d_2, G_2\rangle$ by keys in γ. We use *mksum*, *sumleft* and *sumright* as defined in paragraph 63 to construct a new key space $\beta + \gamma$. The column $ms : [1 \leftrightarrow \beta + \gamma]$ enumerates all new keys, and $sl : [\beta + \gamma \leftrightarrow \beta]$ and $sr : [\beta + \gamma \leftrightarrow \gamma]$ map them back to β and γ, respectively. Having these, we simply convert G_1 and G_2 to the new key space, and adjust the mapping from outer values to nodes accordingly.

6.3 Rewrite rule for the initial algebra

We give frame implementations of two canonical operations on inductively defined types: constructing them using their initial algebra, e.g., $\tau = nil \triangledown cons$, and deconstructing them using a catamorphism $(\!|\cdot|\!)$. In the current section, we consider τ. In a later section, we look at $(\!|\cdot|\!)$.

148 Decomposition. Initial S-algebras $\tau_{\mathsf{S}} : \mathsf{S}(\mu\mathsf{S}) \to \mu\mathsf{S}$ are completely determined by their signature functor S, because S describes precisely how values of type $\mu\mathsf{S}$ are constructed. Given arbitrary S, we wish to derive a frame rule for the corresponding initial algebra τ_{S}, usually just written τ. To derive a frame rule for τ, given a frame $H : \alpha \to \mathsf{S}(\mu\mathsf{S})$ we need to construct a replacement frame H' for the composition $\tau \circ H : \alpha \to \mu\mathsf{S}$.

For general S, we do not explicitly know the frame structure of H. Based on the type $\mathsf{S}(\mu\mathsf{S})$ we know, however, that we can split it into a frame for $\mu\mathsf{S}$, encapsulated in a frame for $\mathsf{S}(_)$. At the position marked by the $_$, this outer frame $H_0 : \alpha \to \mathsf{S}\beta$ will produce keys β. The encapsulated $mu\langle\rangle$-frame $mu\langle r, d, G\rangle : \beta \to \mu\mathsf{S}$ then takes these β to $\mu\mathsf{S}$. Thus, we can always write H as $H = \mathsf{S}\, mu\langle r, d, G\rangle \circ H_0$ with $H_0 : \alpha \to \mathsf{S}\beta$. Pictorially, when rewriting $\tau \circ H$ we are trying to obtain $H' = mu\langle r', d', G'\rangle$ in the commutative diagram

$$
\alpha \xrightarrow{H_0} \mathsf{S}\beta \xrightarrow{\mathsf{S}\, mu\langle r,d,G\rangle} \mathsf{S}(\mu\mathsf{S}) \xrightarrow{\tau} \mu\mathsf{S}\ .
\qquad (H : \alpha \to \mathsf{S}(\mu\mathsf{S}),\quad H' = mu\langle r', d', G'\rangle : \alpha \to \mu\mathsf{S})
$$

Here, H_0 encodes the outer S structure, with the values of $\mu\mathsf{S}$ replaced by keys β. The term $\mathsf{S}mu\langle r, d, G\rangle$ replaces those keys by the actual values, thus achieving the decomposition of H.

Notice that if $\mathsf{S}\,X$ uses its argument X more than once, e.g., $\mathsf{S}\,X = 1+X\times X$, then the frame for $\mathsf{S}(\mu\mathsf{S})$ contains more than one $\mu\mathsf{S}$ frame. All these frames have to be combined into $mu\langle r, d, G\rangle$ in order to reach the decomposition $H = \mathsf{S}mu\langle r, d, G\rangle \circ H_0$. This adds some extra bookkeeping but does not fundamentally alter our approach. In the rest of this chapter we will generally ignore this complication.

149 Rewrite rule. Given $H = \mathsf{S}mu\langle r, d, G\rangle \circ H_0 : \alpha \to \mathsf{S}(\mu\mathsf{S})$ with

$$\begin{array}{rclrcl} H_0 & : & \alpha \to \mathsf{S}\beta, & r & : & [\beta \mapsto \gamma], \\ mu\langle r, d, G\rangle & : & \beta \to \mu\mathsf{S}, & d & : & [\gamma \mapsto !], \\ & & & G & : & \gamma \to \mathsf{S}\gamma, \end{array} \tag{6.10}$$

we look for a frame $H' = mu\langle r', d', G'\rangle$ which satisfies $H' = \tau \circ H$. Such a frame is given by the equations

$$\begin{array}{rcl} H' & = & mu\langle r', d', G'\rangle : \alpha \to \mu\mathsf{S}, \quad \text{with} \\ r' & = & sr^{\cup} : [\alpha \mapsto \gamma + \alpha], \\ d' & = & rtwin(ms) : [\gamma + \alpha \mapsto !], \\ G' & = & (\mathsf{S}atom\langle sl^{\cup}\rangle \circ G \circ atom\langle sl\rangle) \sqcup \\ & & (\mathsf{S}atom\langle r * sl^{\cup}\rangle \circ H_0 \circ atom\langle sr\rangle) : (\gamma + \alpha) \to \mathsf{S}(\gamma + \alpha), \\ ms & = & mksum(sethead(d, \dagger), sethead(\mathrm{dom}\ \ H_0, \dagger)) : [1 \mapsto \gamma + \alpha], \\ sl & = & sumleft(\ldots) : [\gamma + \alpha \mapsto \gamma], \\ sr & = & sumright(\ldots) : [\gamma + \alpha \mapsto \alpha]. \end{array} \tag{6.11}$$

The new frame contains all old nodes, which in H were identified by γ, plus a new node for every key in α. Therefore, the internal key space of the new frame is $\gamma+\alpha$. We build an explicit representation of $\gamma+\alpha$ using *mksum*, *sumleft* and *sumright*.

Consider every part of $mu\langle r', d', G'\rangle$ in turn. The relation $r' = sr^{\cup}$ maps every outer key to the corresponding newly constructed inner key. The middle component d' is an explicit listing of the keys in $\gamma + \alpha$. Finally, G' is the union of two sets of nodes. The first set, $\mathsf{S}atom\langle sl^{\cup}\rangle \circ G \circ atom\langle sl\rangle$, is simply the old nodes as stored in G, but translated to the new key space. The structure of the new nodes is determined by $H_0 : \alpha \to \mathsf{S}\beta$. We use sr to translate from $\gamma + \alpha$ to α, and after applying H_0, the β keys in the result have to be translated back to $\gamma + \alpha$. We do this in two steps. In the first step, r, we translate outer keys β of the old $mu\langle\rangle$-frame to inner keys γ of the old mu frame. In the second step, sl, these are translated to inner keys of the new $mu\langle\rangle$ frame.

150 Proof. The following pointwise calculation proves that with H as in Equation (6.10) and H' as in (6.11), $H' = \tau \circ H$:

$$
\begin{array}{lll}
 & & H'\ k \\
 & = & \quad \{ \text{ Expand } H' \ \} \\
 & & mu\langle r', d', G' \rangle\ k \\
 & = & \quad \{ \text{ Interpretation of } mu\langle\rangle \text{, Eq. (6.8) } \} \\
 & & (\tau \circ \mathsf{S}mu\langle d', d', G' \rangle \circ G')\ r'(k) \\
 & = & \quad \{ \text{ Rearrange, } r' = sr^{\cup} \ \} \\
 & & (\tau \circ \mathsf{S}mu\langle d', d', G' \rangle)\ G'(sr^{\cup}(k)) \\
 & = & \quad \{ \text{ Expand } G' \text{ as in (6.11), know } sl * sr^{\cup} = \emptyset \ \} \\
 & & (\tau \circ \mathsf{S}mu\langle d', d', G' \rangle)\ ((\mathsf{S}atom\langle r * sl^{\cup} \rangle \circ H_0)\ k) \\
 & = & \quad \{ \text{ Rearrange } \} \\
 & & (\tau \circ \mathsf{S}mu\langle d', d', G' \rangle \circ \mathsf{S}atom\langle r * sl^{\cup} \rangle \circ H_0)\ k \\
 & = & \quad \{ \text{ Combine S functors, } d' \text{ is domain complete identity relation } \} \\
 & & (\tau \circ \mathsf{S}mu\langle r * sl^{\cup}, d', G' \rangle \circ H_0)\ k \\
 & = & \quad \{ \text{ Frame equivalence, use Eq. (6.9) with } \varphi = sl \ \} \\
 & & (\tau \circ \mathsf{S}mu\langle r * sl^{\cup} * sl, sl^{\cup} * d' * sl, \mathsf{S}atom\langle sl \rangle \circ G' \circ atom\langle sl^{\cup} \rangle \rangle \circ H_0)\ k \\
 & = & \quad \{ \text{ Simplify, expand } G' \ \} \\
 & & (\tau \circ \mathsf{S}mu\langle r, sl^{\cup} * d' * sl, G \rangle \circ H_0)\ k \\
 (*) & = & \quad \{ \text{ Definition of } mksum\text{, } rtwin \text{ in } sl \text{ and } d' \ \} \\
 & & (\tau \circ \mathsf{S}mu\langle r, d, G \rangle \circ H_0)\ k \\
 & = & \quad \{ \text{ Recognize } H = mu\langle r, d, G \rangle \ \} \\
 & & (\tau \circ H)\ k.
\end{array}
$$

Thus we find $H'\ k = (\tau \circ H)\ k$. The only nontrivial step in this calculation is the one marked $(*)$, which is verified easily by inspection of the definitions in Chapter 3.

