The Extended MdF model

Ulrik Petersen

November 30, 1999

Abstract

The MdF text database model was developed by Crist-Jan Doedens in his 1994 PhD dissertation. It is a general model for storing *expounded text*, i.e., text coupled with information about that text. The MdF model is extremely powerful, is clean, simple, intuitive, and elegant, and has a number of features which other contemporary text database models, e.g., SGML, do not have.

In this Bachelor's thesis, we first describe the standard MdF model in quite a bit of detail. We then go on to detailing the Extended MdF model (EMdF). The EMdF model has the advantage of being closer to an implementation of the concepts set forth in the MdF model than the MdF model itself. We then give some hints on how to implement the EMdF model in an actual database. Finally, we develop, motivate, and explain MQL, a query language which can be used to query EMdF databases. The exposition on the MQL language constitutes the bulk of the thesis.

Contents

Chapter 1

Introduction

1.1 Introduction

This chapter introduces this bachelor thesis by describing the topics of each of the chapters. We also give some conventions used in the thesis.

1.2 The standard MdF model

The standard MdF model was developed by Crist-Jan Doedens in his 1994 PhD dissertation. It is a database model for text databases. In this chapter, we introduce the standard MdF model as detailed in Doedens' book. Specifically, we explain Doedens' thirteen demands on text databases, and give definitions of the concepts of the MdF model. These concepts include "monad", "object", "object type", and "feature". We also introduce a lot of useful concepts which grow naturally out of these four concepts.

1.3 The Extended MdF model

The Extended MdF model (EMdF) is my attempt to make the MdF model less abstract and thus more implementable. In this chapter, we define and name a lot of concepts which are useful when talking about EMdF databases and their implementation. The main contribution is to distinguish between concepts "in the abstract" and concepts "in the implementation". We also define a lot of new concepts which are useful when implementing an EMdF database.

1.4 Implementing the EMdF model

In this chapter, we give a lot of hints on how to implement an EMdF database engine, specifically with a view towards also implementing an MQL query engine. The three main sections of the chapter involve the all-important concept of a set of monads, various specific classes of objects that need to be stored in an EMdF database (not just EMdF objects, but also housekeeping objects), and four algorithms for the insertion and deletion of monads and objects.

1.5 MQL - Mini QL

This chapter represents both the bulk of the work done for the thesis and the bulk of the thesis itself. MQL is a stripped-down version of the QL query language defined in Doedens' work. Thus MQL builds on the concepts set forth in Doedens' QL. As a totally new contribution, I have furnished MQL with a semantics which is given in the form of an operational, syntax-driven specification in a high-level language. The chapter discusses the sheaf, which is the datastructure which holds a query result of an MQL query. It also discusses the syntax of MQL and the constraints upon the syntax. MQL variables are discussed at some length, and many good reasons are given why MQL should be compiled into an Abstract Syntax Tree. A sketch of an architecture for an MQL engine is given. Last, but definitely not least, we give the semantics of MQL in the form of retrieval functions and their explanation.

1.6 Conventions

Standard conventions for mathematical notation is used throughout this thesis. In addition, some thesis-specific conventions are used:

- Literals are given in `typewriter font`. E.g., `first`.
- Syntactic units are also given in `typewriter font`. E.g., `firstlast`.
- Code and code-variables are also given in `typewriter font`. E.g., `begin`, `Sm`.

Chapter 2

The standard MdF model

2.1 Introduction

The MdF model was developed by Crist-Jan Doedens in his 1994 PhD dissertation. It is a database model which is exceptionally well suited to storing linguistic analyses of text. The MdF model gives a high-level view of text databases, where a text database is viewed as text plus information about that text. The MdF model is mathematically clean, simple, intuitive, and elegant, which makes it well suited to conceptualization of solutions to problems which can be solved by a text database.

This chapter gives an introduction to the standard MdF model. The introduction is based heavily on Chapters 2 and 3 in Doedens' book, and follows the same structure.

The bibliographic information for Doedens' PhD dissertation is:

> Doedens, Crist-Jan [Christianus Franciscus Joannes]. *Text Databases. One Database Model and Several Retrieval Languages.* Language and Computers, Number 14. Amsterdam and Atlanta, GA: Editions Rodopi Amsterdam, 1994. Extent: xii + 314 pages. ISBN: 90-5183-729-1.

2.2 On text-databases generally

2.2.1 Text-dominated databases, expounded text

Two concepts are a key in understanding the state of the art in text databases. I here give two quotes from Doedens:

Text-dominated databases are "collections of data, predominantly composed of characters, in which we can perceive structure" (p. 18). This is the current viewpoint on databases of text.

Expounded text is "An interpreted text, i.e. a combination of text and information about this text, stored in a computer, and structured for easy update and access" (p. 19). The MdF model embodies the idea of expounded text.

2.2.2 What is a database model?

Doedens defines a **data model** or **database model** as

> "a toolbox of concepts which can be used to describe the handling by the computer of data in certain domain(s). The concepts should allow easy formulation by humans of the structuring and handling of the data in the domain(s). The concepts can be grouped as follows:
>
> - The data structures supported by the model
> - The access language. This language, or set of languages should allow the definition of the structure and types of the data and allow creation, insertion, change, deletion and retrieval of the data." (p. 23)

The MdF model is not a full database model, in that it does not specify an access language, but only the data structures supported by the model. This, a database model without its access language component, is what Doedens calls a **static database model**.

2.2.3 Demands on a text database model

Doedens says:

> "The fundamental requirement for a text database model is that it should be able to support the structural description of a text and its associated annotations." (p. 25)

He then lists thirteen demands which he percieves should be set on text database models:

D1. Objects: We should be able to identify separate parts of the text.

This can be realized with instances of the concept of objects.

D2. Objects are unique: Each object should be uniquely identifiable.

Otherwise, we may not know what we are talking about.

D3. Objects are independent: Each object in the database should exist without direct reference to other objects.

The advantage of having this is that we can have the best of two worlds: Independence and dependence, isolation and referentiality. The independence comes from D3 being met, and the dependence can arise through D4-D7 being met.

D4. Object types: We need 'object types': we should be able to assign the same generic name to like objects.

For example, we would like to be able to identify certain parts of a book stored in a text database as "paragraphs", other parts of a book as "chapters", and other parts as "pages".

D5. Multiple hierarchies: It should be possible to have different hierarchies of types.

For example, we might have a hierarchy of types which form a textual hierarchy ('character', 'word', 'line', 'page', 'book'), and a hierarchy of types which form a logical hierarchy ('word', 'sentence', 'paragraph', 'chapter').

D6. Hierarchies can share types:

See D5 for a useful example.

D7. Object features: We should be able to assign features and values for these features to objects.

D8. Acommodation for variations in the surface text: E.g. variations in spelling should be attributable to the same words in the surface text.

D9. Overlapping objects: We need objects of the same type to be able to overlap.

For example, to describe recursivity in linguistic phenomena. Take for example the string of words, 'A noun phrase and a conjunction phrase': How do we analyze this? As two noun phrases ("A noun phrase" and "a conjunction phrase") conjoined by a conjunction ("and"), or as one compound noun phrase (the whole thing)? We want to be able to represent both choices. Overlapping objects of the same type can help us in doing this.

D10. Gaps: We need objects with 'gaps'.

For example to describe clauses which are discontiguous, as in "John, who was having a cow, freaked out.", where "John ... freaked out" is a discontiguous clause.

So far, the demands have dealt with the data-structures to support the model. The following demands reflect the demands which a full model (one in which there is also an access language) must meet:

D11. Type language: We need a type language in which we can specify object types and the types of their features.

D12. Data language: We need a strongly typed data language in which we can specify the creation, insertion, change, deletion, and retrieval of data.

D13. Structural relations between types: It should be possible to specify standard structural relations between objects of different types.

None of the text database models available today satisfies these 13 demands. The MdF model satisfies D1-D10. In contrast, SGML does not, and neither do any of its derivatives, e.g.. XML.

2.3 Gentle introduction to the MdF model

2.3.1 Key concepts

The MdF model has four key concepts:

1. Monads: These are the basic building blocks of the database. They are simply integers. Monads are ordered relative to each other, such that a string of monads emerges.

2. Objects: Objects are made of monads. An object is a set of monads.

3. Object types: Objects are grouped in types. An object type determines what features an object has.

4. Features: A feature is a function on objects. A feature takes an object as its argument and returns some value usually based on that object.

2.3.2 An example

An example of an MdF database can be seen in figure 2.1 on page 14. It has five object types: Word, Phrase, Clause_atom, Clause, and Sentence. Object type Word has two features: surface, and part_of_speech. Object type Phrase has one feature: phrase_type. Object types Clause_atom, Clause, and Sentence have no features. The first Phrase object consists of the set of monads $\{1, 2\}$, the third of the set of monads $\{4\}$, and the fourth of the set of monads $\{5, 6, 7\}$. The first Clause object consists of the monads

<table>
<tr><td></td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr>
<tr><td>Word</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr>
<tr><td>surface</td><td>The</td><td>door,</td><td>which</td><td>opened</td><td>towards</td><td>the</td><td>East,</td><td>was</td><td>blue.</td></tr>
<tr><td>part_of_speech</td><td>def.art.</td><td>noun</td><td>rel.pron.</td><td>verb</td><td>prep.</td><td>def.art.</td><td>noun</td><td>verb</td><td>adject.</td></tr>
<tr><td>Phrase</td><td colspan="2">1</td><td>2</td><td>3</td><td rowspan="2"></td><td colspan="2">5</td><td>6</td><td>7</td></tr>
<tr><td>phrase_type</td><td colspan="2">NP</td><td>NP</td><td>VP</td><td colspan="2">NP</td><td>VP</td><td>AP</td></tr>
<tr><td>Phrase</td><td colspan="4" rowspan="2"></td><td colspan="3">4</td><td colspan="2" rowspan="2"></td></tr>
<tr><td>phrase_type</td><td colspan="3">PP</td></tr>
<tr><td>Clause_atom</td><td colspan="2">1</td><td colspan="5">2</td><td colspan="2">3</td></tr>
<tr><td>Clause</td><td colspan="2">1</td><td colspan="5">2</td><td colspan="2">1</td></tr>
<tr><td>Sentence</td><td colspan="9">1</td></tr>
</table>

Figure 2.1: MdF example

$\{1, 2, 8, 9\}$. Note that the first Word object *consists of* the set of monads $\{1\}$, and *is* not monad 1, and likewise with the rest of the Word objects.

2.4 Monads

2.4.1 General

The MdF model was developed for text databases, but can be used for storing anything that is linear in nature, e.g., DNA sequences. The backbone of an MdF database is a linear string of minimal, indivisible elements, called monads. The precise nature of the entities which the monads represent is of no importance to the model. The only thing that matters is the relative ordering of the monads.

A monad is simply an integer. It represents the rank number in the string of monads, starting from 1. Since monads are integers, we can apply all the usual relational and arithmetic operators to them. For example, if we have monads a and b, we can test whether $a < b$, $a = b$, or $a > b$. Or we can test whether $a + 3 = b$.

2.4.2 Application of monads to text flows

In everyday thinking about text, any given text is conceptualized as a one-dimensional string. The text may be laid out on a two-dimensional medium,

- testament
- book
- chapter
- verse
- word
- phrase
- clause
- sentence

Table 2.1: Sample Object types

e.g., paper, but there is still only one string. The elements in this string have an ordering called the **reading order**. For example, the English reading-order is left-to-right, line-by-line, downwards. The Arabic reading-order is different, and the Japanese reading-order is different yet again. From the point of view of the MdF model, the reading-order is dictated by the monads.

The MdF model has no concept of parallel or unrelated text flows, e.g., footnotes or margin notes vs. the main text. Fortunately, this is not a problem, since the object types allow us to direct attention to any text flow at will. For example, the implementer of an MdF database might decide to intertwine the main text and the footnotes. The text is then called "footnote" or "main text" simply by defining appropriate object types and defining its objects appropriately. For an MdF database with several books, it might be more intuitive to place each book after the other, defining an object type called "book" and defining its objects appropriately.

2.5 Objects, object types

An MdF object is a set of monads. The monads in this set need not be contiguous. This is a great advantage, since it allows us to have objects with gaps.

Objects are grouped in types. For a sample list of object types which might occur in an MdF database, look at table 2.1.

It is an important characteristic of objects that no two objects of the same object type may consist of the same monads, i.e., there must not be two objects of the same object type for which there is set equality between the two sets of monads. The reason for this restriction is that it gives us a simple and clear criterion for what different objects are: An object is unique in its set of monads. On the other hand, two objects of different types may consist of the same monads. And two objects of the same type may share monads, so long as their sets of monads are not identical.

Note that objects do not consist of other objects. Instead, objects consist of monads. This is a great advantage, since it allows us to specify multiple hierarchies.

The fact that objects are sets of monads gives rise to a rich set of descriptive terms, which can all be formulated in terms of the basic operators on sets, such as set equality, the subset relation, set intersection, set union, and the "member of" relation. Also, the fact that there is an ordering, $<$, on the monads gives rise to a number of interesting properties. We shall return to these properties in section 2.7 when discussing concepts related to the MdF model.

2.6 Features

A feature is a function taking one argument: An object. The object type of an object determines what features the object has. The domain (type of argument) of a feature function is the set of objects of a given object type. The codomain (type of return value) of a feature function can be anything: The MdF model puts no restrictions on the codomain. This allows the implementor to implement anything at all which he or she might feel should be a feature. For a list of sample features which might be present in an MdF database, look at table 2.2.

In particular, the codomain of a feature can be another function taking other arguments, thereby in effect producing a feature with arguments.

Features can be partial functions, i.e., there can be objects for which a feature's value is not defined.

There is no such thing as a "genuine" feature. All features are considered to have equal status from the point of view of the MdF database. For example, the feature returning the surface of a word has the same status as a feature that returns the sum of the monads in its argument object.

2.7 Extending the basic framework

2.7.1 Introduction

This section is almost a duplicate of section 3.6 in Doedens' book, but leaving out some explanations, examples, justifications, and a few concepts which are irrelevant to the understanding of EMdF.

Throughout this section, we will refer to figure 2.1 on page 14.

Object type	Feature name	Feature function
book	book_name	maps a book object to its name
book	book_number	maps a book object to its number
chapter	chapter_number	maps a chapter to its number
word	lemma	maps a word to its lemma
word	Friberg_tag	maps a word to its Friberg grammar tag
word	part_of_speech	maps a word to its part of speech
word	case	maps a word to its case (if applicable)
word	gender	maps a word to its gender (if applicable)
word	number	maps a word to its number (if applicable)
phrase	phrase_type	maps a phrase to its type (e.g., VP, NP)
phrase	arthricity	maps a phrase to its status as being arthrous
phrase	function	maps a phrase to its function (e.g., Subj, Obj)
phrase	determinedness	maps a phrase to its status as determined or not
clause	clause_type	maps a clause to its clause type
clause	function	maps a clause to its function

Table 2.2: Sample features

2.7.2 Some special types: all_m, any_m, pow_m

Any MdF database implicitly defines the object types all_m, any_m, and pow_m. The reason for this is that it is convenient when talking about MdF databases.

All_m is the object type which has just one object: The one consisting of all monads in the database.

Any_m is the object type which has for each monad one object consisting of that monad.

Pow_m is the object type which has a member for each member of the powerset of the monads.

None of these three special object types has any application-specific features.

2.7.3 Linear ordering of objects per type

The linear ordering of objects per type is based on a 'lexicographic' ordering of the monads. This is done using the smallest monad as sort key. If this gives a tie, we go on to the next smallest, the next smallest, and so on. If at any point object O_2 has a monad which object O_1 does not, then object O_1 is stipulated to have the smaller ordinal of the two. For any object type T, the ordering relation within the type is denoted $<_T$. Thus, for example, if we have an object type T , then these relations hold:

$\{1\} <_T \{1,2\}$

$\{1,3\} <_T \{1,2\}$

$\{1,3,4,5\} <_T \{1,3,4,5,6\}$
$\{1,2,3,4,5,6,7\} <_T \{2\}$
$\{3,4\} <_T \{5,6\}$

2.7.4 Ordinal of an object, object id

The linear ordering of objects per type can be used to assign ordinals to objects of a given type. We just assign the ordinal 1 to the first object in the linear ordering, and go on from there.