6.4 Implementing catamorphisms by iteration

151 Implementing catamorphisms. Having defined a frame representation for inductively defined types, the next step is to figure out how to evaluate catamorphisms such as $sum = (\!|\underline{0} \triangledown +|\!)$ over it. The basic principle is simple. The left hand side of a rewrite rule for Dodo generally consists of an operation

composed with a frame. The right hand side consists of a transformed frame which incorporates the action of the operation. Thus, the general form of a rewrite rule for catamorphisms is

$$(\!|\alpha|\!) \circ mu\langle r, d, G\rangle = H, \tag{6.12}$$

with H a frame. Our task is to figure out what H is.

By combining (6.4), (6.6) and (6.12) (twice!) we directly obtain a commutative diagram from which we can read off the law which H must satisfy to fit the above equation. As discussed in paragraph 145, for simplicity we consider $mu\langle d, d, G\rangle$ rather than $mu\langle r, d, G\rangle$.

$$\begin{array}{ccc}
\mathsf{S}\beta & \xleftarrow{\;G\;} & \beta \\
\downarrow{\scriptstyle \mathsf{S}mu\langle d,d,G\rangle} & & \downarrow{\scriptstyle mu\langle d,d,G\rangle} \\
\mathsf{S}(\mu\mathsf{S}) & \underset{\tau^{\cup}}{\overset{\tau}{\rightleftarrows}} & \mu\mathsf{S} \\
\downarrow{\scriptstyle \mathsf{S}(\!|\alpha|\!)} & & \downarrow{\scriptstyle (\!|\alpha|\!)} \\
\mathsf{S}A & \xrightarrow{\;\alpha\;} & A
\end{array}
\qquad \mathsf{S}H : \mathsf{S}\beta \to \mathsf{S}A, \quad H : \beta \to A$$

$$H = (\!|\alpha|\!) \circ mu\langle d, d, G\rangle = \alpha \circ \mathsf{S}H \circ G. \tag{6.13}$$

Thus, when Dodo encounters $(\!|\alpha|\!) \circ mu\langle r, d, G\rangle$, it should replace it by $r * H$, with H a frame satisfying $H = \alpha \circ \mathsf{S}H \circ G$. In the rest of this section we examine how Dodo obtains such H.

152 Example. Consider $F' = mu\langle d, d, G\rangle$ with d and G as in (6.5). This F' is similar to the F in that equation, except that $r = d$. We know that composing F' with the function $sum = (\!|\underline{0} \triangledown +|\!)$ yields

$$H = sum \circ F' = \left\{\begin{array}{l} \beta_1 \mapsto 0, \\ \beta_2 \mapsto 36, \\ \beta_3 \mapsto 19, \\ \beta_4 \mapsto 0 \end{array}\right\},$$

and indeed

$$sum \circ \mathsf{S}H \circ G \;=\; (\!|\underline{0} \triangledown +|\!) \circ \mathsf{S}\left\{\begin{array}{l} \beta_1 \mapsto 0, \\ \beta_2 \mapsto 36, \\ \beta_3 \mapsto 19, \\ \beta_4 \mapsto 0 \end{array}\right\} \circ \left\{\begin{array}{l} \beta_1 \mapsto inl\ \dagger, \\ \beta_2 \mapsto inr\ (17, \beta_3), \\ \beta_3 \mapsto inr\ (19, \beta_4), \\ \beta_4 \mapsto inl\ \dagger \end{array}\right\}$$

$$= \quad (\!|\underline{0} \triangledown +|\!) \circ \left\{\begin{array}{l} \beta_1 \mapsto \mathit{inl}\ \dagger, \\ \beta_2 \mapsto \mathit{inr}\ (17,19), \\ \beta_3 \mapsto \mathit{inr}\ (19,0), \\ \beta_4 \mapsto \mathit{inl}\ \dagger \end{array}\right\} = \left\{\begin{array}{l} \beta_1 \mapsto 0, \\ \beta_2 \mapsto 17+19, \\ \beta_3 \mapsto 19+0, \\ \beta_4 \mapsto 0 \end{array}\right\} = H.$$

153 Our approach, informally. The problem in determining H is that the result for certain keys, such as β_2 in the above example, depends on the result for other keys. This is inherent to the structure of the *mu*$\langle\rangle$ frame, in which nodes refer to other nodes. To deal with this, we compute H in stages. We extend the the range type of H with a special value, bottom, which means "not yet known." Then, we repeatedly evaluate a modified version of equation (6.13), until for every key, the value has become known. An important property of this iteration is that it works in parallel over all nodes in the frame. In the first iteration, the catamorphism is evaluated for every subtree of depth 0, e.g., every leaf. The next iteration deals with subtrees of depth 1, which can now be processed because the partial result at their children is now known. The iteration goes on until every node is "known."

To extend the domain of H with the value "unknown," we use the bottom functor $\bot X$. If $H : \beta \to A$, the frames for the partial results have type $\beta \to \bot A$. This requires some modifications to Equation (6.13). If we substitute $\hat{H} : \beta \to \bot A$ into (6.13), we obtain $\mathsf{S}\hat{H} \circ G : \beta \to \mathsf{S}(\bot A)$. However, α expects arguments of type $\mathsf{S}A$. The solution we choose is similar to our solution to flattening λ-terms in Section 2.5.3: we introduce a family of operators $E_{\mathsf{S}} : \mathsf{S}(\bot A) \to \bot(\mathsf{S}A)$ which returns "known" values only if all $\bot A$ within the S are known. For a tuple type, for instance, this means that all constituents are known. Using E_{S}, we can protect α from working on arguments which are not yet known:

$$\hat{H}_{j+1} = \bot\alpha \circ E_{\mathsf{S}} \circ \mathsf{S}\hat{H}_j \circ G. \tag{6.14}$$

154 The bottom type. The most straightforward implementation of $\bot X$ is of course $\bot X = 1+X$. Then, we denote unknown values with *inl* † and known values with *inr* x. In Dodo, however, we prefer to create a special $\bot$ type with constructors *nothing* : $1 \to \bot X$ and *just* : $X \to \bot X$. This type can then be implemented with the specific rewrite scheme for catamorphisms in mind, keeping the point-free expressions more readable, and the frame representation less verbose and possibly more efficient.

Here is the frame representation of the bottom type: bottom types are represented using a frame *bot*$\langle d, F\rangle$, where d is the domain of the whole frame, and the domain of F is restricted to the keys of the known values. As a consequence, if all values are unknown, dom $F = \emptyset$, and if all values are known, dom $F = d$.

The interpretation of the $bot\langle\rangle$-frame is given by

$$\begin{array}{rcll} bot\langle d, F\rangle\ k & = & nothing\ \dagger, & \text{if } k \notin \operatorname{dom} F \\ bot\langle d, F\rangle\ k & = & just\ (F\ k), & \text{if } k \in \operatorname{dom} F \end{array}$$

The standard frame operations on it are

$$\begin{array}{rcl} dom\ bot\langle d, F\rangle & = & d, \\ r * bot\langle d, F\rangle & = & bot\langle twin(r * d), r * F\rangle, \\ bot\langle d_1, F_1\rangle \sqcup bot\langle d_2, F_2\rangle, & = & bot\langle d_1 \cup d_2, F_1 \sqcup F_2\rangle. \end{array}$$

One can think of many different internal representations for the $\perp$ type. This is just one of them. The choice depends only on what is most convenient when processing iteration at the column level, see paragraph 162.

155 The extraction functions E_{S}. The definition of $E_{\mathsf{S}} : \mathsf{S}\perp X \to \perp\mathsf{S}X$ for arbitrary functor S is that for every appropriately typed x and y,

$$E_{\mathsf{S}}\ x = just\ y \iff x = \mathsf{S}\ just\ y. \tag{6.15}$$

It follows from this definition that $E_{\mathsf{S}}\ x = nothing\ \dagger$ for those x which are not of the form $x = \mathsf{S}\ just\ y$. For example, if $\mathsf{S}\ X = X \times X$, then

$$\begin{array}{rcl} E_{\mathsf{S}}\ (just\ 1, just\ 2) & = & just\ (1, 2), \\ E_{\mathsf{S}}\ (just\ 1, nothing\ \dagger) & = & nothing\ \dagger\ . \end{array}$$

Notice that E_{S} can be regarded as the dual of D_{S}. The distribution function $D_{\mathsf{S}} : A \times \mathsf{S}B \to \mathsf{S}(A \times B)$ distributes the product $(A\times)$ *into* S, where $E_{\mathsf{S}} : \mathsf{S}(1 + B) \to 1 + \mathsf{S}B$ extracts the sum $(1+)$ *out* of S.

156 Theorem (catamorphism iteration). Let S be a signature functor, $\alpha : \mathsf{S}A \to A$, and $mu\langle d, d, G\rangle : \beta \to \mu\mathsf{S}$ a fixpoint frame with $G : \beta \to \mathsf{S}\beta$. Define the sequence of frames $\{\hat{H}_j\}_j$ as

$$\begin{array}{rcl} \hat{H}_0\ k & = & nothing\ \dagger, \\ \hat{H}_{i+1}\ k & = & (\perp\alpha \circ E_{\mathsf{S}} \circ \mathsf{S}\hat{H}_i \circ G)\ k. \end{array} \tag{6.16}$$

If there exist H and j such that $\hat{H}_j$ is of the form $\hat{H}_j = just \circ H$, then H satisfies $H = \alpha \circ \mathsf{S}H \circ G$, and thus $H = (\!|\alpha|\!) \circ mu\langle d, d, G\rangle$.

Proof We begin with showing that the sequence $\{\hat{H}_j\}_j$ is monotonous, that is, that

$$\hat{H}_j\ k = just\ x \implies \hat{H}_{j+1}\ k = just\ x, \qquad (P_j)$$

for all j. We call this monotonous because whenever a value becomes known, it remains unchanged afterwards. The proof is by induction to j. First notice that (P_0) holds trivially because $\hat{H}_0\ k = nothing\ \dagger$ for every k. Then, assuming (P_{j-1}) holds, we find

$$\begin{array}{ll}
& \hat{H}_j\ k = just\ x_1 \\
\equiv & \quad \{ \text{ expand } \hat{H}_j \text{ according to (6.16) } \} \\
& (\bot\alpha \circ E_{\mathsf{S}} \circ \mathsf{S}\hat{H}_{j-1} \circ G)\ k = just\ x_1 \\
\equiv & \quad \{ \text{ There exists } x_2 \text{ such that } x_1 = \alpha\ x_2 \ \} \\
& (E_{\mathsf{S}} \circ \mathsf{S}\hat{H}_{j-1} \circ G)\ k = just\ x_2 \\
\equiv & \quad \{ \text{ Eq. (6.15): } E_{\mathsf{S}}\ x = just\ y \iff x = \mathsf{S}\ just\ y. \ \} \\
& (\mathsf{S}\hat{H}_{j-1} \circ G)\ k = \mathsf{S}\ just\ x_2 \\
\implies & \quad \{ \text{ Property } (P_{j-1}) \ \} \\
& (\mathsf{S}\hat{H}_j \circ G)\ k = \mathsf{S}\ just\ x_2 \\
\equiv & \quad \{ \text{ Eq. (6.15): } E_{\mathsf{S}}\ x = just\ y \iff x = \mathsf{S}\ just\ y. \ \} \\
& (E_{\mathsf{S}} \circ \mathsf{S}\hat{H}_j \circ G)\ k = just\ x_2 \\
\equiv & \quad \{ \ x_1 = \alpha\ x_2 \ \} \\
& (\bot\alpha \circ E_{\mathsf{S}} \circ \mathsf{S}\hat{H}_j \circ G)\ k = just\ x_1 \\
\equiv & \quad \{ \text{ recognize as } \hat{H}_{j+1} \ \} \\
& \hat{H}_{j+1}\ k = just\ x_1
\end{array}$$

Thus, (P_{j-1}) implies (P_j) and therefore (P_j) holds for every j. We conclude that once the value for a particular key k becomes known, it does not change in later iterations.

It follows immediately that if $\hat{H}_j$ can be written as $\hat{H}_j = just \circ H$ for certain H, then also $\hat{H}_{j+1} = just \circ H$. Moreover,

$$\begin{array}{ll}
& just \circ H \\
= & \\
& \hat{H}_{j+1} \\
= & \\
& \bot\alpha \circ E_{\mathsf{S}} \circ \mathsf{S}\hat{H}_j \circ G \\
= & \\
& \bot\alpha \circ E_{\mathsf{S}} \circ \mathsf{S}just \circ \mathsf{S}H \circ G \\
= & \quad \{ \text{ Definition of } E_{\mathsf{S}} \ \}
\end{array}$$

$$= \frac{\bot\alpha \circ \mathit{just} \circ \mathsf{S}H \circ G}{\mathit{just} \circ \alpha \circ \mathsf{S}H \circ G}$$

and thus $H = \alpha \circ \mathsf{S}H \circ G$, which proves the theorem. □

157 Convergence. The above theorem guarantees that iteration (6.16) converges, because all frames $\hat{H}_j : \beta \to \bot A$ have the same finite domain β and for every key, the value changes at most once, from *nothing* to *just*. Therefore, the number of iterations is bound by the number of keys in the frame. Usually, the number of iterations is related to the depth of the nesting in the data structure.

The theorem does not exclude the possibility that the iteration converges to a state where not all nodes are known. This happens, for instance, if G in $mu\langle d, d, G\rangle$ has cyclic references between nodes. In such cases, the computation of $(\!|\alpha|\!)$ simply does not terminate for those nodes. Dodo cannot be expected to compute answers to nonterminating queries, so the most straightforward response is to simply abort query evaluation whenever unknown values remain after iteration. This can be refined by only aborting execution when the unknown values are actually needed. Recall that, to evaluate $(\!|\alpha|\!) \circ mu\langle r, d, G\rangle$, we compute $H = (\!|\alpha|\!) \circ mu\langle d, d, G\rangle$. We do not need *all* values of H to become known, only those referenced by r. Therefore, after iteration has terminated, it is sufficient to check whether $r * \hat{H}_j$ has the form $r * \hat{H}_j = \mathit{just} \circ H$ and then use H as the replacement for $(\!|\alpha|\!) \circ mu\langle r, d, G\rangle$. In fact, this condition may be used to terminate iteration early even in non-cyclic cases.