This means that objects can be identified by their type plus their ordinal. Alterntively, since objects are unique in their monads, they can be identified by their type plus their monads. Both ways of identifying an object can be useful.

When identifying objects by their type plus their ordinal, the resulting id is called an **object id_o**. For example, objects of type T might be called T-1, T-2, T-45, etc.

When identifying objects by their type plus their monads, the resulting id is called an **object id_m**. For example, objects of type T might be called T-{1,2,3,5}, T-{4}, etc. Object id_m's are most useful for the three special object types, all_m, any_m, and pow_m, since they are specifically defined in terms of monads. There is also a practical reason for the usefulness of object id_m's: If we have a database of 138,019[1] monads, the object id_o's of the pow_m object type range from 1 to $2^{138{,}019}$, which would take on the order of $138,019$ bits to implement - something which is clearly intractable.

2.7.5 Part_of, overlap

2.7.5.1 Part_of

The subset relation gives rise to a relation between objects which is quite crucial in building hierarchies. Take two objects, O_1 and O_2.

$$O_1 \text{ part\_of } O_2 \iff O_1 \subseteq O_2$$

Thus if the monads of O_1 are all in the set of monads comprising O_2, then O_1 is part_of O_2.

In our example figure on page 14, Phrase-5 is part_of Phrase-4. Phrase-2 is part_of Clause_atom-2.

2.7.5.2 Overlap

Objects can share monads. The notion of overlap formalises this idea. This notion is expressed in terms of the set intersection operator. If the inter-

[1] The number of words in the Greek New Testament as published in the Nestle-Aland 27^{th} edition is 138,019.

section of the monads of two objects is non-empty, then they share monads, and are thus overlapping. Take two objects, O_1 and O_2.

O_1 overlaps with $O_2 \iff O_1 \cap O_2 \neq \emptyset$

In our example figure on page 14, Phrase-5 overlaps with Phrase-4. Phrase-5 does not overlap with Phrase-6. Clause_atom_2 overlaps with Sentence-1.

Object types can also said to be overlapping or non-overlapping:

An **object type is overlapping** if and only if some of its objects overlap.

An **object type is non-overlapping** if and only if it is not overlapping.

In our example figure on page 14, only the Phrase object type is overlapping. The rest of the object types are non-overlapping.

It could be made part of the type-system of a full database model based on the MdF model that one could specify that a given object type was overlapping or non-overlapping, and, in the latter case, uphold this constraint automatically when attempting to add objects of this type.

2.7.6 Covered by and buildable from

In some applications, we want certain object types to form a hierarchy. For example, sentences may be formed from words, and paragraphs may be formed from sentences.

Since objects are made of monads, not other objects, we need some way of specifying hierarchies. This is done using the notions covered_by and buildable_from. These notions are, in the MdF model, only extensional in nature, i.e., only by inspection can we decide whether an object type is covered_by or buildable_from another. It could, however, be made part of the type system in a full database model based on the MdF model, such that constraints could be upheld when adding or deleting objects.

Despite their names, these two notions are not opposites of each other. Rather, buildable_from expresses the same as covered_by, only with an additional constraint.

2.7.6.1 Covered by

An **object type T_{high} is covered_by object type T_{low}** if and only if the union of all the monads in all the objects of the set of objects of T_{high} has set equality with the union of all the monads in all the objects of the set of objects of T_{low}, AND for all objects O_{high} of T_{high} there exists a set of objects S of T_{low} such that the monads of the union of all the objects in S are the same as the monads of O_{high}, AND for all objects O_{low} of T_{low}, it is the case that there exists exactly one object O_{hg} of T_{high} such that O_{low} part_of O_{hg}.

Note that the name 'covered_by' is slightly counter-intuitive: It is the larger that is covered by the smaller, not the smaller that has the larger as a canopy over it.

Doedens says in his book (p. 70) that covered_by induces a partial ordering on objects types. This is not true: As a counterexample, the following setup makes covered_by fail to be antisymmetric: Suppose that we have an MdF database with just three monads and just two object-types, T_1 and T_2, and suppose that both T_1 and T_2 have only one object: The one consisting of the monads $\{1, 2, 3\}$. Then T_1 is clearly covered_by T_2, and T_2 is clearly covered_by T_1, yet $T_1 \neq T_2$.

In our example figure on page 14, Phrase is covered_by Phrase and Word, Clause_atom is covered_by Clause_atom, Phrase and Word, and Clause is covered_by Clause, Clause_atom, Phrase, and Word. Sentence is covered_by everything.

2.7.6.2 Buildable from

We sometimes want to say something that is a little stronger than that two object types are in a covered_by relationship. We sometimes want to specify that the two object types are non-overlapping as well.

An **object type T_{high} is buildable_from object type T_{low}** if and only if T_{high} is covered_by T_{low}, AND both T_{high} and T_{low} are non-overlapping.

Doedens notes in his book that buildable_from induces a partial ordering on object types that are non-overlapping. This is, of course, not true, since covered_by is not a partial order.

In our example figure on page 14, the only object type that is not buildable_from something is Phrase, since it is overlapping.

2.7.7 Consecutive, gaps

2.7.7.1 Consecutive

Basically, two objects are consecutive if they follow each other in a neat row without any gaps in between. This is the heart and soul of text representation.

However, sometimes it is handy to exclude certain parts of the database from consideration. For example, in our example figure on page 14, we may wish to concentrate on “The door, ..., was blue” as embodied in Clause-1. We say that “was blue” is consecutive to “The door,” with respect to the set of objects constituting Clause-1. We call Clause-1 the “Universe”. Note that this usage of “Universe” is closer to the usage of “Substrate” which will follow later than to the later usage of “Universe”.

In order to formally define the notion of consecutive, we first need a definition of a range:

Definition: A range of monads is denoted by a .. b, and has the following meaning: If $b < a$, then the range is empty. If $b \geq a$, then the range denotes the set of monads starting at a and including all monads up to and including b.

For example, "1..3" denotes the set $\{1, 2, 3\}$, while "3..2" denotes $\emptyset$.

Definition: A set of n monads, M, where $n \geq 2$, is consecutive with respect to a set of monads, U, if, when ordering the elements of M according to '<', such that $m_1 < \ldots < m_n$, for all i, $1 \leq i \leq n-1$, it holds that $(m_i + 1..m_{i+1} - 1) \cap U = \emptyset$.

Definition: A set of n objects, S, where $n \geq 2$, is consecutive with respect to a set of monads, U, if the objects can be ordered as $O_1, \ldots, O_n$ such that for all i, $1 \leq i \leq n-1$, it holds either that $(O_i = \emptyset \vee O_{i+1} = \emptyset)$ or (i.e., exlcusive or) the set consisting of the last monad of O_i and the first monad of O_{i+1} is consecutive with respect to U.

2.7.7.2 Gaps

As we have seen, objects need not consist of a contiguous string of monads. For example, in our example figure on page 14, Clause-1 consists of "The door, was blue.". There is a gap in Clause-1 with respect to "The door, which opened towards the East, was blue.".

An object is said to have gaps if its monads are not consecutive. The monads of a gap in an object are not part_of the object. Furthermore,they are not outside of the object, i.e., all of the monads in a gap in an object O are both $\geq$ the first monad of O and $\leq$ the last monad of O. Gaps are always "maximal", i.e., we group as many monads as possible into a gap.

Definition: A gap in an object O with respect to a set of monads U is a set of monads, H, such that:

1. $H \neq \emptyset$ AND
2. H is consecutive with respect to U AND
3. $\neg$ (H overlaps with O) AND
4. $\forall h_i \in H . h_i \geq O.first() \wedge h_i \leq O.last()$ AND
5. $\forall H'$ gap in O with respect to U. $H' \subseteq H$ (i.e., H is maximal)

where $O.first()$ and $O.last()$ refer to the first and last monad of O, respectively.

2.7.8 Border, separated, inside

2.7.8.1 Border

An object always has a first and a last monad (which is a consequence of the well-formedness axiom for natural numbers). The first monad of an object is called its **left border**, and the last monad of an object is called its **right border**. Together, the left border and the right border of an object constitute its **borders**. Two notions that can be defined in terms of the borders of objects are "separated" and "inside".

In our example figure on page 14, the left border of Phrase-4 is 5, and its right border is 7. The left border of Phrase-2 is the same as its right border, namely 3. The left border of Clause-1 is 1, and its right border is 9.

2.7.8.2 Separated

Two objects are separated if and only if the right border of one of the objects is $<$ the left border of the other. Thus even if we add the gaps in the objects to the objects, they still do not overlap.

In our example figure on page 14, Word-3 and Phrase-3 are separated. Phrase-3 and Phrase-4 are separated. Clause-1 and Clause-2 are *not* separated.

2.7.8.3 Inside

An object O_1 is inside O_2 if and only if the left border of O_2 is $\leq$ the left border of O_1 and the right border of O_2 is $\geq$ the right border of O_1. Note that, even though an object is inside another object, they need not overlap, since objects may have gaps.

In our example figure on page 14, Phrase-5 is inside Phrase-4. Clause-2 is inside Clause-1. Word-4 is inside Clause-2.

2.8 Conclusion

In this chapter, we have presented, in condensed and abridged form, Doedens' work as it applies to demands on text databases and to the MdF model itself. Not much has been original in this chapter. We have touched upon text-dominated databases and expounded text, upon what constitutes a database model, and upon Doedens' thirteen demands on a text database model. We have given a gentle introduction to the MdF model, followed by in-depth discussions of the four key concepts of the MdF model. We have then defined a lot of useful concepts in relation to the four basic concepts in the MdF model.

Chapter 3

The Extended MdF model

3.1 Introduction

In this chapter, we motivate and develop the Extended MdF model (EMdF). It is basically the standard MdF model with a few enhancements. We shall explicitly say when something in the EMdF model diverges from the MdF model, but we shall not explicitly say it when something is an obvious extension to the MdF model.

We begin by giving a short introduction to the EMdF model, in which the philosophy of the EMdF model is explained and a few important concepts introduced. We then go on to detailing the changes and additions that the EMdF model makes to the MdF model as it applies to monads, objects, object types, features, enumerations, and values. Finally, we give a concluding section.

3.2 Short introduction to the EMdF model

The EMdF model builds on top of the MdF model a structure of concepts that makes it easier to implement the model. The EMdF model does not remove any functionality of the MdF model[1]. Rather, it extends the functionality slightly. Hence the "Extended" predicate.

The EMdF model assumes some sort of Object Oriented Database Management System (OODBMS) for its implementation. One of the key features that such an OODBMS must have is that it must be able to assign unique **object ids** to objects. Furthermore, These object ids must have some ordering on them, e.g., if they are integers, then the "less than" ordering. Other than that, it is assumed that the OODBMS supports lists of values, and that these values can be object ids or anything else.

The most obvious difference between MdF terminology and EMdF terminology is that EMdF terminology makes a distinction between abstrac-

[1]Except for putting restrictions on the codomain of features.

tions (such as "object") and their implementation. In the EMdF model, the abstractions are often suffixed with "_m", whereas the concrete implementation is often suffixed with "_d". For example, there is a distinction between "objects" in the "MdF sense", which in the EMdF terminology become "object_m's", and "objects" as seen from the viewpoint of the OODBMS, which are called "object_d's". Thus some object_d's implement object_m's, but there are object_d's with no object_m counterpart.

The object ids assigned by the OODBMS to object_d's are called object id_d's.

3.3 Monads

3.3.1 Introduction

Monads are treated differently in the EMdF model than in the basic MdF model. In the EMdF model, there are two concepts which are both named "monad": monad_m's and monad_d's.

Monad_m's are the EMdF name for MdF monads. Thus they are integers with two functions:

1. They are the substance out of which objects are built, and
2. They keep track of the sequence of text.

Monad_d's are object_d's in the database which correspond to monad_m's. They have a special structure which is used when querying and updating the database.

We begin by describing the concept of a monad_m. We then go on to describing the concept of a monad_d as well as four functions on monad_d's. Finally, we define the concept of the max_m.

3.3.2 monad_m

"Monad_m" is the EMdF name for an MdF monad . The reason for having a distinct name for this concept is that it is convenient to be able to talk about "abstract" monads, as opposed to the concrete implementations of structures in the database which correspond to monads.

In the EMdF model, the string of monad_m's starts at 0, not at 1 as in the MdF model. This is because 0 is the "real" bottom element of the natural numbers, thus rendering the EMdF model slightly more "clean" than the MdF model. Besides, having the first monad_m be 0 simplifies implementation slightly.

3.3.3 monad_d, mdm, mmd, mdo, mdstarthere

A monad_d is an object_d. It represents a monad_m.

There are a number of functions associated with a monad_d. They are: mdm, mmd, mdo, and mdstarthere.

The **mdm** function takes a monad_d and returns the monad_m to which it corresponds.

The **mmd** function takes a monad_m and returns the monad_d to which it corresponds.

The **mdo** function takes an object type T and a monad_m m and returns the set of objects of type T that contain m.

The function **mdstarthere** takes an object type T and a monad_m m and returns the set of objects of type T that start at m. This function is useful when querying the database.

These functions basically manipulate the contents of a monad_d, except mmd, which must use information outside of the monad_d's if it is to be implemented efficiently.

There are exactly as many monad_d's as there are monad_m's in the database.

3.3.4 max_m

The string of monads in an EMdF database starts at 0 and ends at max_m. Thus there are max_m + 1 monads in the database. The number max_m may change over time as more monads are added to the database, or as monads are deleted from the database.

3.4 Objects

3.4.1 Introduction

In this section, we detail quite a lot of concepts that are useful when talking about objects. Objects are one of the more important concepts of the MdF model, and so the EMdF model treats them with care. We first discuss the important change from the MdF model that objects are not unique in their monads. We then go on to discussing object_m's, object_d's, object_dm's, and object id's. Finally, we discuss some functions on objects, monads, first, and last.

3.4.2 Objects are not unique in their monads

One very important difference between the MdF model and the EMdF model is that, whereas the MdF model stipulates that objects are unique in their monads, the EMdF model makes no such claim. In fact, it explicitly states that objects need not be unique in their monads. Thus two objects of the same type may in fact have exactly the same monads.

The reason for this change is that this restriction was found to be cumbersome in real applications. It was also found to be unnecessary, given that:

1. An OODBMS is assumed which can assign object ids uniquely, and
2. For querying purposes, it is not necessary that objects be unique in their monads.

The latter fact arose out of the process of designing MQL.

3.4.3 object_m

The objects of the MdF model are called object_m's in the EMdF model. The reason for this name-change is that we need to distinguish between objects in the "abstract" MdF sense and objects stored in the database. Many objects in the database are not counterparts of object_m's, but are there for house-keeping.

3.4.4 object_d

The objects that are stored in the database are called object_d's. These objects may or may not correspond to object_m's. It is assumed that each object_d has a unique object id_d.

3.4.5 object_dm

Object_dm's are object_d's that correspond to object_m's. Giving them a special name allows us to refer to them explicitly, and helps in removing ambiguity when talking about them.

3.4.6 object ids

3.4.6.1 object id_d

An object id_d is a unique object identifier given to an object in the database by the OODBMS. This object id_d may be the id of any object in the database, including housekeeping objects that are not counterparts of objects in the MdF sense.

Notation: $O.id$ denotes, for the object_d O, the object id_d of that object.

3.4.6.2 object id_m

Object id_m's of the MdF model are not redefined in the EMdF model. However, it is necessary to realize that, since objects are no longer unique in their monads, the object id_m need not be an id for any other object types than the special object types all_m, any_m, and pow_m. Thus object id_m's should only be used for these types. For application-specific object types, the concept of object id_m has been supplanted by the concept of object id_d. It is assumed that there is some total ordering on the object id_d's.

3.4.6.3 linear ordering of objects per types

The concept of linear ordering of objects per type has had to be redifined slightly from the EMdF model. The reason is that the objects_dm's are no longer unique in their monads. Thus the concept of linear ordering per type is now defined as follows. Take an object type T and two objects of type T, O_1 and O_2:

1. If O_1 and O_2 do not have the same monads, then linear ordering is decided as in the MdF model (see section 2.7.3).
2. If O_1 and O_2 have the same monads, then $O_1 <_T O_2$ iff $O_1.id < O_2.id$.