158 Example: sum. Consider again the frame F of equation (6.5). To calculate $\mathit{sum} \circ F = (\!|\underline{0} \triangledown + |\!) \circ F$, we have to solve $H = \alpha \circ \mathsf{S}H \circ G$. In paragraph 152 we demonstrated that this indeed yields the correct result. Based on the type $\beta \to \bot\mathbb{Z}$, we infer that the $\hat{H}_j$ is of the form $\mathit{bot}\langle d_j, \mathit{atom}\langle h_j\rangle\rangle$. Likewise, using abbreviated notation, G is of the form $e\langle a\langle e\rangle, p\langle a\langle f\rangle, a\langle s\rangle\rangle\rangle$ for certain columns e, f and s. Thus, most elements in the equation $\hat{H}_{j+1} = \bot\alpha \circ \mathsf{S}\hat{H}_j \circ G$ are now known. Adding E_{S} for $\mathsf{S}X = 1 + \mathbb{Z} \times X$, we obtain the following:

$$\begin{array}{rcl} \alpha & = & \underline{0} \triangledown (+) \\ \hat{H}_j & = & \mathit{bot}\langle d_j, a\langle h_j\rangle\rangle \\ G & = & e\langle a\langle e\rangle, p\langle a\langle f\rangle, a\langle s\rangle\rangle\rangle \\ E_{\mathsf{S}} \circ e\langle a\langle x\rangle, p\langle a\langle y\rangle, \mathit{bot}\langle z, Z\rangle\rangle\rangle & = & \mathit{bot}\langle x \cup z, e\langle a\langle x\rangle, p\langle \mathrm{dom}\ Z * a\langle y\rangle, Z\rangle\rangle\rangle \end{array}$$

Now we substitute these into $\hat{H}_{j+1} = \bot\alpha \circ \mathsf{S}\hat{H}_j \circ G$ step by step:

$$
\begin{aligned}
& G = e\langle a\langle e\rangle, p\langle a\langle f\rangle, a\langle s\rangle\rangle\rangle \\
\Longrightarrow\quad & \{ \text{ Prepend } \mathsf{S}\hat{H}_j \text{, calculate effect on frame expression } \} \\
& \mathsf{S}\hat{H}_j \circ G = e\langle a\langle \underbrace{e}_{x}\rangle, p\langle a\langle \underbrace{f}_{y}\rangle, bot\langle \underbrace{twin(s * d_j)}_{z}, \underbrace{a\langle s * h_j\rangle}_{Z}\rangle\rangle\rangle \\
\Longrightarrow\quad & \{ \text{ Plug } x, y, z \text{ and } Z \text{ as above into definition of } E_{\mathsf{S}} \} \\
& E_{\mathsf{S}} \circ \mathsf{S}\hat{H}_j \circ G = bot\langle e \cup twin(s * d_j), \\
& \qquad e\langle a\langle e\rangle, p\langle a\langle twin(s * h_h) * f\rangle, a\langle s * h_j\rangle\rangle\rangle\rangle \\
\Longrightarrow\quad & \bot\alpha \circ E_{\mathsf{S}} \circ \mathsf{S}\hat{H}_j \circ G = bot\langle e \cup twin(s * d_j), \\
& \qquad a\langle settail(e, 0) \cup op_{+}(twin(s * h_h) * f, s * h_j)\rangle\rangle \\
\Longrightarrow\quad & \{ \text{ Recognize } \hat{H}_{j+1} \} \\
& \hat{H}_{j+1} = bot\langle d_{j+1}, a\langle h_{j+1}\rangle\rangle = bot\langle e \cup twin(s * d_j), \\
& \qquad a\langle settail(e, 0) \cup op_{+}(twin(s * h_h) * f, s * h_j)\rangle\rangle,
\end{aligned}
$$

Both forms of $\hat{H}_{j+1}$ are of the form $bot\langle \ldots, a\langle \ldots\rangle\rangle$. Thus, we conclude that at the column level,

$$
\begin{aligned}
\hat{H}_j &= bot\langle d_j, a\langle h_j\rangle\rangle, \\
d_{j+1} &= e \cup twin(s * d_j), \\
h_{j+1} &= settail(e, 0) \cup op_{+}(twin(s * h_h) * f, s * h_j).
\end{aligned}
$$

Determining the initial values is easy: we need $\operatorname{dom} \hat{H}_j = \operatorname{dom} F$, therefore $d_0 = d$. Similarly, because initially every value is unknown, $\operatorname{dom} a\langle h_0\rangle = \emptyset$, thus $h_0 = \emptyset$. The iteration stops when $\hat{H}_j = just\ H$ for certain H and j, i.e., when $\operatorname{dom} a\langle h_j\rangle = \operatorname{dom} bot\langle d_j, a\langle h_j\rangle\rangle$, meaning $twin(h_j) = d_j$.

Performing this iteration, we find

d_j	
β_1	β_1
β_2	β_2
β_3	β_3
β_4	β_4

h_0	

h_1	
β_1	0
β_4	0

h_2	
β_1	0
β_4	0
β_3	$19+0$

h_3	
β_1	0
β_4	0
β_3	$19+0$
β_2	$17+19$

Notice that in fact, $d_j = d$ for every j. This is easy to derive automatically from the calculation above, but follows directly from $\operatorname{dom} \hat{H}_{j+1} = \operatorname{dom} \hat{H}_j$.

6.5 Implementing iteration in the column layer

159 Adding iteration to Dodo. In the previous section we discussed how catamorphisms can be implemented as a repeated frame transformation.

We have not yet discussed how this can be implemented concretely at the column level. Dodo as described in Chapter 2, without inductive types, is rather straightforward to implement. Simply begin with a query and transform it to point-free form. Then gradually transform the point-free query into a large frame expression representing the query result. Take the column expressions out of the frame and evaluate them, possibly after eliminating commons subexpressions. Finally, in cooperation with the application on top of Dodo, turn the frame expression into a nested return value ("interpretation function").

When inductively defined types are added, we have to fit iteration into this scheme. In the following paragraphs we briefly describe two possible approaches.

160 Approach 1: heavy-weight rewrite rule. In the first approach, the iteration is implemented inside a huge rewrite rule, which, when encountering $(\!|\alpha|\!) \circ mu\langle r, d, G\rangle$, basically constructs a series of new queries of the form $\bot\alpha \circ E_{\mathsf{S}} \circ \mathsf{S}\hat{H} \circ G$ and evaluates them in another Dodo instance. Once the iteration stabilizes, the rewrite rule replaces $(\!|\alpha|\!) \circ mu\langle r, d, G\rangle$ by the result of the iteration. The benefit of this approach is that it requires very little change in the rest of Dodo. It is simply a rewrite rule which takes a subquery $f \circ F$ and replaces it by an equivalent frame F'. On the other hand, repeatedly invoking other Dodo instances makes this rewrite rule *very* expensive. It also does not fit well with the white-box approach behind Dodo, because there now is a whole mechanism locked up within a single component, a single rewrite rule. In a layered design, query rewriting and query execution should generally not be mixed in such a way. Especially, query evaluation should not be *hidden* in a rewrite rule.

161 Approach 2: an iteration frame. The second approach we describe here is to make the column execution layer aware of the notion of iteration. We need to introduce a new language construct which implements iteration somewhere at the boundary of the frame sublanguage and the column sublanguage. Then, the rewrite rule for $(\!|\alpha|\!) \circ F$ can generate an iteration expression and the execution engine can perform the iteration, separating the tasks of the layers.

In contrast with the rest of this chapter, where for many problems there was a "canonical solution," the implementation we give here is just one of many possible implementations. Many variations are possible, some of which might benefit from changes to the implementation of the $bot\langle\rangle$ frame. This is one of the reasons we introduced $bot\langle\rangle$ as a functor with a custom frame representation.

The most important property of the column-level iteration construct proposed here, is that to the rest of Dodo, it looks like a frame. It is written $iter\langle\rangle$, has dom and $\sqcup$ operations, etc. Only the execution engine treats it specially. During execution, normal frames are traversed and copied as-is, replacing col-

umn expressions by corresponding result columns. However, $iter\langle\rangle$ frames are not simply copied, they are iterated and replaced by the result of the iteration.

162 Iteration construct. Our iteration construct has the following structure:

$$\begin{array}{ll} iter\langle & result : \alpha \to A, \\ & domain : [\alpha \mapsto\!\!\!\rightarrow !] \\ & template : \beta \to \bot A, \\ & initial : \beta \to \bot A, \\ & step : \beta \to \bot A, \\ & condition : [\beta \mapsto\!\!\!\rightarrow !] \\ & \rangle : \alpha \to A \end{array}$$

Interpretation function: the meaning of $iter\langle res, dom, templ, init, step, cond\rangle\ k$ is $res\ k$, with template variables in res determined by the final iteration of the following process: the first iteration executes $init$ and assigns the results to the corresponding names found in $templ$. Subsequent iterations execute $step$ with the template variables as bound by the previous iteration. After execution, the template variables are rebound according to $templ$. Iteration stops when the column expression $cond$ no longer changes.

163 Example. With the sum iteration as given in paragraph 158, the iteration frame for

$$(\!|\underline{\triangledown}(+)|\!) \circ mu\langle d, d, e\langle a\langle e\rangle, p\langle a\langle f\rangle, a\langle s\rangle\rangle\rangle\rangle$$

would be

$$\begin{array}{lll} iter\langle & result = atom\langle h^*\rangle, & \\ & domain = d, & \\ & template = bot\langle d^*, atom\langle h^*\rangle\rangle, & \\ & initial = bot\langle d, atom < emptycol()\rangle >, & \\ & step = bot\langle & e \cup twin(s * d^*), \\ & & atom\langle settail(e, 0) \cup op_+(twin(s * h^*) * f, s * h^*)\rangle\rangle, \\ & condition = h^*, & \\ & \rangle : \alpha \to A & \end{array}$$

By pattern matching against $template$, the execution engine can extract d^* and h^* from $initial$ and $step$. Matching against $initial$, this yields

$$d_0 \quad = \quad d^* = d,$$

$$h_0 \quad = \quad h^* = \mathit{emptycol}().$$

Against *step*, it yields

$$\begin{aligned} d_{j+1} &= d^* = e \cup \mathit{twin}(s * d_j), \\ h_{j+1} &= h^* = \mathit{settail}(e, 0) \cup \mathit{op}_+(\mathit{twin}(s * h^*) * f, s * h^*). \end{aligned}$$

164 Frame properties of iteration construct. Except for execution, iteration frames behave as normal frames:

$$\begin{aligned} f \circ \mathit{iter}\langle R, d, T, I, S, c\rangle &= \mathit{iter}\langle f \circ R, \mathit{twin}(r * d), T, I, S, c\rangle, \\ \mathrm{dom}\ \mathit{iter}\langle R, d, T, I, S, c\rangle &= d, \qquad \text{(this we needed } d \text{ for)} \\ r * \mathit{iter}\langle R, d, T, I, S, c\rangle &= \mathit{iter}\langle r * R, \mathit{twin}(r * d), T, I, S, c\rangle, \\ \mathit{iter}\langle R, d, T, I, S, c\rangle \sqcup F &= \mathit{iter}\langle R \sqcup F, d, T, I, S, c\rangle, \\ F \sqcup \mathit{iter}\langle R, d, T, I, S, c\rangle &= \mathit{iter}\langle F \sqcup R, d, T, I, S, c\rangle. \end{aligned}$$

In the $(r*)$, $(f\circ)$ and $(F\sqcup)$ rules we see that everything which happens *after* the iteration is simply incorporated into the *result* component. Queries which perform multiple iterations sequentially, e.g., $(\!|\alpha|\!) \circ \mathit{foo} \circ (\!|\beta|\!)$ lead to nesting in the *result* component. Such nesting indeed means that first the outer $\mathit{iter}\langle\rangle$ frame is evaluated, and then the results are pasted into template variables of the inner $\mathit{iter}\langle\rangle$ frame. On the other hand, nested catamorphisms, e.g., $(\!|f\ (\!|\alpha|\!)|\!)$, lead to nesting in the *step* component, which leads to nested execution of $\mathit{iter}\langle\rangle$ frames: Every iteration of the outer frame involves a complete execution of the inner frame.

165 Problem: internal types. Theorem 156 states that the equation $H = \alpha \circ \mathsf{S}H \circ G$ can be solved by computing the sequence of frames

$$\hat{H}_{j+1} = (\bot\alpha \circ E_{\mathsf{S}} \circ \mathsf{S}\hat{H}_j \circ G). \qquad (*)$$

Assuming a frame representation for $\hat{H}_j$ involving columns $h_j^1, \ldots, h_j^n$, expansion of $(\bot\alpha \circ E_{\mathsf{S}} \circ \mathsf{S}\hat{H}_j \circ G)$ under the Dodo rules yields a frame for $\hat{H}_{j+1}$ containing column expressions $f_{j+1}^1(h_j^1, \ldots, h_j^n)$, $\ldots$, $f_{j+1}^n(h_j^1, \ldots, h_j^n)$. The idea is now, that iteration $(*)$ can be performed by iterating the column expressions $h_{j+1}^k = f_{j+1}^k(\ldots)$.

Here, however, lies a problem. Many data types use *internal key types*, that is, types not visible in their external semantics. For instance, the frame $\mathit{bag}\langle d, r, F\rangle : \alpha \to \mathsf{Bag}A$ internally uses a key type β such that $r : [\alpha - \beta]$ and $F : \beta \to A$.

Rewrite rules often change the internal types used in such frames, for instance through th *mkprod*() and *mksum*() operators (Section 3.2). Therefore, it is possible that the type of $f_{j+1}()$ is incompatible with the type of h_j, making it impossible to perform the iteration

$$(h^1, \ldots, h^n) \mapsto \big(f^1(\ldots), f^n(\ldots)\big)$$

without deriving new expression for the f^k at every iteration. Obviously, this is not a desirable situation.

To work around this, we introduce a new required frame operation, *canonical_type*. The function of *canonical_type* is to "reset" all internal types to a types for which the internal representation is known to be compatible. An obvious choice would be subsets of integers. The *canonical_type* operator can then be applied to the *initial* and *step* members of the *iter*⟨⟩ frame, ensuring that the types used in the column expressions will be compatible and can be used for repeated evaluation.

6.6 Summary

In this chapter we have examined how general inductively defined types can be stored in Dodo. This poses significant problems, because the Dodo system as described in Chapters 2 and 3 assumes that the query is translated to a fixed series of flattened column operators, independent of the actual data in the database. When executing queries over recursive data, the number of items and inter-item dependencies inherently depends on the data in the database, making the fixed sequence of operations infeasible. As a solution, we introduce an iteration construct in the column layer, where a given series of column operations is iterated until a condition holds. This is sufficient to deal with the data-dependence of the query evaluation process.

To compile queries over inductively defined types (catamorphisms) into such iterations, we first derive from the categorical properties of the frame construct a fixpoint equation which the result frame should satisfy. This equation is transformed in an iteration scheme for frames, out of which a column level iteration of column operators can be derived.

Adding inductively defined datatypes to Dodo is a fundamental addition. Due to the breaking of the aforementioned assumption, it is not possible to add inductive data types purely as a Dodo *extension*. Changes to the framework itself are required. In particular, in the enhanced Dodo framework, extensions are now also required to provide

- for every new frame type $f\langle\rangle$, a *canonical_type* operator (paragraph 165),
- and for every higher-order function Σ, an instance of E_Σ (paragraph 155).

Chapter 7

Summary and future work

The theory developed in this thesis provides a method to improve the efficiency of querying nested data.

166 Background. The roots of this research lie in the tension between data model expressiveness and performance. Obviously, more expressive data models are more convenient for application programmers. For many non-traditional database applications, such as GIS, IR and multimedia applications, a more expressive data model is desirable. Building an expressive system with good performance turns out to be a hard problem. Common pitfalls are:

- Modern hardware requires simple and predictable memory access patterns and predictable branches to perform well; query evaluation on complex data models tends to result in the opposite.
- Complex data models are hard to understand for a query optimizer, making it hard to transform the query into an effective execution plan.
- With extensible databases, the above problem gets worse. If a query optimizer cannot "look inside" the extensions, it misses opportunities to improve the query plan. Optimizations which cut across more than one extension are even harder.

One promising solution is the *extensible abstract datatype* (E-ADT) approach, where the database consists of a framework into which E-ADTs are plugged. E-ADTs extend the database with new functionality and operations, and also offer an interface to the query optimizer. However, cross-extension optimization is still hard, as is cross-extension code reuse, because E-ADTs

are built right on top of the storage manager and cannot access each others functionality.

The multi-model database architecture, proposed by de Vries as a way to enable cross-extension interaction, distinguishes several layers, each of which offers another data model. Data is stored decomposed over simple bulk structures, and queries are gradually transformed down through the layers before execution. The multi-model architecture follows the open implementation approach, where components (layers) in the system are not black boxes but offer a meta-interface through which advise can be given in the choice of algorithms. Thus, the system is extensible at every layer, and extensions at higher layers can mix and match functionality provided at lower layers, even by other extensions. This facilitates cross-extension optimization and code reuse.

167 Problems. In our work we encountered, and solved, several problems:

Generality The problem is to make the theory sufficiently general to apply to a wide range of level data models and query languages.

We turned to category theory (Chapter 5) for inspiration. The categorical theory of data types [BW90, MFP91, Fok92] is a general, expressive, but concise language for reasoning about the properties of data types. Monads, for instance, provide us with a flexible mechanism for implementing comprehensions. Moreover, we found a way to deal with inductively defined types in general, executing recursive queries in bulk on the underlying platform (Chapter 6).

Extensibility The problem is to build an extensible system in which the boundaries between independently developed extensions do not hinder global optimization too much.