Note that object id_o's of the three built-in types (all_m, any_m, and pow_m) are not affected by this change, since they never have two distinct objects with the same monads.

3.4.6.4 object ordinal, object id_o

The concepts of object ordinal and of object id_o from the MdF model are not changed in the EMdF model (see section 2.7.4 on page 18). They are still based on the linear ordering of objects per type.

3.4.7 O.monads()

It is useful to define, on object_m's and object_dm's, a function, **monads**, which returns the set of monad_m's of which the object consists.

Notation: This function with the object_m O as the argument may be denoted "O.monads()".

3.4.8 O.first(), O.last()

It is useful to define, on object_m's and object_dm's, functions which return the left border and the right border respectively. Thus **first** and **last** return, for a given object, a monad_m.

Notation: The notational convention O.first() and O.last() is used to denote these functions taken on object O.

3.5 Object types

3.5.1 Introduction

The concept of object types has not changed from the MdF model. However, a few new concepts have been introduced. These include type_m, type_d, type_id, and inst, which we discuss below.

3.5.2 type_m

A type_m is an object type of an object_m "in the abstract". Thus it is precisely what is called an object type in Doedens' book.

3.5.3 type_d

A type_d is an object_d which stores information in the database about a given type_m.

3.5.4 type_id

Type_id is an integer uniquely identifying a type_m within the context of an EMdF database. Thus all type_m's have an identifier. For implementation purposes, this type_id could be an object id_d of a type_d, or it could be an integer index into some array of type_d's.

3.5.5 inst

In the EMdF model, a special function, inst, is assumed. It is assumed to be efficiently computable. It takes an object type and a universe as its arguments and returns an ordered list of object id_d's. This list is sorted by object ordinal.

A Universe is a contiguous set of monads which starts at some monad_m a and ends at some monad_m b, where $a \leq b$.

The inst function returns, for a given object type T and a given universe U, all the objects of type T which are part_of U.

Thus inst(T, all_m-1) gives the list of all objects of type T, sorted on object ordinal.

3.6 Features

3.6.1 Introduction

In this section, we discuss some useful concepts in relation to features. First we discuss feature_m's, then three implementation-aids called feature_value_d, feature_info_d, and feature_id.

3.6.2 feature_m

A feature_m is a feature of an object_m "in the abstract". Thus it is precisely what Doedens calls a feature (were it not for restrictions which I later put on features).

3.6.3 feature_value_d

A feature_value_d is an object_d in the database which implements the value of a feature_m when applied to a particular oject, when that value is stored (as opposed to computed).

3.6.4 feature_info_d

A feature_info_d is an object_d in the database which implements information about a particular feature_m type.

3.6.5 feature_id

Feature_id is an integer which defines a feature uniquely with respect to a certain object type. Thus a pair (type_id, feature_id) uniquely determines which feature is in question.

3.7 Enumerations

3.7.1 Introduction

Enumerations are meant to be a convenient way of naming integers and grouping them in a type. Thus each enumeration constant has both a value and a type associated with it.

In this section, we first discuss two concepts relating to actual enumeration constants, enum_constant_name and enum_constant_value. We then discuss an implementation aid called enum_info_d which gives information about the type of an enumeration constant. Finally, we discuss two concepts which can be useful when implementing enumeration constant types, namely enum_id and enum_index.

3.7.2 enum_constant_name, enum_constant_value

An enum_constant_name is a string of characters which uniquely identifies a particular enum_constant_value (which is an integer) within an enumeration type.

3.7.3 enum_info_d

An enum_info_d is an object_d which stores information about a particular enumeration type. In particular, it stores some kind of representation of a mapping between enum_constant_names and enum_constant_values.

3.7.4 enum_id, enum_index

An enum_id is an integer uniquely identifying a particular enumeration type within the database. An enum_index is some kind of pointer to a mapping element in an enum_info_d, where a "mapping element" consists of an enum_constant_name and an enum_constant_value.

3.8 Values

3.8.1 Introduction

I have chosen, for implementation purposes, to restrict the values that a feature might take on in the EMdF model, to be the following atomic types:

- integer
- string
- enumeration constant

The reason for doing this is that, at some point, a design decision like this has to be taken. Since the EMdF model is meant to be easily implementable, and to serve as the design playground for an actual implementation that I am working on, this choice seems to be right. Any other implementation might wish to implement fewer, more, or other values. But these seem to be basic.

Object id_d is not included because including them makes it more difficult to delete objects from the database. Two ways to get around this could be as follows: One is to delete the transitive closure of objects that point to each other when deleting an object which has a pointer to another object. The other is to simply assign the "NULL" object id_d to a stored feature in object B which points to object A, when deleting object A. The user could be given the choice of which one they wanted. Note that both of these approaches requires that we keep track of which objects point to each object.

Binary Large Objects (BLOBs) are not included because they are not necessary for the application domain for which I am developing EMdF. They could easily be added.

3.8.2 Integer, String

The integer and string types are well known datatypes in computer science. Thus I will not comment on them.

3.8.3 Enumerations

Enumerations have been treated above and deserve no further comment.

3.9 Conclusion

In this chapter, we have presented, in abstract form, a lot of concepts which are useful when implementing an EMdF database. We have given a short introduction to the EMdF model. We have discussed monads, including the concepts of monad_m, monad_d, and max_m. We have discussed objects and how they are affected by the EMdF model. One important change in the EMdF model from the MdF model is that objects are not unique in their monads. We have discussed the concepts of object_m (which correspond to MdF objects "in the abstract"), object_d (which are concrete objects in the database, not necessarily corresponding to object_m's), object_dm (which are object_d's corresponding to object_m's), and object ids (which have not changed from the MdF model: Only the underlying definition of linear ordering per type has changed). Finally, we have defined some functions on objects which are useful for implementation.

Chapter 4

Implementing the EMdF model

4.1 Introduction

This chapter gives some implementation details which are suggestions for how to implement the EMdF model in an actual OODBMS.

4.2 Sets of monads

4.2.1 Introduction

In this section, we introduce two datatypes which constitute an elegant way of representing sets of monads. First, we circumscribe the Monad_Set_Element, which is just a range of monads, $a..b$. Second, we define a datatype, Set_of_monad_ms, which is a way of representing a set of monads using Monad_Set_Elements.

4.2.2 Monad_Set_Element

We need a datastructure to hold two monads: first and last. The intended interpretation is that it denotes the range first..last. The invariant is that first $\leq$ last. The first and last attributes of a Monad_Set_Element mse can be accessed as mse.first and mse.last.

4.2.3 Set_of_monad_ms

4.2.3.1 Introduction

The Set_of_monads is the basic building block of an MdF object_m. It can be implemented as a vector of Monad_Set_Elements. The intention is that the Monad_set_elements in the vector be sorted, and that there be at least one monad of "gap" in between each Monad_set_element. This invariant (stated more precisely below) gives rise to some very useful properties.

The vector of monad_ms of a Set_of_monad_ms A is denoted by A.monad_ms.

First, we give an invariant on the vector of Monad_Set_Elements in a Set_of_monad_ms. Then, from this invariant, we prove a uniqueness proposition on the vector, namely that any vector of Monad_Set_Elements which adheres to the invariant is the unique representation of a set of monad_m's. Finally, we give a list of easily implementable operations which one must be able to perform on a Set_of_monad_ms.

4.2.3.2 Invariant

There is the following invariant on the Monad_Set_Element's stored in the vector:

For all Monad_Set_Elements mse in the vector, it is the case that:

1. Either
   (a) Its predecessor prev is empty (i.e., is not there), OR
   (b) prev.last+1 < mse.first,
2. AND either
   (a) Its successor succ is empty (i.e., is not there), OR
   (b) mse.last+1 < succ.first

This means:

∀ mse IN vector: (∀ prev before mse: prev.last+1 < mse.first) ∧ (∀ succ after mse: mse.last + 1 < succ.first)

Here the two nested for-all quantifiers take care of 1(a) and 2(a) by being vacuously true when prev and succ are not there.

This mathematical predicate captures these two intuitions:

1. All Monad_Set_Elements are *maximal*, in the sense that they extend as far as they can without violating the other intuition, which is that
2. The Monad_Set_Elements are *sorted* in such a way that, for any two Monad_Set_Element's A and B, where A is the direct predecessor of B,

   A.last < B.first

   which can be strengthened, by intuition 1., to:

   A.last + 1 < B.first

   which means that there is at least one monad_m in between each Monad_Set_Element.

4.2.3.3 Uniqueness

We will give a detailed proof that two sets of monads are identical if and only if their representations are identical. The proof rests on the invariant.

Thus the representation of a given set is a unique representation – there is only one.

Proposition:

Any two Set_of_monad_ms represent the same set of monad_m's if and only if their monad_ms vectors are identical.

Proof:

Take two Set_of_monad_ms's A' and B'. Let A = A'.monad_ms and B = B'.monad_ms.

"if": Assume that the two vectors are identical. Then they trivially represent the same set.

"only if:" Assume that the two vectors represent the same set. The proof works by induction on the subscript operator i.

Base case: i = 0.

To prove: A[0] = B[0].

Let mseA = A[0] and mseB = B[0].

Since the two sets are identical, certainly mseA.first == mseB.first, since this is the first monad of the sets. However, it is also the case that mseA.last == mseB.last. To see this, consider the invariant given above.

Let mseA' = A[0+1] and mseB' = B[0+1].

Now, since the invariant holds, it is the case that

$$\text{mseA.last} + 1 < \text{mseA'.first}.$$

It is also the case that

$$\text{mseB.last} + 1 < \text{mseB'.first}.$$

Assume for the sake of contradiction that mseA.last $\neq$ mseB.last. Assume Without Loss of Generality that mseA.last > mseB.last. Then there must be at least one monad m in the set represented by A that is not in the set represented by B. This is because there must be a distance of at least 1 between mseA.last and mseA'.first, as there must be between mseB.last and mseB'.last.

Thus there is at least one monad m = mseB.last+1 which is not in B but which is in A. To see this, note that, first, m cannot be in B, since

$$\text{mseB.last}+1 = \text{m} < \text{mseB'.first}.$$

Second, note that, since

$$\text{mseA.last} > \text{mseB.last},$$

m is in A, since this is equivalent to

$$\text{mseA.last} >= \text{mseB.last}+1 = \text{m} >= \text{mseB.first} = \text{mseA.first}$$

Thus, there is a contradiction, and mseA.last = mseB.last. Thus the base case holds.

Induction step Assume that mseA' = A[i-1] and mseB' = B[i-1] are identical. Specifically, the following holds:

$$\text{mseA'.first} = \text{mseB'.first} \wedge \text{mseA'.last} = \text{mseB'.last}$$

We wish to prove that mseA = A[i] and mseB = B[i] are identical.

First, we prove that mseA.first = mseB.first. Second, we prove that mseA.last = mseB.last.

To see that mseA.first = mseB.first, assume for the sake of contradiction that

$$\text{mseA.first} \neq \text{mseB.last}.$$

Assume Without Loss of Generality that

$$\text{mseA.first} < \text{mseB.first}.$$

Then there must be at least one monad

$$\text{m} = \text{mseA.first}$$

which is in the set represented by A, but which is not in the set represented by B. To see this, note first that m is in A. Second, note that m is not in B by virtue of being in mseB. Note also that m cannot be in B by virtue of being in mseB', because of the assumption that

$$\text{mseB'.last} = \text{mseA'.last}.$$

Since m is clearly not in B, m is in the set represented by A but not in the set represented by B. Since the two sets of monad_m's are assumed to be the same, we have a contradiction. Thus our assumption is false that mseA.first $\neq$ mseB.first, and thus it is true that

$$\text{mseA.first} = \text{mseB.first}.$$

Second, we prove that

$$\text{mseA.last} = \text{mseB.last}.$$

Assume, for the sake of contradiction, that

$$\text{mseA.last} \neq \text{mseB.last}.$$

Assume, Without Loss of Generality, that

$$\text{mseA.last} > \text{mseB.last}.$$

Then there is at least one monad, m = mseB.last + 1, which is in the set represented by A but which is not in the set represented by B.

To see this, note that if i points to the last element in B, then certainly there is a contradiction, since then the last monad in the set represented by A is not in the set represented by B, since the vectors are sorted and mseA.last > mseB.last.

Assume, therefore, that mseB" = B[i+1] exists. Then, by the invariant,

$$\text{mseB.last} + 1 = \text{m} < \text{mseB''.first}.$$

Note that this means that m is not in the set represented by B, since it "falls between two chairs" or, rather, two Monad_set_elements.

Note also that m is in the set represented by A, since

$$\text{mseA.last} > \text{mseB.last}$$

which is equivalent to

$$\text{mseA.last} \geq \text{m} = \text{mseB.last} + 1 > \text{mseB.first} = \text{mseA.first}.$$

Thus, m is in the set represented by A, but not in the set represented by B, so the two sets are not identical, which we assumed. Therefore, there is a contradiction, and it is therefore false that mseA.last ≠ mseB.last. Therefore, it is true that mseA.last = mseB.last.

We have thus proved the induction step.

We have thus proved that, if the sets represented by two vectors in two Set_of_monad_ms's are identical, then the vectors will be identical.

We have thus proved the proposition.

4.2.4 Operations on sets of monad_ms:

For MQL and the implementation suggested in this chapter, at least the following operations are needed on sets of monad_ms:

1. A boolean function indicating part_of between two sets.
2. A boolean function indicating whether the set is empty.
3. A function returning a set of monad_ms which is the largest gap in a set of monad_ms starting at some monad_m.
4. Set difference.
5. Set construction from a range $a..b$.
6. Offsetting a set of monad_ms from or to any given monad_m (see below).

All of these can easily be implemented given the invariants on Monad Set Element and Set Of Monad_ms.

4.3 Specific object_d classes

4.3.1 Introduction

In this section, we discuss a lot of useful ways of implementing specific object_d classes. First, we discuss the monad_d. Then follow object_dm, seq_d (which keeps track of the sequence of monads), inst_d, feature_value_d, feature_info_d, type_d, enum_info_d, and lastly root_d, which binds the whole database together.

4.3.2 monad_d

A monad_d contains the following information:

1. The monad_m to which this monad_d corresponds.
2. For each user-defined object type in this database, a set of object id_d's of object_dm's of the given type which contain this monad. Coupled with each object id_d should be a boolean which answers the question "does this object_m start at this monad_m?"

These lists must be optimized for insertion and deletion, and need not support `[]` indexing.

The reason for having the monad_m in there is the following:

- We need some way of relating monad_d's to monad_m's.

The reasons for having the lists of object id_d's are the following:

- When traversing the database, it is useful to be able to go from monad_m's to those object_dm's which contain this monad_m. For example, one might be interested in all the words that are part of a clause which contains a word with a known monad_m. To find all the words, one needs just look at which clauses contain the known monad_m, and work backwards from there. This is important when displaying the contents of an EMdF database in a full-screen view.
- When inserting or deleting monads, it is necessary to be able to go from monad_m's to those object_m's that contain them.
- When querying the database, it is absolutely necessary to know which object_m's start where.

The reasons for having the boolean that says "starts here" is:

- For querying purposes, we have to have some way of telling which object_m's start at a given monad. This boolean is one way of doing it.

4.3.3 object_dm

An object_dm is basically these three things:

1. A pointer to the monad_d which corresponds to the first monad_m of the set of monad_m's of which this object_dm is built,
2. A set of monad_m's which is built relative to the first monad_m. That is, the set $\{4, 5, 6\}$ is stored as $\{0, 1, 2\}$ with respect to the first monad_m, which is 4. The first monad_m can be found by looking at the monad_d to which we have a pointer.
3. A list of feature-values.
4. A type_id indicating the type of the object_dm.

The reason for having this setup of a set of monads relative to some first monad, and a pointer to the monad_d which represents this monad_m is:

- It makes things much simpler, easier, and faster when inserting and deleting monads.