We applied the multi-model approach outlined in Section 1.3, thus following ideas put forward by Kiczales [KB96] and de Vries [dV99]. The categorical data model of the upper layer provides generic ways to combine existing data types. Alternatively, new type formers can be added by defining them in terms of relational structures provided by extensions on lower layers (Chapter 2 and 3). In our layered data model, we have a flat binary relational model in the lower layers. The binary relational model fits our function based nested model well, and is representative of the array-like data structures often used as the physical data model in efficient systems.

Bulk orientation The problem is to avoid the use of item-at-a-time processing, even when the shape of the original query suggests a nested loop query plan.

We introduced bulk orientation in queries over the nested data model, which are typically item oriented, by expressing the query in point-free form. This is again a categorical influence. Expressing the query in a form which does not involve variables is a natural step up to bulk operations. However, the elimination of nested scopes (Section 2.5.3) is not trivial.

Not only the query, also the data is expressed in a point-free style. Expressing both queries and data as compositions of functions allows us to freely mix parts of the query with data. By repeatedly replacing operations on nested data by operations working on the flattened representation of that data, the query is gradually rewritten to a form where it consists only of bulk operators applied to flattened data.

The pattern of replacing item oriented nested loop style query plans by bulk operations which evaluate many instances of an expression *in parallel* is also clearly recognizable in other systems. Pathfinder (Chapter 4) is an example of a system where a similar approach is applied with great success to XQuery processing. We show that Pathfinder can accurately be modelled as a Dodo extension plugged into the generic Dodo rewrite rules.

168 Validation. Regarding validation of our theory we mention the following:

Realization We have built a working prototype called Dodo (Chapter 3). In addition to the theory put forward in Chapter 2, we had to define a suitable column algebra. At the type level, there is a natural “impedance mismatch” between the function oriented upper layer and the relation oriented lower layer. Our type system had to be carefully designed to take this into account.

Applicability We have shown (Chapter 4) that the core ideas in Pathfinder are instantiations of our more general theory (Chapter 2). This holds in particular for *loop lifting*. The staircase join is an example of an extension which does not *follow* from our theory, but whose elaboration benefits from the structure provided by the theory.

169 Future work. We are quite satisfied with the results obtained so far, but realize that more work needs to be done in order to achieve a theory that can be applied without surprise problems. This includes the following issues:

Validation A more detailed exploration of a Pathfinder like system based on Dodo could provide insight in how close to a well performing system one can get by straightforward application of the Dodo design pattern. In

particular, whether use of the renumbering operation $r*$ on external node identifiers defined in paragraph 113 is really needed, or whether it only exists to satisfy Dodo's formal rules and can be avoided in practice. This could be characterized as "deeper validation."

We are also interested in "broader validation," applying Dodo to a wider range on application domains, such as GIS and IR, and also other complex data models. For the latter, one could think of an object-based data model, array models [vBdVK03] or alternative data models for XML, such as $\mathbb{C}$*Duce* [BCF03].

Recursion The treatment of the flattening of recursive queries in Chapter 6 only scratches the surface of this topic. The appearance of an E_Σ operator which turns out to be dual to the distribution function D_Σ used in Section 2.5.3 suggests that the way we solve the fixpoint equation by iteration is less ad-hoc than it may seem at first sight. Further study is required here.

System building A Dodo extension consists of frame definitions, declarations of operations, lots of rewrite rules, and generally also extensions at the column level, in the form new primitive data types and column operations. The Dodo framework specifies relations between all these elements, but in practice, it may be hard for the extension writer to keep track it all. In paragraph 90 we described several approaches for making this easier.

In a real-world system, managing extensions also requires a well-designed module system. In the Dodo setting, this is particularly complicated because extensions can extend the system at every layer, and the layers are each of a very different nature. On this topic, too, more research is required.

Error handling In this thesis, we transform a query from nested to flat, evaluate it, and transform the result back from flat to nested. An important question not discussed here is what to do when run-time errors occur. Generating understandable error messages is not a well understood topic anyway, but the presence of loop lifted operators makes it even more complex. It would be interesting to extend the theory of Dodo with means to unflatten not only query results but also error messages, pinpointing accurately which part of query and data caused the error.

Column algebra optimization For the sake of simplicity and generality, the Dodo rewrite rules often emit redundant query plans, in the expectation that subsequent phases eliminate common subexpressions and remove unnecessary column operations in a peep hole optimization phase. The Dodo

prototype contains several ad-hoc peep hole optimization rules to keep the query plans readable and manageable, but there is much work to be done. In Section 3.4 we sketched how better static analysis of column types could benefit from this kind of optimization.

We look forward to seeing Dodo come alive.

Bibliography

[BCF03] V. Benzaken, G. Castagna, and A. Frisch, *CDuce: An XML-centric general-purpose language*, Proceedings of the ACM International Conference on Functional Programming, August 2003, pp. 51–64.

[BCF+05] Scott Boag, Donald D. Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon, *XQuery 1.0: An XML query language*, World Wide Web Consortium, Candidate Recommendation CR-xquery-20051103, November 2005.

[BDK92] François Bancilhon, Claude Delobel, and Paris C. Kanellakis (eds.), *Building an Object-Oriented Database System, The Story of O2*, Morgan Kaufmann, 1992.

[BdVBA01] Henk Ernst Blok, Arjen P. de Vries, Henk M. Blanken, and Peter M.G. Apers, *Experiences with IR Top N Optimization in a Main Memory DBMS: Applying 'The Database Approach' in New Domains*, Proceedings of BNCOD 18, Advances in Databases (Brian Read, ed.), Lecture Notes in Computer Science, vol. 2097, Springer, jul 2001, pp. 126–151.

[BGvK+06] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner, *Fast and Scalable XQuery Processing Using A Purely Relational Approach*, ACM SIGMOD International Conference on Management of Data, Chicago, USA, June 26-29, 2006, June 2006.

[BHC+01] Henk Ernst Blok, Djoerd Hiemstra, Sunil Choenni, Franciska de Jong, Henk M. Blanken, and Peter M. G. Apers, *Predicting the cost-quality trade-off for information retrieval queries: Facilitating database design and query optimization.*, CIKM, ACM, 2001, pp. 207–214.

[Bir87] R.S. Bird, *An introduction to the theory of lists*, Logic of Programming and Calculi of Discrete Design (M. Broy, ed.), Springer, 1987, Also Technical Monograph PRG–56, Oxford University, Computing Laboratory, Programming Research Group, October 1986, pp. 3–42.

[BK99] Peter A. Boncz and Martin L. Kersten, *MIL Primitives for Querying a Fragmented World*, VLDB J. **8** (1999), no. 2, 101–119.

[BM96] R.S. Bird and O. de Moor, *Algebra of programming*, Prentice-Hall International, 1996.

[Bon02] Peter A. Boncz, *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*, phdthesis, University of Amsterdam, 2002.

[BQK96] P. A. Boncz, W. Quak, and M. L. Kersten, *Monet and its Geographical Extensions: a Novel Approach to High-Performance GIS Processing*, Proceedings of the International Conference on Extending Database Technology (EDBT) (Avignon, France), Lecture Notes in Computer Science/Lecture Notes in Artificial Intelligence (LNCS/LNAI), Springer-Verlag, vol. 1057, June 1996, pp. 147–166.

[BW90] M. Barr and C. Wells, *Category theory for computing science*, Prentice Hall, 1990.

[BWK98] P. A. Boncz, A. N. Wilschut, and M. L. Kersten, *Flattening an Object Algebra to Provide Performance*, Proceedings of the IEEE International Conference on Data Engineering (ICDE) (Orlando, FL, USA), February 1998, pp. 568–577.

[BZN05] Peter A. Boncz, Marcin Zukowski, and Niels Nes, *MonetDB/X100: Hyper-pipelining query execution*, CIDR, 2005, pp. 225–237.

[CAB+81] Donald D. Chamberlin, Morton M. Astrahan, Mike W. Blasgen, Jim Gray, W. Frank King III, Bruce G. Lindsay, Raymond A. Lorie, James W. Mehl, Thomas G. Price, Gianfranco R. Putzolu, Patricia G. Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost, *A history and evaluation of system r*, Commun. ACM **24** (1981), no. 10, 632–646.

[CHS+95] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, M. Flickner, A. W. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J. H. Williams, and E. L. Wimmers, *Towards heterogeneous multimedia information systems: The garlic approach*, Research Issues in Data Engineering (RIDE '95) (Los Alamitos, Ca., USA), IEEE Computer Society Press, March 1995, pp. 124–131.

[Clu98] Sophie Cluet, *Designing OQL: Allowing objects to be queried*, Information Systems (1998), 279–305.

[Cod70] E. F. Codd, *A relational model of data for large shared data banks*, Comm. ACM **13** (1970), no. 6, 377–387.

[dV99] A.P. de Vries, *Content and multimedia database management systems*, phdthesis, University of Twente, Enschede, The Netherlands, dec 1999.

[dVEK98] Arjen P. de Vries, Brian S. Eberman, and David E. Kovalcin, *The design and implementation of an infrastructure for multimedia digital libraries*, IDEAS, 1998, pp. 103–120.

[dVLB03] A.P. de Vries, J.A. List, and H.E. Blok, *The Multi-model DBMS Architecture and XML IR*, Intelligent Search on XML Data (H.M. Blanken, T. Grabs, H.-J. Schek, R. Schenkel, and G. Weikum, eds.), LNCS, no. 2818, Springer-Verlag, 2003, pp. 179–191.

[dVvDBA99] Arjen P. de Vries, Mark G. L. M. van Doorn, Henk M. Blanken, and Peter M. G. Apers, *The Mirror MMDBMS Architecture*, VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK (Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, eds.), Morgan Kaufmann, 1999, pp. 758–761.

[FMM+05] Mary F. Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh, *XQuery 1.0 and XPath 2.0 data model (XDM)*, World Wide Web Consortium, Candidate Recommendation CR-xpath-datamodel-20051103, November 2005.

[Fok92] Maarten M. Fokkinga, *Law and Order in Algorithmics*, phdthesis, University of Twente, 1992.

[Gru99] Torsten Grust, *Comprehending Queries*, phdthesis, University of Konstanz, sep 1999.

[Gru02] Torsten Grust, *Accelerating xpath location steps*, SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data (New York, NY, USA), ACM Press, 2002, pp. 109–120.

[GST04] Torsten Grust, Sherif Sakr, and Jens Teubner, *XQuery on SQL Hosts*, Proceedings of the 30th Int'l Conference on Very Large Databases (VLDB 2004) (Toronto, Canada), August 2004.

[GT04] Torsten Grust and Jens Teubner, *Relational algebra: Mother tongue—xquery: Fluent*, Proceedings of the first Twente Data Management Workshop on XML Databases, 2004.

[GvKT03] T. Grust, M. van Keulen, and J. Teubner, *Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps*, Proceedings of the 29th International Conference on Very Large Databases, VLDB'2003, Berlin, Germany (J.-C. Freytag, P.C. Lockemann, S. Abiteboul, M. Carey, P. Selinger, and A. Heuer, eds.), Morgan Kaufmann Publishers, sep 2003, pp. 524–535.

[GvKT04] Torsten Grust, Maurice van Keulen, and Jens Teubner, *Accelerating XPath Evaluation in Any RDBMS*, Transactions on Database Systems (TODS) **29** (2004), no. 1, 91–131.

[KB96] Gregor Kiczales and : Beyond the Black Box: Open Implementation. IEEE Software, *Beyond the Black Box: Open Implementation*, IEEE Software **13** (1996), no. 1, 8–11.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson, *Functional programming with bananas, lenses, envelopes and barbed wire*, Proceedings 5th ACM Conf. on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA, 26–30 Aug 1991 (J. Hughes, ed.), vol. 523, Springer-Verlag, Berlin, 1991, pp. 124–144.

[MGvKT04] S. Mayer, T. Grust, M. van Keulen, and J. Teubner, *An Injection with Tree Awareness: Adding Staircase Join to PostgreSQL*, Proceedings of the 30th International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004, Morgan Kaufmann, August 2004, pp. 1305–1308.

[Pie02] Benjamin C. Pierce, *Types and programming languages*, MIT Press, 2002.

[PJ87] Simon L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall, 1987.

[PTM+05] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner, *Pathfinder: XQuery – The Relational Way*, Proceedingsof the 31th Int'l Conference on Very Large Databases (VLDB 2005) (Trondheim, Norway), August 2005.

[PTS+05] Peter Boncz, Torsten Grust, Stefan Manegold, Jan Rittinger, and Jens Teubner, *Pathfinder: Relational XQuery Over Multi-Gigabyte XML Inputs In Interactive Time*, Tech. Report INS-E0503, CWI, Amsterdam, March 2005.

[RFK04] Joeri van Ruth, Maarten Fokkinga, and Maurice van Keulen, *The Dodo Query Flattening System*, techreport TR-CTIT-04-41, Centre for Telematics and Information Technology, University of Twente, The Netherlands, sep 2004.

[SABdB94] Hennie J. Steenhagen, Peter M. G. Apers, Henk M. Blanken, and Rolf A. de By, *From Nested-Loop to Join Queries in OODB*, VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile (Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, eds.), Morgan Kaufmann, 1994, pp. 618–629.

[Ses98] Praveen Seshadri, *PREDATOR: A resource for database research*, j-SIGMOD **27** (1998), no. 1, 16–20.

[Sto93] Michael Stonebraker, *The SEQUOIA 2000 project*, IEEE Data Eng. Bull. **16** (1993), no. 1, 24–28.

[TK78] D. Tsichritzis and A. C. Klug, *The ANSI/X3/SPARC DBMS framework report of the study group on database management systems*, Information Systems **3** (1978), no. 3, 173–191.

[Tur79] D.A. Turner, *A new implementation technique for applicative languages.*, Softw., Pract. Exper. **9** (1979), no. 1, 31–49.

[vBdVK03] Alex R. van Ballegooij, Arjen P. de Vries, and Martin L. Kersten, *RAM: Array processing over a relational DBMS*, techreport, CWI, 2003.

[vKVdV+03] 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*, Proceedings of the 14th International Conference on Database and Expert Systems Applications (DEXA2003), Prague, Czech Republic (V. Marik, W. Retschitzegger, and O. Stepankova, eds.), LNCS, no. 2736, Springer-Verlag, sep 2003, pp. 67–76.

[vR05] Joeri van Ruth, *A general approach to query flattening*, Proceedings VLDB PhD workshop 2005 (Trondheim, Norway), August 2005.

Samenvatting

Het centrale thema van dit proefschrift is *query flattening.* Dit "platslaan" van zoekopdrachten komt voort uit het spanningsveld tussen de behoefte aan expressieve, geneste datastructuren aan de kant van de applicatieprogrammeur, en de noodzaak van eenvoudige, ongeneste (platte) datastructuren voor efficiente verwerking op moderne computerhardware. Gezien huidige ontwikkelingen op het gebied van computerarchitectuur wordt namelijk steeds belangrijker om gebruik te maken van bulkoperaties op simpele datastructuren.

In dit proefschrift ligt de nadruk op het verwerken van reken- en data-intensieve queries en speelt het invoeren en bijwerken van gegevens (updates) een ondergeschikte rol. Dit type queries komt veel voor bij bijvoorbeeld geografische toepassingen (GIS), information retrieval (IR) en multimedia toepassingen.

Vanwege het grote volume van de data ligt het voor de hand om de gegevens op te slaan in een database management system (DBMS). In het verleden is het echter moeilijk gebleken om dit soort toepassingen te integreren met "klassieke" relationele databasetechnologie. Het is vanwege de complexere structuur van de gegevens vaak lastig en tijdrovend om met de hand de gegevens op te breken in allerlei tabellen. Verder zijn bestaande databases vaak meer toegespitst op administratieve toepassingen, met meestal relatief eenvoudige queries en veel updates. Voor niet-traditionele toepassingen valt de performance daarom vaak tegen.

Een veelgekozen oplossing is om de complexe gegevens dan maar in een apart systeem naast de bestaande database op te slaan. Dit is geen bevredigende oplossing omdat het dan moeilijk is om vragen te beantwoorden die betrekking hebben op gegevens die over beide systemen verspreid opgeslagen liggen. Daarnaast valt op dat in veel verschillende gespecialiseerde systemen, dezelfde technieken terugkomen voor het efficienter maken van query verwerking. Een belangrijk voorbeeld daarvan is *query flattening.* De achterliggende visie van het onderzoek waar dit proefschrift deel van uitmaakt, is dan ook om te proberen de bestaande databasetheorie zodanig uit te breiden dat het niet meer nodig is om aparte systeem te bouwen voor complexe datastructuren.