4.3.4 seq_d

In any EMdF database, there is an object_d called **seq_d**. This object_d is used to keep track of the sequence of monads. It is simply a **list [0..max_m] of object id_d's of monad_d's**. The index into this list maps monad_m's to monad_d's.

4.3.5 inst_d

In any EMdF database, there is a special object_d, **inst_d**, which is used to implement the inst function (see section 3.5.5).

This object has, for each object type, a **list of pointers to object_d's** which each contains the following:

1. An object id_d of an object_dm.
2. The first and last monad_m of this object_dm.

These lists are sorted on object ordinals.

The reason for sorting these lists on object ordinals is the same as the reason for including the first and last monad_m:

- Doing things this way helps in implementing the inst function.

When asked for the inst function for a particular universe, that value can be expressed in terms of an upper and a lower index into this array. These boundaries can be found by binary search, since we have the first and last monad_m's of each object_dm. Thus the computation complexity of finding the boundaries is $O(\log(\text{max\_m}))$.

4.3.6 feature_value_d

A feature_value_d holds information about the value of a certain feature taken on a certain object. This is so only if the feature is stored and not calculated. It consists of the following:

1. The kind of value (integer, string, or enumeration constant).
2. An integer to hold the contents if they are an integer.
3. A pointer to a string to hold the contents if they are a string.
4. A pair (enum_id, enum_index) to hold the contents if they are an enumeration constant.

The reason for holding what is effectively a pointer to the value of an enumeration constant, rather than the value itself, is that we may wish to change the underlying values of the enumeration constants after the creation of the enumeration type. This is then easily done (see enum_info_d below).

4.3.7 feature_info_d

A feature_info_d holds information about the type of a feature_m and consists of the following:

1. A string containing the name of the feature_m to which this feature_info_d corresponds.
2. A value indicating which type the feature is (integer, string, enumeration constant, or some calculated function).
3. An enum_id for the type of the enumeration constant, if the feature returns an enumeration constant.
4. Some way of specifying which function within the EMdF engine to call if this feature is calculated. Probably an integer specifying where in a table of functions to look.

4.3.8 type_d

A type_d holds information about an object type. It consists of the following:

1. A string specifying the name of the type_m.
2. An array of feature_info_d's in feature_id order. This is the way you go from feature_id to feature_info_d.

It is probably a good idea if the implementation of an EMdF engine, upon startup, reads all the type_d's (see root_d below) and makes these mappings (hash-tables or some sort of balanced binary trees) between these things:

- Between feature_name and feature_id.
- Between object type name and type_id.

4.3.9 enum_info_d

An enum_info_d holds information about the type of a given enumeration constant. It contains the following:

1. Maybe a type name, depending on whether enumeration constants can be given a name in the type language of the full access model.
2. Definitely an array of pairs (enum_constant_name, enum_constant_value). This is how we go from enum_index to the other two.

It is probably a good idea if the implementation of an EMdF engine, upon startup, reads all the enum_info_d's an builds some sort of mapping between the following:

- For each enumeration type, a mapping between enum_constant_name and enum_constant_value.
- For all enumeration types, a mapping between (enum_id,enum_index) and enum_constant_value.

4.3.10 root_d

The root_d object (there is only one) in an EMdF database is the single most important object_d. It contains all the information that is necessary for accessing the rest of the database. It contains the following:

1. A pointer to the seq_d object (see above).
2. A pointer to the inst_d object (see above).
3. An array of type_d's indexed by type_id.
4. An array of enum_info_d's indexed by enum_id.

4.4 Algorithms

4.4.1 Introduction

In this section, we give some algorithms which are useful when inserting or deleting monads or objects from an EMdF database. We first treat monads: Their insertion and their deletion. We then treat objects: Their insertion and their deletion.

4.4.2 Insertion of monads

Inserting monads in an EMdF database is assumed not to be a common task. It is assumed that the text is more or less fixed, and that insertion and deletion of monads is infrequent.

The problem can be classified in this way:

1. Inserting a single monad at monad b, pushing monad b and all subsequent monads one upwards.
2. Inserting a sequence of monads $0 \mathinner{..} a$ at monad b, pushing monad b and all subsequent monads $a+1$ upwards.
3. Inserting a set A of monads relative to a monad b.

I choose to treat only the second instance. The third instance can easily be built from this instance, while it is more useful than the first instance and, it seems, not much more difficult.

One problem immediately presents itself: What do we do with objects which start before b yet either end inside the range $b \mathinner{..} b+a+1$ or end after it? Do we insert the monads int these objects? If so, do we insert all the monads? Do we also do it to objects which have no monads inside this range (which is conceivable if all the monads are before and after the range)? Do we do this to all object types which have objects which are like this? In any case, the problem is not simple.

I propose to simply leave a hole in those objects.

To insert , we would need to:

1. Allocate a new seq_d,
2. copy the object id_d's from 0 to $b-1$ into the new seq_d,
3. Make $a+1$ new monad_d's, giving them monad_m's in the range $b \mathinner{..} b+a$,
4. Insert these in the new seq_d,
5. Increment max_m by $a{+}1$, keeping the old max_m somewhere around.

6. Copy all of the old object id_d's from b to the old max_m into the new seq_d, starting from $b + a + 1$,
7. Access all the monad_d's from $b + a + 1$ till the new max_m and increment the monad_m field by $a + 1$,
8. Go through the monad_d's in the range $b + a + 1 \,..\, b + 2a + 1$. Pick out the object_d's which started before b (i.e., which have a pointer to a monad_d before b), make a "hole" in these, inserting a "gap" of monad_m's in the range $b \,..\, b + a$.
9. Go through each list in inst_d, updating those first and last monad_m's that need updating.

4.4.3 Deletion of monads

As with insertion, deleting monads in an EMdF database is assumed not to be a common task. It is assumed that the text is more or less fixed, and that insertion and deletion of monads is infrequent.

The problem can be classified in this way:

1. Deleting a single monad at monad b, moving monad $b + 1$ and all subsequent monads one downwards.
2. Deleting a range of monads $0 \,..\, a$ at monad b, moving monad $b + a + 1$ and all subsequent monads $a + 1$ downwards.
3. Deleting a set A of monads relative to a monad b.

I choose to treat only the second instance. The third instance can easily be built from this instance, while it is more useful than the first instance and, it seems, not much more difficult.

One problem immediately presents itself: What do we do with those objects that contain monads in the range to be deleted?

I propose to delete all objects which contain monads in the range to be deleted. This is the simplest and easiest thing to do.

To delete a range of monads $0 \,..\, a$ starting at a monad b:

1. Go through the monad_d's corresponding to the range $b \,..\, b + a + 1$ and gather all the object id_d's which need to be deleted (i.e., which have monads in this range).
2. Delete these objects (see "deletion of objects" below).
3. Move all object id_d's in seq_d $a + 1$ down, starting from $b + a + 1$ and finishing at max_m.
4. Update the usage count of seq_d and update max_m.

5. Go through the monad_d's pointed to by seq_d, starting from b and finishing at max_m. Subtract $a + 1$ from the monad_m in each monad_d.

6. Go through the lists in inst_d, updating those first and last monad_m's that need updating.

4.4.4 Insertion of objects

There are two instances of this problem:

1. Insert a single object.

2. Insert a list of objects.

Since the second instance is more general, is not too difficult to implement, and has certain advantages over the single-object instance, I choose to deliberate on the second instance.

It is assumed that the list of objects S is given in a form in which each object O is represented by an object type $O.T$, a set of monads $O.m$, and a set of values of data-oriented features $O.f$.

The insertion can be done this way:

1. Check to make sure that all the objects stay within 0 .. max_m. If not, append the right number of monad_m's to the database (insert them at the end). This check can be done by finding the largest last monad L of the $O.m$'s and comparing it to max_m. If L is larger, insert monads such that max_m becomes L.

2. Create a list R of object id_d's with as many entries as there are objects in S. Each entry must be "empty". This list is meant to be returned to the user, and must contain, in the same order as the objects in S, the object id_d's of the created objects.

3. For each object type T represented in S, create an empty list I_T. This list is meant to hold a product of indices and object id_d's. The indices are into the appropriate lists in inst_d which must have object id_d's inserted.

4. For each object O in S at index i:

   (a) Create a new object_dm o with object id_d $o.id$.

   (b) Insert $O.m$ into o such that it is represented relative to the first monad.

   (c) Create the object_d's necessary to hold the data $O.f$, and insert the data.

(d) Make sure that o points to the right monad_d.

(e) Insert $o.id$ in the appropriate list for each monad_d corresponding to the monad_m's in $O.m$, remembering to set the "starts here" boolean to true only for the first monad_d and to false for all others.

(f) By binary search, find the appropriate index at which to insert $o.id$ into the appropriate list in inst_d. Place this index, along with $o.id$, into I_T.

(g) Put $o.id$ in $R[i]$.

5. For each list I_T:

   (a) Sort I_T such that higher indices are first. If two indices are the same (which it might be, if we are to insert "at the same place"), sort according to the set of monads in the object_dm's pointed to by the object id_d's that are coupled with the indices. If this gives a tie, sort on object id_d. (Note that this corresponds to our definition of linear ordering of objects per type.)

   (b) Run through I_T in sorted order, inserting the right object_id and first and last monad_m's in the appropriate list in inst_d.

6. Return R.

The reason for returning the list of object id_d's is that the user might wish to know the object id_d's of the newly created objects.

4.4.5 Deletion of objects

There are two instances of this problem:

1. Delete one object.

2. Delete a set of objects.

Since the second instance is the more general, not too difficult to implement, and handy for deletion of monads, I choose to deliberate on this problem.

It is assumed that the set of objects S to be deleted is expressed in terms of their object id_d's.

It can be done this way:

1. For each object o in S, run through the list in each monad_d corresponding to the monad_m's of which o is built, deleting the pointers to o.

2. For each object type T of object types actually represented in S, make a set I_T of indices into the appropriate list in inst_d whose entries must be deleted. That is, for each object type T represented in S, for each object O of type T in S, find O's index into the appropriate list in inst_d by binary search and store this index in I_T.

3. Sort each set I_T such that the highest indices are first.

4. Run through each set I_T in sorted order, deleting the entries in inst_d.

5. Delete all objects in S together with their associated data.

4.5 Conclusion

In this chapter, we have described the design of one way to implement some of the key elements in an EMdF database. We began by looking at the crucial concept of a set of monad_m's. We then went on to several classes of object_d's, including monad_d, object_dm, seq_d, inst_d, feature_value_d, feature_info_d, type_d, enum_info_d, and root_d. We then described four algorithms: One for insertion of monads, one for deletion of monads, one for insertion of objects, and one for deletion of objects.

All of the material in this chapter has been original.

Chapter 5

MQL - Mini QL

5.1 Introduction

In this chapter, I develop, motivate, and explain the MQL language. MQL stands for "Mini QL", and is a subset (though not a proper subset) of the QL language developed in Doedens' book.

Two key structures are developed in this chapter: The sheaf, which is a datastructure to hold the results of a query, and the syntax and semantics of MQL itself. The semantics of MQL are given in operational terms, making it relatively easy to implement in practice, given the right primitives. The operational specifications are given in terms of functions written in some Pascal-like language which is not defined rigorously here. Anyone familiar with procedural, imperative programming languages like Pascal, C, Oberon, etc. should have no difficulty reading the specifications.

We first give some motivation for why MQL is necessary. We then touch briefly on the notion of 'topographic languages' which Doedens defined in his dissertation. MQL is a topographic language. We then give the definition of two concepts, namely Universe and Substrate. These are central in the semantic definition of MQL. We then list and explain some differences between Doedens' work and my work. After that, we give an informal introduction to MQL by means of some examples. This section should give the reader some "feel" for MQL which will be helpful when reading the rest of the chapter. We then give the definition and explanation of the *sheaf*, which is the datastructure which holds the results of an MQL query. We then touch briefly on a possible architecture for an MQL query engine. The next section details quite a bit of information about MQL: Datatypes needed for MQL, lexical rules for MQL, the grammar for MQL, the concatenation operators of MQL, MQL variables, and a long exposition on why MQL needs to be compiled and what should be checked for while compiling. The next section talks about the concept of a state as embodied in the semantic specification of MQL. The next section, which is the bulk of this chapter, gives the semantics of

MQL in an operational specification. This is done by defining functions on syntactic units of MQL, and explaining them in plain English. Finally, we draw some conclusions.

5.2 Motivation

QL is a nice language, but it has one main drawback: It is very, very difficult to implement in practice, because it is so powerful, and its semantics are so abstract. The fact that Doedens gave the semantics as denotational semantics means that there is no easy path to follow from syntax and semantics to implementation.

MQL attempts to address this problem.

One way of doing this has been to reduce the power of the language to a minimum, while still retaining power enough for "most applications", whatever that may be. The guiding design principle here has been to "take all elements from QL which are easily implementable, and to leave out the rest".

The other way of doing this has been to give the semantics of each syntactic construct in MQL, not as denotational semantics, but as an operational specification.

5.3 The notion of 'topographic languages'

In his PhD dissertation, Doedens defines a new notion, namely that of 'topographic languages'. A language is topographic if there are isomorphisms between the graphs denoting the structure of the expressions of the language and the graphs denoting the objects denoted by the expressions of the language.

I believe that the notion of 'topographicity' is closely related to the linguistic notion of 'iconicity', whereby there is an 'iconic' (cf. 'topographic') relationship between the linguistic expression and the object it denotes. Here 'iconic' means that the linguistic expression 'looks like' or 'resembles' the object which it denotes. The notion of 'topographicity' is, however, more formally defined than the concept of 'iconicity', since it is defined in terms of graphs and isomorphisms between graphs.

QL (and, therefore, MQL) is a topographic language. This means that there is an isomorphism between the abstract syntax trees made from instances of QL- (and MQL-) queries and the graphs that could be drawn of the part_of relationships between objects in the MdF (or EMdF) database.

5.4 Preliminary definitions

5.4.1 Universe

A universe is a single contiguous string of monad_ms. This is not a definition taken from Doedens' book. In Doedens' book, a universe can be any set of monads. However, as a consequence of the semantics of MQL, what Doedens uses as a universe in QL is always a contiguous string of monads in MQL.

5.4.2 Substrate

A substrate is a set of monad_ms. It may have gaps, but it always starts at some monad_m a and ends at some monad_m b (where we may have $b = a$). It is always part_of the accompanying universe. The first and last monads of a substrate need not, however, be the same as the first and last monad of the accompanying universe.

The significance of subtrates is that we match relative to substrates!

5.5 Changes from Doedens' work

This section details the changes that I have made to QL in order to obtain MQL. It assumes a lot of familiarity with Doedens' work.

The core elements are still there – `topograph`, `blocks`, `block_string`, `block`, and `object_block`. The `power_block` is also there, but in another form.

The main change, semantics-wise, is that a `blocks` need *not* match the whole universe. Thus there is no need for "`..`" before and after the `object_block`s which one wishes to retrieve.

Instead, the first block in a `block_string` has to be a special `object_block`, called `object_block_first`, which can optionally be specified as having to be the first in the universe. The semantics-function for this `object_block_first` is passed an *index* into an array, assumed to be easily computable, (called inst(T,U)) which gives all the objects of type T which are part_of the universe U. These are assumed to be ordered according to the lexicographic ordering within the type. The semantics function for `object_block_first` does not return until it either finds the next match within this array (starting at the index), or it exhausts the array.

Note that I have changed the definition of a `sheaf` slightly, but not much. The main difference is that a `matched_object` can now be several different things, which it couldn't in Doedens' definition of the `sheaf`.

Also, there is no provision for extracting *features* into the `sheaf`. It is assumed that the implementation only wishes to obtain object_id's from the MQL engine, and that the objects themselves can then be inspected for features if desired. This is because, in the application from which MdF sprang (i.e., a syntactic database of Hebrew and Aramaic), and in the application for which I am planning to use EMdF, it is normal practice, when a query returns a lot of matches, that the user can then pick from a list of matches and only *then* the features looked for are retrieved. This saves memory.

There *is* a provision for variables. However, it is an *imperative* understanding of variables, not a *functional* understanding, as in QL. Thus there is a *state*, not an *environment* (or Free_var_bindings).