De basis van onze benadering is de zogenaamde *Multi-model architectuur*, waarin de verwerking van queries plaatsvindt in een aantal lagen, en de query stap voor stap wordt vertaald van complexe datastructuren in de bovenste laag naar simpele, platte datastructuren in de onderste laag. Op zichzelf is het idee van een meerlaags model niet nieuw. Het interessante aan de multi-model architectuur is dat elke laag op zich uitbreidbaar is, waarbij een uitbreiding op hogere niveaus toegang heeft tot alle functionaliteit op de lagere niveaus. Bij het bouwen en verwerken van complexe structuren op het hoge niveau wordt dan gebruik gemaakt van bestaande functionaliteit op het lage niveau, plus enkele applicatie-specifieke uitbreidingen op het lage niveau.

Door de decompositie van complexe structuren in efficiente bulkstructuren niet over te laten aan een generieke standaardmethode, maar in de handen van de uitbreidingsprogrammeur te leggen, is het mogelijk gebleken om met relationele databases zeer efficient complexe datastructuren te verwerken. Een voorbeeld hiervan is te vinden in hoofdstuk 4 van dit proefschrift, waar we schetsen hoe de zeer efficiente *MonetDB-XQuery* database geneste XML structuren afbeeldt op platte tabelstructuren, en hoe deze afbeelding opgevat kan worden als een specialisatie van het in dit proefschrift beschreven *Dodo* query flattening framework.

In dit proefschrift beschrijven we een algemeen framework voor het vertalen van queries van een datamodel met geneste data types naar een datamodel met alleen platte tabelstructuren. De achterliggende gedachte is, dat de data plat opgeslagen ligt, maar dat queries geformuleerd kunnen worden in termen van geneste data typen. Het Dodo framework schrijft in principe niet voor hoe complexe structuren in tabelvorm moeten worden gebracht. Dat blijft een creatief ontwerpproces waarbij Dodo slechts helpt de gedachten te bepalen. Wel geeft Dodo een universele taal waarin het *flattening* proces kan worden uitgedrukt. Bovendien geeft Dodo regels die helpen de correcte relatie te leggen tussen de operaties op platte data "onderin" de database, en operatie op geneste data "bovenin" de database.

Het centrale concept in Dodo is de zogenaamde *punt-vrije vorm*. Dat is een notatie waarbij de query volledig wordt uitgedrukt zonder gebruik te maken van variabelen. De eerste stap in het verwerken van een query is het omschrijven van de query naar punt-vrije vorm. Dit gebeurt aan de hand van regels die specifiek zijn voor de gebruikte query taal. Behalve query wordt ook de in de database aanwezige data in een punt-vrije vorm genoteerd. Hoe dit gebeurt, is door de uitbreidingsprogrammeur vastgelegd bij het definieren van de uitbreiding.

De uitbreidingsprogrammeur definieert operaties op genestte data in punt-vrije vorm. Omdat er in punt-vrije vorm geen variabelen zijn, kunnen operaties niet verwijzen naar individuele data elementen, en daarmee is de punt-vrije vorm een geschikte eerste stap naar bulk orientatie: de definitie van een operatie

beschrijft hoe je de operatie uitvoert op een hele rij data elementen tegelijk.

Op het moment dat we zowel de query als de data in punt-vrije vorm hebben, kunnen we ze met elkaar gaan mengen. De punt-vrije herschrijfregels transformeren geneste operaties naar bulkoperaties op het platte niveau, die efficient kunnen worden uitgevoerd door de onderliggende hardware. De bijdrage van het Dodo-framework is dat de programmeur niet meer na hoeft te denken over het elimineren van nesting uit de query. Het enige dat aan de programmeur gevraagd wordt, is om voor elke operatie de concrete vraag te beantwoorden: hoe voer je operatie uit op punt-vrij opgeslagen data, gebruikmakend van specifieke kenmerken en algoritmen. Dit alles wordt nader uitgewerkt in Hoofdstukken 2 en 3.

De theorie zoals hierboven beschreven heeft een belangrijke beperking: hij is beperkt tot datastructuren die genest zijn qua type, bijvoorbeeld verzamelingen van tupels van polygonen die elk bestaan uit lijnstukken die opgebouwd zijn uit punten. Veel data heeft echter ook een recursieve structuur waarvan de diepte niet door het type wordt vastgelegd. Denk daarbij bijvoorbeeld aan boomstructuren, waarbij elke node in de boom verwijzingen kan hebben naar ander nodes.

In Hoofdstuk 6 onderzoeken we een manier om deze restrictie te verlichten. We geven een mechanisme waarmee Dodo automatisch een serie platte operaties kan afleiden voor een belangrijke klasse boom-operators, de *catamorfismen*. Veel bekende recursieve operaties, bijvoorbeeld de *sum* functie op lijsten, kunnen worden uitgedrukt als een catamorfisme. Hoewel een door de uitbreidingsprogrammeur geschreven implementatie over het algemeen efficienter zal blijven, maken de door Dodo gegenereerde query plannen goed gebruik maken van bulkoperaties op het platte niveau, vooral als de boomstructuren ondiep maar breed zijn.

Het in dit proefschrift beschreven Dodo framework valideren we op verschillende manieren. We kijken hoe queries vertaald worden door een prototype. We demonstreren hoe de essentiele kenmerken van MonetDB-XQuery overeenkomen met de kernpunten van de Dodo benadering. Tenslotte bouwen we onze benadering op een krachtige theoretische basis, de categorische theorie van datatypen.

SIKS Dissertation Series

[2005-01] Floor Verdenius (UVA), *Methodological Aspects of Designing Induction-Based Applications*

[2005-02] Erik van der Werf (UM), *AI techniques for the game of Go*

[2005-03] Franc Grootjen (RUN), *A Pragmatic Approach to the Conceptualisation of Language*

[2005-04] Nirvana Meratnia (UT), *Towards Database Support for Moving Object data*

[2005-05] Gabriel Infante-Lopez (UVA), *Two-Level Probabilistic Grammars for Natural Language Parsing*

[2005-06] Pieter Spronck (UM), *Adaptive Game AI*

[2005-07] Flavius Frasincar (TUE), *Hypermedia Presentation Generation for Semantic Web Information Systems*

[2005-08] Richard Vdovjak (TUE), *A Model-driven Approach for Building Distributed Ontology-based Web Applications*

[2005-09] Jeen Broekstra (VU), *Storage, Querying and Inferencing for Semantic Web Languages*

[2005-10] Anders Bouwer (UVA), *Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments*

[2005-11] Elth Ogston (VU), *Agent Based Matchmaking and Clustering - A Decentralized Approach to Search*

[2005-12] Csaba Boer (EUR), *Distributed Simulation in Industry*

[2005-13] Fred Hamburg (UL), *Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen*

[2005-14] Borys Omelayenko (VU), *Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics*

[2005-15] Tibor Bosse (VU), *Analysis of the Dynamics of Cognitive Processes*

[2005-16] Joris Graaumans (UU), *Usability of XML Query Languages*

[2005-17] Boris Shishkov (TUD), *Software Specification Based on Re-usable Business Components*

[2005-18] Danielle Sent (UU), *Test-selection strategies for probabilistic networks*

[2005-19] Michel van Dartel (UM), *Situated Representation*

[2005-20] Cristina Coteanu (UL), *Cyber Consumer Law, State of the Art and Perspectives*

[2005-21] Wijnand Derks (UT), *Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics*

[2006-01] Samuil Angelov (TUE), *Foundations of B2B Electronic Contracting*

[2006-02] Cristina Chisalita (VU), *Contextual issues in the design and use of information technology in organizations*

[2006-03] Noor Christoph (UVA), *The role of metacognitive skills in learning to solve problems*

[2006-04] Marta Sabou (VU), *Building Web Service Ontologies*

[2006-05] Cees Pierik (UU), *Validation Techniques for Object-Oriented Proof Outlines*

[2006-06] Ziv Baida (VU), *Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling*

[2006-07] Marko Smiljanic (UT), *XML schema matching – balancing efficiency and effectiveness by means of clustering*

[2006-08] Eelco Herder (UT), *Forward, Back and Home Again - Analyzing User Behavior on the Web*

[2006-09] Mohamed Wahdan (UM), *Automatic Formulation of the Auditor's Opinion*

[2006-10] Ronny Siebes (VU), *Semantic Routing in Peer-to-Peer Systems*