Note also that, when comparing features to values (for selecting only those objects which match certain *feature* criteria), the value is *always* a *single* value, and the feature must be *equal* to this value. This differs from QL, where a value is basically a set of values, and a feature's value must be *a member of* this set for the query to match.

I have also restricted the values to be

- enumeration constants (for things like "`singular`", "`plural`", etc.)
- integers
- strings
- variables (which must be of one of the above three types)

Variables are implictly typed, and can have a standard value, which means that no value is assigned.

Finally, in QL, objects are not retrieved unless they are explicitly marked for retrieval. In MQL, the reverse is true: objects are retrieved unless they are explicitly marked for non-retrieval.

5.6 Informal introduction to MQL by means of some examples

5.6.1 Introduction

This section informally introduces MQL by way of a number of examples. The example database which we will use is the same as in Doedens' book, namely part of Melville's "Moby Dick":

> "CALL me Ishmael. Some years ago - never mind how long precisely - having little or no money in my purse, and nothing particular to interest me on shore, I thought I would sail about a little and see the watery part of the world. It is a way I have of driving off the spleen, and regulating the circulation. Whenever I find myself growing grim about the mouth; whenever it is damp, drizzly November in my soul; whenever I find myself involuntarily pausing before coffin warehouses, and bringing up the rear

> of every funeral I meet; and especially whenever my hypos get such an upper hand of me, that it requires a strong moral principle to prevent me from deliberately stepping into the street, and methodically knocking people's hats off - then, I account it high time to get to sea as soon as I can. [...]
>
> "[...] By reason of these things, then, the whaling voyage was welcome; the great flood-gates of the wonder-world swung open, and in the wild conceits that swayed me to my purpose, two and two there floated into my inmost soul, endless procesions of the whale, and, mid most of them all, one grand hoofed phantom, like a snow hill in the air."

Suppose that we have in this EMdF database the domain-dependent object types "paragraph", "sentence", and "word", which correspond to paragraphs, sentences, and words of the text. And suppose that we add to the object type "sentence" the feature "mood", which draws its values from the enumeration type { imperative, declarative }. And suppose that we add to the object type "word" the features "surface" (which gives the surface text of the word) and "part_of_speech" (which gives the part of speech of the word). The codomain of the feature "part_of_speech" on the object type "word" draws its values from the enumeration type { adjective, adverb, conjunction, determiner, noun, numeral, particle, preposition, pronoun, verb }. This hypothetical database will give the background for most of the examples in our informal introduction to MQL.

5.6.2 A query does not match the whole database

In the QL language in Doedens' book, a query matches the whole database. This need not hold true for an MQL query, for two reasons. First, because the universe and substrate over which an MQL query is processed need not be the whole database. And second, which is a more subtle reason, because the syntactic construction of an MQL query does not reflect any supposition that a query matches the whole database, as a QL query does.

5.6.3 topograph

An MQL query is called a topograph. Consider the following topograph:

```
[sentence]
```

This topograph retrieves the set of all sentence objects in the database.

5.6.4 features

A query can specify which features an object must have for it to be retrieved. For example, consider the following topograph:

```
[word
   surface = "Ishmael" or part_of_speech = verb;
]
```

This topograph retrieves the set of all words which are either "Ishmael" on the surface of the text, or whose part of speech is "verb".

5.6.5 object_block,object_block_first

There are several types of blocks. They are meant to come in a string of blocks, where each block in the string must match some part of the database in order for the whole string to match. Two such blocks are the `object_block` and the `object_block_first`.

Object blocks are the heart and soul of MQL queries. They are used to match objects and objects nested in other objects. An object block (be it an `object_block` or an `object_block_first`) consists of the following parts:

1. The opening square bracket, '`[`'.
2. An identifier indicating the object type of the objects which we wish to match.
3. An optional keyword, "`noretrieve`" or "`retrieve`". The default is "`retrieve`". The keyword "`noretrieve`" says as much as "I do not wish to retrieve this object, even if matched". It is useful for specifying the context of what we really wish to retrieve.
4. An optional keyword, "`first`" or "`last`", which says as much as "this object must be first/last in the universe against which we are matching.
5. An optional Boolean expression giving what features need to hold true for this object for it to be retrieved.
6. An optional list of assignments to variables.
7. An optional inner `blocks` which matches objects inside the object.
8. The closing square bracket, '`]`'.

Note that only the first object block in a string of blocks can have the "`first`" keyword, and only the last `object_block` in a string of blocks can have the "`last`" keyword. In order to build this into the semantics of MQL, I have made a special syntactic category for the *first* object block in a string of blocks, namely `object_block_first`.

Consider the following `topograph`:

```
[sentence
    mood = imperative;
    [word noretrieve first
         surface = "CALL";
    ]
    [word]
]
```

This `topograph` retrieves the set of sentences which are imperative, and whose first word is “CALL”. Within each sentence in that set, we retrieve the second word, but not the first. The only sentence in our example database which qualifies is the first sentence.

5.6.6 power_block

The power_block is used to indicate that we allow some distance in between two blocks. A power_block must always stand between two other blocks, and can thus never be first or last in a query. It comes in two varieties: A “plain vanilla” power block, syntactically denoted by two dots, “`..`”, and a power block with a limit to how many monads may come between the two embracing blocks, denoted by the two dots followed by the “less than” symbol, followed by an integer, e.g., “`.. < 5`”. Consider the following topograph:

```
[sentence
    [word
        part_of_speech = preposition]
    .. < 4
    [word
        part_of_speech = noun]
    ..
    [word last
        surface = "world"]
]
```

This topograph retrieves the set of sentences which have a word that has part of speech preposition, followed by a word which has part of speech noun, and which is within 4 monads of the preposition, followed by the last word of the sentence, which must be “world”. Within that sentence, retrieve all the three words. The only sentence which qualifies is the second.

5.6.7 opt_gap_block

An opt_gap_block is used to match an optional gap in the text. It consists of:

1. The opening square bracket, '`[`'.
2. The keyword "`gap?`".
3. An optional "`noretrieve`" or "`retrieve`". The default is "`noretrieve`"
4. The closing square bracket, '`]`'.

The opt_gap_block matches gaps in the *subtrate* against which we are matching. Thus if we look at the example in figure 2.1 on page 14, we can construct the following topograph:

```
[clause
    [clause_atom
        [word
             surface = "door,"
        ]
    ]
    [gap? noretrieve]
    [clause_atom noretrieve]
]
```

This retrieves all clauses which happen to have inside them a clause_atom which contains the word "door,", followed by a gap, followed by a clause_atom. The gap and the second clause_atom are not retrieved. This would retrieve clause-1. The gap need not be there. Since this is the case, Clause-2 would also be retrieved, were it not for the fact that it contains no word "door,".

The default is for the result of an opt_gap_block not to be retrieved. Thus one needs to explicitly write "`retrieve`" if one wishes to retrieve the gap.

5.6.8 variables

Variables are declared before each group of object blocks (see later). The understanding of variables is imperative rather than functional in MQL. Thus variables must be assigned a value before they can be used to match against something. Variables are declared and used like this:

```
var $c, $n, $g;
[word
    part_of_speech = article;
    $c := case;
    $n := number;
    $g := gender;
]
[word
```

```
        (part_of_speech = noun or part_of_speech = adjective)
         and case = $c and number = $n and gender = $g;
    ]
```

Assuming that the word object type has features part_of_speech, case, number, and gender, this topograph retrieves all pairs of words for which the part of speech of the first word is "article", the part of speech of the second word is "noun" or "adjective", and for which the case, number, and gender agree for the two words.

This concludes our gentle, informal introduction to MQL.

5.7 Sheaf

5.7.1 Introduction

The `sheaf` is the datastructure which holds the results of an MQL query. It has been amended from the Sheaf in Doedens' book. The reason for choosing to use a sheaf-like structure to hold the results of an MQL query is the following: I found that, even though the sheaf is a little hard to understand, it nicely captures the results we expect from a query, as well as the topographic nature of MQL queries.

The `sheaf` is a set of `straws`. A `straw` is a set of `matched_objects`. A `matched_object` denotes a "hit" of one object block against an object in the database. A `matched_object` can, in one of its syntactic incarnations, have a `sheaf` inside it, thus capturing the embedding that can take place in hierarchies of MdF object types. This embedding closely matches the structure of an MQL query.

5.7.2 Grammar

The grammar for the sheaf is:

```
sheaf  :  "//"  my_set_of_straws  ;
my_set_of_straws  :  /* empty */  |  set_of_straws  ;
set_of_straws  :  "{}"  | straw  (  ','  straw  )*
straw  :  "{}"  |  matched_objects  ;
matched_objects  :  matched_object  |
                  matched_objects  matched_object  ;
matched_object  :  NIL_mo  |  EMPTY_mo  |  '['  OBJECT_ID  sheaf  ']'  ;
```

5.7.3 Explanation

As already remarked, a `sheaf` is a set of `straws`, and a `straw` is a set of `matched_objects`. It is well to begin our discussion of the `sheaf` by explaining in detail what a `matched_object` is.

5.7.3.1 matched_object

A `matched_object` can be one of three things:

1. `NIL_mo`
2. `EMPTY_mo`
3. an object id coupled with a `sheaf`.

The first syntactic incarnation, `NIL_mo`, never gets into any `straw`. It is simply a value which a `matched_object` as a syntactic object may have during execution of a query. It conveys the information that a match failed.

The second syntactic incarnation, `EMPTY_mo`, also never gets into any `straw`. It is used during the execution of a query to indicate that an `opt_gap_block` was matched against an empty stretch of the database (i.e., no monads).

The third syntactic incarnation is what gets into `straws`. The object id may either be an object id_d or an object id_m. In the case of an object id_d, it denotes a match of an `object_block` (or an `object_block_first`) against an object_m in the database. In the case of an object id_m, it denotes a match of an `opt_gap_block` against a gap in the database, and as such is the object id_m of a pow_m object.

Note also that, for the third syntactic incarnation, there can be a `sheaf` inside the `matched_object`. The intuition of this is that inside an object, there can be strings of objects which need to be matched. These then go into the `sheaf`.

5.7.3.2 straw

A `straw` is a set of `matched_objects`. It can either be "`{}`", which denotes an "empty set", or it can be a `matched_objects`. The intuition of a `straw` is that it denotes a "string" of `matched_objects`. Of course, a set is unordered, but the intention is that a `straw` denote one match of a string of blocks that need to be matched.

For example, consider the following topograph:

```
[word
   surface = "the";
]
[word
   part_of_speech = noun;
]
```

One `straw` will contain two `matched_objects`, one for the "the", and one for the following noun.

5.7.3.3 sheaf

A `sheaf` is a set of `straws`. It can be one of the following things:

1. "`//`", in which case the matching failed, or
2. "`// {}`", in which case the matching did not fail, but the results were empty, or
3. "`//`" followed by a nonempty set of straws, which constitutes a nonempty successful match.

By way of example, the topograph in the previous section about the `straw` would retrieve a `sheaf`, which would contain as many `straws` as there were combinations of "the" and a noun.

5.7.3.4 Correlation of sheaf-structures and MQL-structures

This section explains in a little more detail the correlation between sheaf-structures and MQL-structures. It assumes familiarity with certain MQL-structures. Therefore, it may be skimmed or skipped until a thorough understanding of MQL has been gained. The reader is then advised to return to this section, and even to refer back to it as he or she reads through the exposition on MQL.

Because MQL is a topographic language, there should also be topographicity between the results of a query and the query from which it was constructed.

A `sheaf` constitutes a matching of a `blocks`. Note that there are two places in the MQL grammar which refer to a `blocks`. One is the top-level start symbol, `topograph`. This means that the result of a query is a `sheaf`. The other is `my_blocks`. The `my_blocks` syntactic construct is used in two places: in an `object_block_first` and in an `object_block`. It can either be a `blocks`, meaning that we want to match something inside the `object_block(_first)`, or it can be empty, meaning that we put no restrictions on what must be inside the `object_block(_first)`. If it is a `blocks`, the result will, of course, be a `sheaf`. This is why the third syntactic incarnation of a `matched_object` has a `sheaf` inside it.

A `straw` constitutes *one* matching of a `block_string`. However, the the result of a match of a `block_string` is a *set of `straws`*, not a single `straw`! This is because a `block_string` may result in more than one `straw`. This will become clear as we study the semantics of the `block_string`.

A `matched_object` corresponds to a block of some kind, be it an `object_block`, an `object_block_first`, or an `opt_gap_block`.

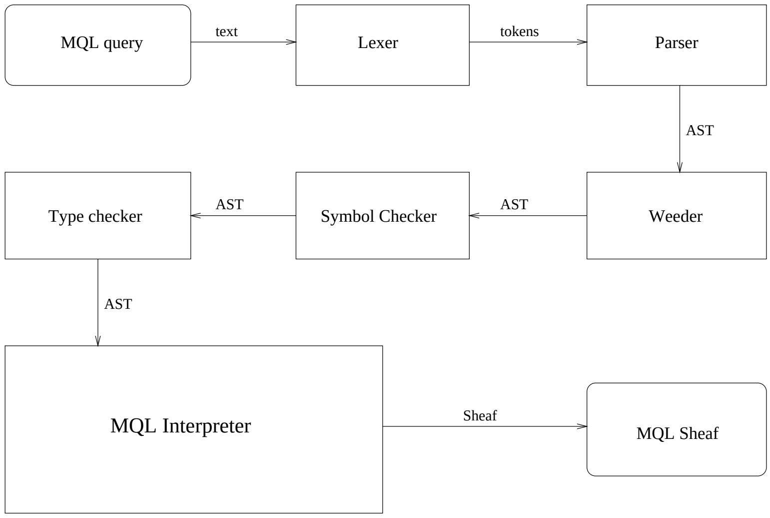

Figure 5.1: An architecture for an MQL query engine

5.7.4 Sets as lists

The semantics of MQL are such that lists can be used instead of sets. These lists must be optimized for prepending and appendening of other lists. Thus, even though the concept is a set, a list can be used without checking for "membership" when doing "union" with (i.e., prepending or appending of) another list.

5.8 An architecture for an MQL query engine

In figure 5.1 on page 57, we have suggested an architecture for an MQL query engine. The figure should be straightforward to read and should deserve no further comment.

5.9 MQL

5.9.1 Datatypes

We need the following datatypes:

- `INTEGER` - ordinary integers.
- `index` - an integer suitable for indexing into inst(T,U)
- `monad` - a monad_m
- `boolean` - a Boolean

- `sheaf`, `set_of_straws`, `straw`, `matched_object` - as in the grammar for the sheaf.
- `mo_nr` - a pair (`matched_object`,`boolean`) which gives a `matched_object` paired with a `boolean` saying whether or not it is to be retrieved (MatchedObject_NoRetrieve).
- `set_of_mo_nr` - a set of "`mo_nr`"s.
- `enum enum_first_last {no_first_last, first, last}` - an enumeration.
- `set_of_monad_ms` - a set of monad_m's : the same as in an object. Used for universe, substrate, and pow_m object id_m's.
- `instances` - an array intended to hold inst(T,U).
- `state` - a state of variables in a `blocks`.
- `object` - an `object_d`.

5.9.2 Lexical rules

Whitespace is to be ignored in the lexer, except as delimiters of tokens. The token `IDENTIFIER` is any identifier valid in C or Pascal. The token `STRING` is a string enclosed in "double quotes". The token `INTEGER` is any integer, positive or negative. The lexer is case sensitive.

5.9.3 Grammar

In this section, we give the full grammar of MQL. It is specified in a form of Backus-Naur Form which resembles that used in Yacc and Bison. The only liberty that I have taken with the Yacc/Bison syntax is to place tokens which are strings in "double quotes", e.g., `"var"`. Other tokens are in UPPER CASE, e.g., `IDENTIFIER`.

This grammar has been tested with Bison and has been found to be unambiguous and LALR(1).

The full grammar for MQL is:

```
topograph  :  blocks ;
blocks  :  variable_declarations  block_string  ;
variable_declarations  :  /* empty */  |  "var"  var_dec_variables  ';'  ;
var_dec_variables  :  variable  |  var_dec_variables
                      ','  variable  ;
variable  :  '$' IDENTIFIER  ;
block_string  :  block_str  |
                 block_str  power  block_string  ;
```

```
block_str  :  object_block_first  |
              object_block_first  '!'  rest_of_block_str  |
              object_block_first  rest_of_block_str  ;
object_block_first  :  '['  type  retrieval
                       firstlast features
                       variable_assignments
                       my_blocks  ']'  ;
type  :  IDENTIFIER
retrieval  :  /* empty */  |  "noretrieve"  |  "retrieve"  ;
firstlast  :  /* empty */  |  "first"  |  "last"  ;
features  :  /* empty */  |  ffeatures  ';'  ;
ffeatures  :  fterm  |  ffeatures  "or"  fterm  ;
fterm  :  ffactor  |  fterm  "and"  ffactor  ;
ffactor  :  "not"  ffactor  |
            '('  ffeatures  ')'  |  feature  ;
feature  :  feature_name  '='  value  ;
feature_name  :  IDENTIFIER  ;
value  :  enum_val  |  INTEGER  |
          STRING  |  variable  ;
enum_val  :  IDENITIFIER  ;
variable_assignments  :  /* empty */  |  vvas  ;
vvas  :  variable_assignment  |
         vvas  variable_assignment  ;
variable_assignment  :  variable  ":="
                        feature_name  ';'  ;
my_blocks  :  /* empty */  |  blocks  ;
rest_of_block_str  :  block  |
                      block  '!'  rest_of_block_str  |
                      block  rest_of_block_str  ;
block  :  opt_gap_block  |  object_block  ;
opt_gap_block  :  '['  "gap?"  opt_retrieval  ']'  ;
opt_retrieval  :  /* empty */  |  "noretrieve"  |  "retrieve"  ;
object_block  :  '['  type  retrieval  last
                 features  variable_assignments
                 my_blocks  ']'  ;
last  :  /* empty */  |  "last"
power  :  ".."  restrictor  ;
restrictor  :  /* empty */  |  '<'  limit  ;
limit  :  INTEGER  ;  /* non-negative integer,
                         may be 0.  */
```

5.9.4 The concatenation operators

There are two places in the grammar where an exclamation mark ('`!`') is used. One is in `block_str` between an `object_block_first` and a `rest_of_block_str`. The other is in `rest_of_block_str` between a `block` and a `rest_of_block_str`. It will be noted that both of these constructs have counterparts with no exclamation mark in between. The exclamation mark is called a "concatenation operator", as is the "invisible" counterpart.

A prepass should be done on the query so that, before parsing the query, the following syntactic operations should be applied:

- Each exclamation mark concatenation operator must be erased.
- Each invisible concatenation operator must be replaced with the string "`[gap? noretrieve]`".

Thus there are no exclamation mark concatenation operators in the Abstract Syntax Tree that gets passed to the MQL query interpreter.

Note that the net effect of these syntactic operations could equivalently be achieved after parsing with a simple manipulation of the Abstract Syntax Tree. This is actually the recommended approach.

The reason for having these concatenation operators is that we do not want the user to have to know about possible optional gaps in the text. This is very useful, for example in dealing with classical Greek, where any sentence-initial phrase may be split in the middle by a word from a certain set of well-known words. We do not want the user to explicitly mark that such gaps could optionally be there. It is better, when asking for results, to be shielded from one's ignorance than to retrieve incomplete results. Thus this amendment embodies the principle that "what you don't know won't hurt you".

5.9.5 MQL variables

5.9.5.1 Introduction

In this subsection, we expand on what we have already said about MQL variables by commenting on naming, usage, and typing.

5.9.5.2 Naming

Variables are set apart from the rest of the namespace by a prefixed dollar sign: '`$`'. It is the intention that there be no whitespace between the dollar sign and the `IDENTIFIER` which makes up the variable name. Scope rules could be applied to variables, but for now, let us simply have one namespace for each topograph, and let us say that a variable can only be declared once in each topograph.

5.9.5.3 Usage

Variables must be declared before they can be used. This is done before each `blocks`. Since a `blocks` can be nested inside an `object_block(_first)`, there can be several `variable_declarations`.

Variables must be assigned a value before they can be used. That is why the `variable_assignments` comes after the `features` inside the `object_block(_first)`. The intuition is, that a variable is not of much use within the `features` of the same `object_block(_first)`. A variable only becomes useful in comparing with a feature of another `object_block`. Thus, by enforcing the rule that a variable must be assigned a value before it can be used, and by ensuring that it gets assigned a value only after any feature comparison within the same `object_block(_first)`, we keep these intuitions.

A variable can only be assigned to once in a `blocks`.

A variable's "existence" is only within the `blocks` before which it has been declared. Thus a variable cannot be accessed outside of its `blocks`.

5.9.5.4 Typing

MQL variables are implicitly typed. As said earlier, an MQL variable can have any of these types:

- enumeration constants (for things like "construct", "absolute", etc.)
- integers
- strings

The type of a variable is determined by the type of the feature with which it is assigned. The compiler should check that a variable is used only in a context in which it is compared to something which is of the same type as the variable itself.

5.9.6 MQL needs to be compiled

5.9.6.1 Introduction

As noted in section 5.8,I think it would be wise to transform an MQL query, written in MQL, into an AST (Abstract Syntax Tree). Various checks should be performed on this AST before passing it to an interpreter which actually builds the sheaf. For example, variables should be type-checked (they are implicitly typed), and various well-formedness criteria should be checked for. In the following sections, we list some of the criteria that should be checked.

5.9.6.2 Scope-rules

To keep things simple, I suggest that we stipulate the requirement that no two variables in a `topograph` may be called the same. The scoping rules could be such that a given variable can only be "seen" within the `blocks` in which it was declared, but in order not to have to mess with variables of the same name in different, nested scopes, it is easier to stipulate that no two variables in a topograph may be named the same. It would be easy later to change this, since scoping-rules are well understood, but for now, let us stick with this.

5.9.6.3 Type-checking of object types

The type of object in an `object_block(_first)` must be a valid object type (i.e., the object type must exist in the EMdF database, and it must not be all_m, pow_m, or any_m!).

5.9.6.4 Type-checking of feature_name

The compiler should check that `feature_name` (of `feature`) is part of the type for the immediately enclosing object-type.

5.9.6.5 Type-checking of variable usage

There must be type-consistency (type-compatibility) between what a variable has been assigned and that to which it is compared.

5.9.6.6 Non-assigned variables

Note also that in the usage of value in feature, a variable may not occur which has not previously been assigned a value by a `variable_assignment`.

5.9.6.7 Only one variable assignment

It is the intention that there be only one variable_assignment per variable per `blocks`. The compiler should check this.

5.9.6.8 Variable usage only within its blocks

It is the intention that a variable only be used inside the `blocks` before which it was declared. The compiler should check this.

5.9.6.9 Usage of "first"

It is the intention that the "`first`" modifier of `object_block_first` only be used for the first `object_block_first` in a `blocks`. The compiler should enforce this. Note that the grammar allows an `object_block_first` after a `power`, and so this is not enforced by the grammar. It could probably (depending on the implementation of the AST) be checked by checking to see whether the `object_block_first` is the leftmost leaf in the `blocks` subtree. When we have explained the retrieval functions, it will be apparent that there is a very good reason for allowing an `object_block_first` after a `power`, in spite of the fact that it should not be allowed to be "first" in the universe.

5.9.6.10 Usage of "last"

It is the intention that the "`last`" modifer on `object_block_first` and `object_block` only be used for the last `object_block(_first)` in a `blocks`. The compiler should enforce this. Note that it is not in any way enforced by the grammar. It could probably (depending on the implementation of the AST) be checked by checking to see whether the `object_block(_first)` is the rightmost leaf in the `blocks` subtree.

5.10 State

5.10.1 Introduction

As we all know, a state is basically a mapping from variable names to their values at any given point in time. In this section, we briefly expand on the operations needed for the state concept in an MQL interpreter. We also give hints on how to implement such a state.

5.10.2 Operations on a state

Basically, we need three operations on a state:

- Creation of a (variable name,value) pair.
- Assigning a value to a variable.
- Reading the value of a variable.

If we later implement scoping of variable names, we will need to have two other operators, in order to allow overshadowing of variable names:

- Scoping of a state, which basically means putting an overriding state on the top of a stack, the new state now being the more current one, leaving the old state(s) available for reference for variable names which are not in the current state.
- Unscoping of a state, which is removing a state from the top of the stack of states.

In this paper, we will use the following notation for the above operations. For all states `S` and all variable names `v`:

- "`S + v`" indicates creation of a (variable name, value) pair in `S`. The variable name is `v`. The value is undefined at creation time. If `v` is there already, nothing special happens. Specifically, its value is not changed.
- "`S.v :=` *value*" indicates assigning a value to `v` in `S`.
- "`S.v`" indicates reading the value of `v` in `S`.

Since we have chosen not to implement scoping yet, we do not give any notation for scoping and unscoping states.

5.10.3 Hints on implementation

A state can be implemented as a hash-table of (variable name,value) pairs, where we hash on the variable name. This will allow easily for the three operations needed. If we later decide to implement scoping, our state will just be a stack of hash tables.

5.11 The Retrieval Functions R

5.11.1 Introduction

This section details the syntax and semantics of a set of retrieval runctions for MQL. Each retrieval function either operates on some syntactic construct in MQL or are helper functions.

The suggested way of reading this section is as follows: Once from front to back and once from back to front. That way, the reader will first get the broad picture in a top-down manner, and then the details in a bottom-up manner. The sections are arranged approximately in a depth-first order according to the order in which non-terminals are introduced in the MQL grammar, starting from the start symbol.

The return types of the retrieval functions are given after an arrow, "`-->`". Unlike Pascal functions, these retrieval functions may return more than one value at a time.

The standard value of a variable is indicated by a '`?`' question mark.

5.11.2 Arguments to the functions

The following are standard arguments to the retrieval functions. Not all of them need be in the formal parameters of any given retrieval function, but all of them are in the formal parameters of some retrieval function.

- U, Su - Universe, Substrate. A Universe is a single, contiguous string of monads. A Substrate Su is always part_of the Universe with which it belongs, but it may have gaps. We match relative to substrates!
- S - State. See above for a discussion.
- Sm - Start Monad – the monad at which we are to start our matching.
- i - index into inst(T,U) which we are to use.
- limit - limit of `..` (`power`), counted in monad_ms. If 0, it means "no limit".

The semantics of the arguments is call-by-value, unless prefixed by the keyword "`var`" in the list of formal parameters, in which case they are call-by-name. Nothing is changed which is not call-by-name. Therefore, call-by-value parameters might be implemented as `const`-references, if we use, e.g., C++.

5.11.3 topograph

5.11.3.1 syntax

```
topograph  :  blocks  ;
```

5.11.3.2 semantics

The retrieval function R for the topograph is:

```
/*
 * R_topograph
 * returns : sheaf
 * ON FAIL: "// " (see R_blocks)
 */

function R_topograph(U,Su,topograph) --> sheaf;
begin
  return R_blocks(U,Su,{},blocks);
end;
```

5.11.3.3 explanation

The `R_topograph` retrival function takes a universe `U`, a substrate `Su`, and a `topograph`. It returns a `sheaf`. The only thing it does is to call `R_blocks` with its arguments, as well as an empty state, thus returning all the possible matches of the `topograph`.

Note that we do *not*, as QL does, specify that `U` and `Su` must be all_m-1 and all_m-1 respectively. This is because this decision should be made higher up in the architectural hierarchy. For example, we might wish to search the results previously made, or we might wish to restrict the search to certain books (in a Biblical setting), without specifying it in the MQL query itself.

5.11.4 blocks

5.11.4.1 syntax

```
blocks  :  variable_declarations  block_string  ;
```

5.11.4.2 semantics

```
/*
 * R_blocks
 * returns : sheaf
 * ON FAIL: "// "
 */

function R_blocks(U,Su,var S,blocks) --> sheaf;
var
    Sheaf : sheaf;
    i : index;
    Result : set_of_straws;
    SStraws1 : set_of_straws;
begin
    i := 0;
    Result := {};
    /* expand S by variable_declarations (if not
       already there), and set all variables in
       variable_declarations to the standard
       value */
    R_variable_declarations(S,variable_declarations);

    /* Treat the block_string. */
    repeat
        SStraws1 :=
           R_block_string(U,Su,S,i,0,block_string);
        Result := Result "union" SStraws1;
        i := i + 1;
    until (SStraws1 = {});  /* Return when match
                             fails. */
    if (Result = {}) then
```

```
            Sheaf := "//";
        else
            Sheaf := "//" Result;
        return Sheaf;
    end;
```

5.11.4.3 explanation

The `R_blocks` retrieval function takes a universe `U`, a substrate `Su`, a call-by-name state `S`, and the `blocks` syntactic construct which we wish to treat. It returns a `sheaf`. If it fails, it returns "`//` ".

The idea of `R_blocks` is that it returns a `sheaf` with all the possible matches of the `block_string`.

The function is very simple. The reason why we enclose the call to `R_block_string` in a `repeat` ... `until` statement is that we wish to retrieve all of the `block_string` matches we can get. Thus we increment `i` each time.

The `i` variable is an index into an inst(T,U) array, and it is used as such in `R_object_block_first`. The `i` variable is a call-by-name variable all the way down through the call-path down to `R_object_block_first`, and can be modified by all of the functions along the path.

The reason why we do not include the call to `R_variable_declarations` in the `repeat` ... `until` statement is that it is unnecessary: Once the variables are declared, they need not be redeclared. Further, since we have stipulated (and since the compiler enforces) that all variables should be assigned a value before their first use, we do not need the side-effect that all variables are assigned the standard value in the call to `R_variable_declarations`.

5.11.5 variable_declarations

5.11.5.1 syntax

```
variable_declarations  :  /* empty */  |
                          "var"  var_dec_variables  ';'  ;
var_dec_variables  :  variable  |
                      var_dec_variables  ','  variable  ;
variable  :  IDENTIFIER  ;
```

5.11.5.2 semantics

```
/*
 * R_variable_declarations
 * returns : nothing
 * Note: Modifies S
 */
function R_variable_declarations(var S,variable_declarations);
```

```
begin
    /* Decide syntactic incarnation */
    if (variable_declarations = "var" var_dec_variables ';')
    begin
        for each variable v in var_dec_variables do
        begin
            S := S + v;
            S.v := ?;
        end;
    end;
end;
```

5.11.5.3 explanation

The R_variable_declarations retrieval function takes a call-by-name state S and the variable_declarations syntactic unit which we wish to treat. The function does not return anything, but simply operates on the state S.

There are two possible outcomes of the function. If variable_declarations is empty, nothing happens. If, on the other hand, variable_declarations is not empty, each variable in var_dec_variables gets added to S (if it is not already there), and the variable gets the standard value "?".

5.11.6 Restrict

The Restrict function is used in R_block_string.

5.11.6.1 semantics

One implementation could be like this:

```
/*
 * Restrict
 * returns : set_of_monad_ms
 */
function Restrict(U : set_of_monad_ms, m : monad) --> set_of_monad_ms;
begin
    return U - (U.first()..m-1);
end;
```

5.11.6.2 explanation

The Restrict function is not a retrieval function, in that it does not operate on a syntactic unit of the MQL grammar. Rather, it is a utility function which is used in the R_block_string retrieval function. It takes a set of monad_m's U and a monad_m m. It returns a set of monad_m's.

The result of the `Restrict` function is `U` minus all the monads from `U`.first() to `m`-1. Thus the first monad of the result will be `m`, if {`m`} is part_of `U`.

5.11.7 join

The `join` function is used in `R_block_string`, in `R_block_str`, and in `R_rest_of_block_str`.

5.11.7.1 semantics

```
function join(s : straw, SS : set_of_straws) --> set_of_straws;
var
  Result : set_of_straws;
begin
    Result := {};
    for (each straw s' in SS) do
    begin
        Result := Result "union" { (s "union" s') };
    end;
    return Result;
end;
```

5.11.7.2 explanation

The `join` function is not a retrieval function, in that it does not operate on a syntactic unit of the MQL grammar. Rather, it is a utility function which is used in various retrieval functions. It takes a straw `s` and a set of straws `SS` and returns a set of straws.

What `join` does is to "multiply" `s` into all the straws in `SS`. If concatenation is used for union, `s` is prepended to each straw `s'` in `SS`.

5.11.8 block_string

5.11.8.1 syntax

```
block_string  :  block_str  |
                 block_str1  power  block_string1  ;
```

Note: The `1`'s on the `block_str` and `block_string` in the rule are just there for identification purposes: They are just "`block_str`" and "`block_string`" respectively.

5.11.8.2 semantics

```
/*
 * R_block_string
 * Returns : set_of_straws
 * ON FAIL: {}  Only fails if all the i's within the Universe
 *          have been tried unsuccessfully.
 */
R_block_string(U,Su,var S,var i,limit,block_string) --> set_of_straws;
var
    Straw1 : straw;
    locallimit : INTEGER;
    U',Su' : set_of_monad_ms;
    i' : index;
    SStraws1, SStraws2, Result : set_of_straws;
begin
    /* --- decide which syntactic incarnation to use --- */
    if (block_string = block_str) then
    begin
        return R_block_str(U,Su,S,i,limit,block_str);
    end
    else /* block_str1  power  block_string1 */
    begin
        while(true) do
        begin
            SStraws1 := R_block_str(U,Su,S,i,limit,block_str1);
            if (SStraws1 = {}) then
                return {}; /* This means that the block_str1 was
                              not present, so we needn't search
                              any further for the power and the
                              block_string1. */
            else
            begin
                Result := {};
                for (each straw Straw1 in SStraws1) do
                begin
                    locallimit := R_power(power);
                    /* --- Restrict U and Su to begin with
                       Straw1.last()+1 --- */
                    U' := Restrict(U,Straw1.last()+1);
                    Su' := Restrict(Su,Straw1.last()+1);
                    i' := 0;
                    /* --- This repeat...until statement embodies
                       the idea that the power plus block_string1
```

```
                              matches _all_ those matching block_string1
                              inside the universe, not just the first
                              one. --- */
                           repeat
                               SStraws2 :=
                                  R_block_string(U',Su',S,
                                    i',locallimit,block_string1);
                               if (SStraws2 <> {}) then
                                    /* --- multiply Straw1 into all
                                       straws in SStraws2 --- */
                                    Result := Result "union"
                                              join(Straw1, SStraws2);
                               i' := i' + 1;
                           until (SStraws2 = {});
                       end;
                       if (Result <> {})
                           return Result;
                   end;
                   i := i + 1;
               end;
           end;
       end;
```

5.11.8.3 explanation

The `R_block_string` retrieval function takes a universe `U`, a substrate `Su`, a call-by-name state `S`, a call-by-name Start Monad_m `i`, a limit `limit`, and the `block_string` syntactic construct which we wish to treat. It returns a set of straws. The result of `R_block_string` is one match of the `block_string`.

The invariant on `R_block_string` is that it should only return if it can either return a match to the `block_string` or has tried all possibilities without success.

If the syntactic incarnation at hand is just a single `block_str`, this function simply returns the result of a call to `R_block_str`. This is in harmony with the invariant above, since `R_block_str` also only returns if it has either found a match or has exhausted all possibilities.

If the syntactic incarnation is a `block_str` followed by a `power` followed by a `block_string`, another method is used. Here, we first retrieve the results of the `block_str`. If this is empty (i.e., `R_block_str` failed), we know that the `block_string` after the `power` is not going to match anything either, and we can return an empty set of straws indicating a fail. If, however, the call to `R_block_str` returned a nonempty set of straws `SStraws1`, then we do the following: For each straw `Straw1` in `SStraws1`, we make a new universe and a new subtrate, based on `U` and `Su`, but with all monads up to and including the "last" monad_m in Straw1 taken away.[1] We then retrieve the block_string with successive indices (which are used in R_object_block_first) until R_block_string returns a failed match (empty set of straws), all the while adding the join of Straw1 and the results of R_block_string to a temporary variable, Result, which is a set_of_straws. We do this because we want the semantics that a block_str plus power plus block_string matches *all* the block_strings after the power within the universe, joined to the first block_str, not just the first one. This is consistent with the intuition that a power has semantics similar to those of a "star" wildcard in Unix shell pathnames, e.g., "foo*bar" matches both "foobar", "fooabar", and "foothisisaverylongintermediatestretchbar".

If the Result variable is empty after having gone through the set of straws returned by R_block_str, it was maybe just bad luck that the combination of block_str and block_string did not match. Thus, because we must only return a fail if we have tried all possibilties, we increment i by 1 and do all this again, thus hopefully matching the block_str at a point which which will match with the block_string. Note that this always terminates, since R_object_block_first, which is the one to use i, returns a fail if i is too big, i.e., if it goes beyond the length of inst(T,U). This fail is then propagated up through the hierarchy.

The function is actually quite simple, despite its cluttered look. Its outline is as follows:

1. IF (the syntactic incarnation is block_str) THEN
   (a) return the result of R_block_str with all the parameters just passed on.
2. ELSE (i.e., the syntactic incarnation is block_str1 power block_string1) WHILE (true) DO
   (a) Calculate SStraws1 := R_block_str with all of the formal parameters which we got.
   (b) IF (Sstraws1 is empty) THEN (we know that it is no use looking for the block_string1 after the power, so) return {}, which is a failed match.
   (c) ELSE
      i. Initialize Result to be {}. Result is a set_of_straws.
      ii. FOR (each Straw1 in Sstraws1) DO

[1]Note that this is always well-defined, since in a straw there will always be a "last" object_dm which reaches furthest down. We just take the last monad_m of this object_dm to be the last monad_m of the straw, ignoring inner sheafs.

A. Calculate `U'` to be the restriction of `U` to begin with `Straw1.last() + 1`.
B. Calculate `Su'` to be the restruction of `Su` to begin with `Straw1.last() + 1`.
C. Initialize `i'` to be 0.
D. REPEAT `Sstraws2 := R_block_string` on `block_str1` with `U'`, `Su'`, `i'`, and the limit from `power`. IF this was not a fail, THEN add the join of `Straw1` and `Sstraws2` to `Result`. UNTIL `R_block_string` failed (i.e., there are no more matches to `block_string1` within the universe).

iii. IF (`Result` is not empty, i.e., we matched something, which was also all we could match) THEN

A. Return `Result`.

(d) Increment `i` by 1, then go once more round the WHILE loop.

5.11.9 power

5.11.9.1 syntax

```
power  :  ".."  restrictor  ;
restrictor  :  /* empty */  |  '<'  limit  ;
limit  :  INTEGER  ;  /* non-negative integer, may be 0.  */
```

5.11.9.2 semantics

```
/*
 * R_power
 * Returns : an integer which is the limit, 0 if no restrictor
 */
function R_power(power) --> INTEGER;
begin
    /* --- decide which syntactic incarnation to use --- */
    if (restrictor is empty) then
        return 0;
    else
        return limit;  /* Note: This is an abstraction of "return
                          R_limit(limit)", where R_limit simply
                          returns the integer value of limit. */
end;
```

5.11.9.3 explanation

This retrieval function simply returns the `limit` in the `restrictor`, or 0 if the `restrictor` is empty. The reason why we return 0 when the `restrictor` is empty is that “no `restrictor`” means “no limit”, which is denoted by a limit of 0.

5.11.10 block_str

5.11.10.1 syntax

```
block_str  :  object_block_first  |
              object_block_first  rest_of_block_str  ;
```

5.11.10.2 semantics

```
/*
 * R_block_str
 * Returns : a set of straws
 * ON FAIL: {}
function R_block_str(U,Su,var S,var i,limit,block_str) --> set_of_straws;
var
   mo : matched_object;
   Sm : monad; /* Start Monad */
   noretrieve : boolean;
   sos : set_of_straws;
begin
    /* --- decide which syntactic incarnation to use --- */
    while (true) do
    begin
        if (block_str = object_block_first) then
        begin
            (mo, noretrieve) := R_object_block_first(U,Su,S,
                                     i,limit,object_block_first);
            if (mo = NIL_mo) then
               return {};  /* This means that R_object_block_first
                              failed. */
            else if (not noretrieve)
               return { { mo } };  /* This means, a set of one straw
                                      A, A having only one
                                      matched_object. */
        end else
        begin /*  object_block_first  rest_of_block_str  */
             /* --- We need to keep trying successive i's because
                R_block_str should only fail if absolutely all
                possibilities within the universe have failed.  --- */
            (mo, noretrieve) := R_object_block_first(U,Su,S,
                                    i,limit,object_block_first);
```

```
if (mo = NIL_mo) then
    return {}; /* This means that
                  R_object_block_first failed. */
else
begin
    /* Calculate start monad */
    Sm := mo.last() + 1;
    sos := R_rest_of_block_str(U,Su,S,
             Sm,rest_of_block_str);

    if (noretrieve) then
    begin

       /* The noretrieve flag on the first object
          block was true.  So we should not return
          the result of R_object_block_first, but we
          should return the result of
          R_rest_of_block_str, if this is not empty.
       */

       /* --- Only return if rest_of_block_str
           matched. ---
       */
       if (sos <> {}) then
           return sos;
    end
    else
    begin
        /* Noretrieve was false, so we should
           retrieve the result of
           R_object_block_first.  We should
           only do so, however, if
           R_rest_of_block_str also returned
           something that matched.
           Otherwise, the whole expression did not
           match. */

        /* --- Only return if rest_of_block_str
            matched. ---
        */
        if (sos <> {})
            return join({ mo },sos);
    end;
end;
```

```
            end;
            i := i + 1;
        end;
    end;
```

5.11.10.3 explanation

The retrieval function `R_block_str` takes a universe `U`, a substrate `Su`, a call-by-name state `S`, a call-by-name index `i`, a monad_m limit `limit`, and the `block_str` syntactic construct that we wish to treat. It returns a set of straws. The result of `R_block_str` is one match of a `block_str`. Note that this may involve several straws.

The idea of the `R_block_str` function is that it must only return when either a match has been found, or when absolutely all possibilities have been exhausted (in which case it returns a fail, that is, an empty set of straws).

It is important to understand that `R_object_block_first` returns a pair `(matched_object,boolean)`, where the `boolean` tells whether we should *not* retrieve the `matched_object`. This pair is called an `mo_nr` pair.

This function only returns in these cases:

1. If the syntactic incarnation is just `object_block_first`:
   (a) If the call to `R_object_block_first` failed, we return the empty set of straws.
   (b) If the call to `R_object_block_first` did not fail, but returned a matched_object `mo` which we must retrieve, we return a set of straws containing one `straw` containing `mo`.
2. If the syntactic incarnation is `object_block_first` followed by `rest_of_block_str`:
   (a) If the call to `R_object_block_first` failed, we return the empty set of straws.
   (b) If the call to `R_object_block_first` did not fail, but returned a `matched_object mo`:
      i. If we are *not* to retrieve `mo`, and the call to `R_rest_of_block_str` returned a non-empty set of straws `sos`, we return `sos`.
      ii. If we are to retrieve `mo`, and the call to `R_rest_of_block_str` returned a non-empty set of straws `sos`, we return `{mo}` joined into `sos`.

If none of these cases apply, we increment `i` by one and try once more.

This function is actually quite simple, despite its cluttered look. The outline of the function is as follows:

1. WHILE (`true`) DO
   (a) IF the syntactic incarnation is just `object_block_first`:
      i. Call `R_object_block_first` to retrieve an `mo_nr` pair `(mo,noretrieve)`.
      ii. IF (the call to `R_object_block_first` failed) THEN return the empty set of straws. This is because `R_object_block_first` only returns a fail if all possibilities have been exhausted.
      iii. ELSE IF (we are to retrieve `mo`) THEN return a set of straws with a single `straw`, consisting solely of `mo`.
   (b) ELSE (i.e., the syntactic incarnation is `object_block_first rest_of_block_str`)
      i. Call `R_object_block_first` to retrieve an `mo_nr` pair `(mo,noretrieve)`.
      ii. IF (the call to `R_object_block_first` failed) THEN return the empty set of straws.
      iii. ELSE
         A. Calculate `Sm := mo.last() + 1`. `Sm` is the start monad_m at which we are to find the beginning of the first `block` in the `rest_of_block_str`.
         B. Calculate `sos` as a call to `R_rest_of_block_str` with `Sm` as the start monad.
         C. IF (we are not to retrieve the `matched_object` returned by `R_object_block_first`) AND (`sos` is not empty) THEN return `sos`. This is what we want: `R_object_block_first` did not fail, `R_rest_of_block_str` did not fail, and we must not retrieve the results of `R_object_block_first`. Thus, we return the results of `R_rest_of_block_str`.
         D. IF (we are to retrieve the `matched_object` returned by `R_object_block_first`) AND (`sos` is not empty) THEN return `{mo}` joined into `sos`. This is also what we want: `R_object_block_first` did not fail, `R_rest_of_block_str` did not fail, but returned a set of straws. So we multiply `mo` into all the straws of `sos`.
   (c) Increment `i` by 1, and try the WHILE-loop once more.

5.11.11 retrieval

The `retrieval` syntactic construct is used in `object_block_first` and `object_block`.

5.11.11.1 syntax

```
retrieval  :  /* empty */  |  "noretrieve"  |  "retrieve"  ;
```

5.11.11.2 semantics

```
/*
 * R_retrieval
 * Returns : true if "noretrieve", false otherwise.
 */
function R_retrieval(retrieval) --> boolean;
begin
    if (retrieval = "noretrieve") then
        return true;
    else
        return false;
end;
```

5.11.11.3 explanation

The idea of `retrieval` is that, if the user does not wish to retrieve an object, he or she can choose not to do so. The meaning of the result of `R_retrieval` is the answer to the question "do we want to *not* retrieve this `object_block(_first)`?". Thus this function returns "`true`" if the `noretrieve` modifier is present and false otherwise.

The default for `object_block(_first)` is to assume that we do want to retrieve the objects. This differs from Doedens' QL, where we have to explicitly mark `block`s for retrieval.

If a an `object_block(_first)` B has the `no_retrieve` modifier, no objects matched inside B can be retrieved either.

5.11.12 first_last

The `first_last` construct is used in `object_block_first`.

5.11.12.1 syntax

```
firstlast  :  /* empty */  |  "first"  |  "last"  ;
```

5.11.12.2 semantics

```
/*
 * R_first_last
 * Returns : enum_first_last
 */
function R_first_last(firstlast) --> enum_first_last;
begin
    if (firstlast = "first") then
        return first;
```

```
        else if (firstlast = "last") then
            return last;
        else
            return no_first_last;
    end;
```

5.11.12.3 explanation

The idea of "`first`" and "`last`" in an `object_block_first` is that, if a "`first`" is present, the first monad of any object matched to that `object_block_first` must be the same as the universe's first monad.

On the other hand, if "`last`" is present, the object's last monad has to be the same as the universe's last monad. If neither "`first`" nor "`last`" is present, there is no restriction.

5.11.13 hat

The function `hat` is used in `object_block_first` and `object_block`.

5.11.13.1 syntax

The syntax of hat is "O^U", where O and U are two sets of monads. O is an object and U is a universe.

5.11.13.2 semantics

```
function hat(O : object, U : set_of_monad_ms) -->
        set_of_monad_ms;
begin
    return(O.first() .. O.last());
end;
```

5.11.13.3 explanation

The O^U function is defined on p. 156 in Doedens. There, it is defined as:

> "O^U is the smallest pow_m object without gaps relative to U that O is part_of and which is itself part_of U. In other words O^U is the pow_m object which has the monads of O plus the monads of the gaps of O relative to U." (p. 156)

Here, I have simplified the definition to be simply the range O.first() .. O.last(). This is OK, since U is never empty, and O is always part_of U! This is because we only call this function if O part_of Su, which is always part_of U.

5.11.14 object_block_first

5.11.14.1 syntax

```
object_block_first  :  '['  type  retrieval  firstlast
                       features  variable_assignments  my_blocks
                       ']'  ;
type  :  IDENTIFIER
retrieval  :  /* empty */  |  "noretrieve"  |  "retrieve"  ;
firstlast  :  /* empty */  |  "first"  |  "last"  ;
```

Note that `variable_assignments` come *after* `features`. This is to keep the intuition that variables should only be used (i.e., referred to in `features`) inside the `my_blocks` of the enclosing object. It is not deemed useful to be able to assign variables and use them at the same level (i.e., in the same `object_block`), since the features can just be referred to directly.

5.11.14.2 semantics

```
/*
 * R_object_block_first
 * Returns : mo_nr (pair: (matched_object, noretrieve))
 * NOTE: The matched_object cannot be EMPTY_mo
 * ON FAIL: (NIL_mo,true)
 *
 */

function R_object_block_first(U,Su,var S,var i,limit,
            object_block_first) --> mo_nr ;
var
    O : object;
    Inner : sheaf;
    fl : enum_first_last;
    U' : set_of_monad_ms;
    inst : instances;
begin
    inst := inst(type,U);

    /* --- Calculate first or last. --- */
    fl := R_first_last(firstlast);

    while (true) do
    begin
```

```
/* --- If we have gone too far, we must fail.  --- */
if (i > length(inst)) then
    return (NIL_mo,true);

/* --- Get the object of type "type" which is the i'th
   within the universe U. --- */
O := inst[i];

/* --- If we have a limit, we must make sure it is
   enforced. --- */
if (limit <> 0) then
begin
    if (O.first() > (U.first() + limit)) then
        return (NIL_mo,true);
end;

/* --- Only do rest of checking if the first/last rules
   are obeyed. --- */
if (((fl = first) and (O.first() = U.first()))
    or
    ((fl = last) and (O.last() = U.last()))
    or (fl = no_first_last)) then
begin

    /* --- So, the first/last rules were obeyed. --- */
    if (O part_of Su) then
    begin

        /* --- So, O was part_of the substrate.  Good.
           --- */

        if (R_features(S,O,features)) then
        begin

            /* --- So, the features matched as well.
               Good. --- */

            /* --- Assign variables. --- */
            R_variable_assignments(S,O,
              variable_assignments);

            /* --- Make new universe
               (O.first()..O.last()) --- */
            U' = hat(O,U);  /* O^U */
```

```
                        /* --- Try (i.e., compute) the inner blocks.
                           ---
                        */
                        Inner := R_my_blocks(U',O,S,my_blocks);

                        /* --- Only return the object if the inner
                           did not fail. Make sure that noretrieve
                           rules are obeyed. --- */
                        if (Inner <> "// ")
                            return ([ O.id Inner ],
                                    R_retrieval(retrieval));
                    end;
                end;
            end;
            i := i + 1;
        end;
    end;
```

5.11.14.3 explanation

The retrieval function `R_object_block_first` takes a universe `U`, a substrate `Su`, a call-by-name state `S`, a call-by-name index into an inst(T,U) `i`, a monad_m-limit `limit`, and the `object_block_first` syntactic construct which we wish to treat. It returns an `mo_nr`.

An `mo_nr` is a pair `(matched_object,boolean)`, where the `boolean` tells whether we should *not* retrieve the `matched_object` into a `sheaf`. When returning a `matched_object` that arises from having found an actual object_dm, this `boolean` comes from the `retrieval` part of the syntax for an `object_block_first` (see the syntax, above).

Thus the result of `R_object_block_first` is one matching of an `object_block_first`, or a fail.

The basic idea of this function is to run through the inst(T,U) array, starting at `i`, until we either hit the end of the array or find an object. Here, T is the type specified in the `object_block_first` syntactic construct, and U is the universe `U` which we were passed as a parameter. As will be recalled, inst(T,U) is an array which gives, in order of object ordinal, all the objects of type T which are part_of the universe U.

If this `object_block_first` is the first in a `blocks`, `limit` will be 0. If, however, this `object_block_first` is nested inside a `block_string` in such a way that it comes right after a `power` syntactic construct, `limit` comes from this `power` construct. If there is no limit on the `power` construct, `limit` will be 0. If there is a limit, `limit` will be this limit.

When `limit` is non-zero, it means that there must be at most `limit` monads between the first monad of the parameter-universe `U` and the first monad of an object O matched by the `object_block_first`. When `limit` is 0, it means that no limit is enforced.

This function only returns in the following cases:

1. If we went past the end of the inst(T,U) array, we return `(NIL_mo,true)`, which means "failed match".
2. If we did find an object at some inst(T,U)[i], but the `limit` constraint was not upheld, we also return `(NIL_mo,true)`.
3. If we did find an object at some inst(T,U)[i], and a lot of constraints on this object were upheld (see below), we return the `matched_object` along with the `boolean` returned by `R_retrieval(retrieval)`.

Otherwise, we just keep incrementing `i` until one of the three cases above apply.

This function is actually quite simple, despite its cluttered look. The basic outline is as follows:

1. Calculate `inst :=` inst(`type`,`U`).
2. Calculate `fl := R_first_last(firstlast)`.
3. WHILE (`true`) DO
   (a) IF (`i` is past the length of `inst`, THEN return `(NIL_mo,true)`.
   (b) Calculate `O` := `inst[i]`.
   (c) Make sure that the rules on `limit` are obeyed. If they are not, return `(NIL_mo,true)`.
   (d) IF (the rules on `firstlast` are obeyed) AND (`O` part_of `Su`) AND (the features match) THEN
      i. Call `R_variable_assignments` so that we assign variables with the features of this `O` (see section 5.11.23 on page 96).
      ii. Calculate `U' := hat(O,U)`. (See section 5.11.13 on page 79.)
      iii. Calculate the inner `blocks` by calling `R_my_blocks` with `U'` as the universe, `O` as the substrate, `S` as the state, and `my_blocks` as the syntactic construct.
      iv. IF (this Inner `blocks` was not a failed match, return a `matched_object` consisting of the object id_d of `O` along with the Inner `sheaf`. This is, of course, paired with `R_retrieval(retrieval)`, since we are returning an `mo_nr` rather than a `matched_object`.
   (e) Increment `i` by 1 and take one more pass through the WHILE loop.

5.11.15 my_blocks

5.11.15.1 syntax

```
my_blocks  :  /* empty */  |  blocks  ;
```

5.11.15.2 semantics

```
/*
 * R_my_blocks
 * Returns: sheaf
 *          For my_blocks being empty: // {} (not a fail)
 *          For my_blocks being a blocks:  The sheaf for the blocks
 */

function R_my_blocks(U,Su,var S,my_blocks) --> sheaf;
begin
    /* --- decide which syntactic incarnation to use --- */
    if (my_blocks = blocks) then
        return R_blocks(U,Su,S,blocks);
    else
        return "// {}";
end;
```

5.11.15.3 explanation

As will be recalled from the grammar in section 5.9.3 on page 58, `my_blocks` represents the inner `blocks` in an `object_block` or `object_block_first`. The reason why we have a special nonterminal and not just a `blocks` is, of course, that we want it to be able to be empty. When empty, we should just return the least sheaf which is not a failed match (i.e., "`// {}`"). When non-empty, we should return whatever the retrieveal function on the inner `blocks` returns (see above). `R_my_blocks` does just that.

5.11.16 rest_of_block_str

5.11.16.1 syntax

```
rest_of_block_str  :  block  |  block1  rest_of_block_str1  ;
```

NOTE: The 1's on the second usage of `block` and `rest_of_block_str` are just there for identification purposes.

5.11.16.2 semantics

```
/*
```

```
 * R_rest_of_block_str
 * Returns : set_of_straws
 * ON FAIL: {}
 */
function R_rest_of_block_str(U,Su,var S,Sm,rest_of_block_str) -->
         set_of_straws;
var
    noretrieve : boolean;
    mo : matched_object;
    rest : set_of_straws;
    Sm' : monad; /* --- Start Monad. --- */
    Result : set_of_straws;
    somn : set_of_mo_nr;
begin
    /* --- decide which syntactic incarnation to use --- */
    if (rest_of_block_str = block) then
    begin
        /* --- Get the result of R_block. --- */
        somn := R_block(U,Su,S,Sm,block);

        if (somn has NIL_mo)
            return {};  /* --- This means that block was an
                               object_block, and that the match
                               failed.
                               There will only be one (NIL_mo, true)
                               in the somn set if it failed.  --- */


        /* --- Make somn into the set straws, each straw consisting
           of those matched objects which are in the set_of_mo_nr. --- */

        Result := {};
        for (each (mo,noretrieve) in somn) do
        begin
            /* --- Note: We will always get here, since the match
               did not fail. --- */
            if (not noretrieve) then
                Result := Result "union" { { mo } };
        end;
        return Result;

    end else /* ---  block1  rest_of_block_str1  --- */
    begin
        /* --- Get first block. --- */
```

```
somn := R_block(U,Su,S,Sm,block1);

/* --- If it failed, return {}.  See the counterpart
   above for further explanations. --- */
if (somn has NIL_mo) then
    return {};

/* --- Calculate the rest. --- */
Result := {}
for (each (mo,noretrieve) in somn) do
begin
    /* --- Note:  We will always get here, since the
       match did not fail. --- */

    /* --- EMPTY_mo is there if we matched an
       opt_gap_block which had no gap there.
       In this case, the somn will consist
       _only_ of the single element
       (EMPTY_mo, true). ---  */

    /* --- Calculate Start Monad. --- */
    if (mo = EMPTY_mo) then
        Sm' := Sm;
    else
        Sm' := mo.last() + 1;

    /* --- Get the rest. This is a recursive call to
       ourselves. ---  */
    rest := R_rest_of_block_str(U,Su,S,
              Sm',rest_of_block_str1);

    /* --- Only add if rest matched! --- */
    if (rest <> {}) then
    begin
        /* --- Only add mo if noretrieve was false.
           --- */
        /* --- In both cases, add the rest. */
        if (noretrieve) then
            Result := Result "union" rest;
        else
            Result := Result "union" join({ mo }, rest);
        end;
    end;
end;
```

```
            return Result;
        end;
    end;
```

5.11.16.3 explanation

The `R_rest_of_block_str` retrieval function takes a universe `U`, a substrate `Su`, a call-by-name state `S`, a Start Monad_m `Sm`, and the `rest_of_block_str` syntactic construct which we wish to treat. It returns a set of straws. The result of `R_rest_of_block_str` is one matching of a `rest_of_block_str`. Note that this may involve several straws.

A construct which is crucial in understanding this function is the `set_of_mo_nr`'s. It consists of a set of `mo_nr`'s. An `mo_nr` is a pair, `(matched_object,boolean)`, where the `boolean` is intended to convey information on "do we *not* want to retrieve this `matched_object`?". The `matched_object` in the pair can only be one of these four things:

1. A `NIL_mo`, which means that a subsequent call to `R_object_block` failed.
2. An `EMPTY_mo`, which means that we tried to match an `opt_gap_block` against an empty gap (i.e., there was no gap). This is not a fail.
3. An `[ O.id Inner-Sheaf ] matched_object`, where the `O.id` is an object id_d of an object_dm.
4. An `[ object id_m // {} ] matched_object`, where the object id_m refers to a pow_m object of a gap we wished to match with an `opt_gap_block`.

In the first two cases, the boolean is always "`true`", meaning that we do not wish to retrieve the `matched_object`. This is very significant in this function, as it means that neither of these two will get into any sheaf. In the two last cases, the boolean may be "`false`" or "`true`" depending on the "`noretrieve`" or "`retrieve`" keywords used (or not used) in the `opt_gap_block` or `object_block`.

The outline of the function is not that difficult to understand. It just looks cluttered in the code. The outline is as follows:

1. Deciding syntactic incarnation, if this is the first incarnation, namely `block`, do the following:
   (a) Set `somn :=` the result of a call to `R_block` on the `block`.
   (b) IF (the `somn` is a singleton set with a `(NIL_mo,true)` as the element) THEN return `{}`. This is because we failed.
   (c) If not, initialize `Result` to be `{}`.

(d) FOR each `(mo,noretrieve)` in `somn`, DO

i. IF the `noretrieve` is `false` (i.e., we do want to retrieve the object), THEN

A. make a new straw consisting solely of `mo` and add the new straw to `Result`. (Note that, because of the IF-clause, this only happens to matched_objects of type 3 and 4 above)

(e) Return `Result`.

2. If the syntactic incarnation is "`block1  rest_of_block_str1`", do the following:

(a) Set `somn` := the result of a call to `R_block` on the `block1`.

(b) IF (the `somn` is a singleton set with a `(NIL_mo,true)` as the element) THEN return `{}`.

(c) If not, initialize `Result` to be `{}`.

(d) FOR each `(mo,noretrieve)` in `somn`, DO (note that only `matched_objects` of type 3 and 4 above make it into the `Result` set of straws)

i. IF (`mo` is `EMPTY_mo`) THEN

A. Initialize `Sm'` to be `Sm`

ii. ELSE

A. Initialize `Sm'` to be `mo.last() + 1`[2]

iii. Set `rest` to be the result of a recursive call to `R_rest_of_block_str`, with the parameters the same as our formal parameters, except use `Sm'` in place of `Sm`.

iv. IF (`rest` was not empty) THEN

A. IF (`noretrieve` for the `mo_nr` in the FOR-clause was true) THEN add `rest` to `Result`, but do not add `mo`.

B. ELSE add the result of `join({mo},rest)` to `Result` (see below)

(e) Return `Result`.

The significance of the `join` is that it "multiplies" the straw "`{mo}`" into all the other straws in `rest` (see section 5.11.7 on page 69 for the definition of `join`). This is what we want, since it captures the idea that each `block` should be in a straw with all the other `blocks` (and `object_block_firsts`) with which it goes in the query.

Note that, in the second syntactic incarnation, the result is non-empty only if both the call to R_blocks and the recursive call to R_rest_of_block_str did not fail. This is because the whole rest_of_block_str must be present for the function not to fail.

[2] Where `mo.last()` refers to the last monad_m of the object denoted by the object id_d in the `matched_object` or the last monad_m of the object id_m in the `matched_object`.

5.11.17 block

5.11.17.1 syntax

```
block : opt_gap_block | object_block ;
```

5.11.17.2 semantics

```
/*
 * R_block
 * Returns: set_of_mo_nr
 * ON FAIL: { (NIL_mo,true) }
 *
 */
function R_block(U,Su,var S,Sm,block) --> set_of_mo_nr
begin
    if (block = opt_gap_block)
        return R_opt_gap_block(U,Su,Sm,opt_gap_block);
    else
        return R_object_block(U,Su,S,Sm,object_block);
end;
```

5.11.17.3 explanation

This function takes a universe `U`, a substrate `Su`, a call-by-name state `S`, a Start Monad_m `Sm`, and the `block` syntactic construct which we must treat. It returns a `set_of_mo_nr`, which is a set of `mo_nr`'s. An `mo_nr` is a pair `(matched_object,boolean)`, where the `boolean` indicates whether we do *not* want to retrieve the `matched_object`.

The function just decides syntactic incarnation and calls the relevant retrieval function.

The result of a `block` is either:

1. For `obt_gap_block`:

   (a) An object id_m of a pow_m object indicating a *gap* starting at monad `Sm`, paired with information about whether to retrieve the gap (the decision about whether to include them in a straw is made higher up in the hierarchy), OR

   (b) an `EMPTY_mo matched_object` paired with the information that we *do not* want to retrieve it, OR

2. For `object_block`:

(a) A set of `matched_objects` starting at monad `Sm`, paired with information about whether we do not want to retrieve them (as with `opt_gap_block`, the decision about whether to include them in a straw is made higher up in the hierarchy), OR

(b) A `NIL_mo` `matched_object` paired with the information that we do not want to retrieve it. A `NIL_mo` `matched_object` means "failed match".

5.11.18 opt_retrieval

The `opt_retrieval` syntactic construct is used in `opt_gap_block`.

5.11.18.1 syntax

```
opt_retrieval  :  /* empty */  |  "noretrieve"  |  "retrieve"  ;
```

5.11.18.2 semantics

```
/*
 * R_opt_retrieval
 * Returns:  false on must retrieve, true otherwise
 */
function R_opt_retrieval(opt_retrieval) --> boolean;
begin
    if (opt_retrieval = "retrieve") then
        return false;
    else
        return true;
end;
```

5.11.18.3 explanation

The `opt_retrieval` syntactic construct is used in the `opt_gap_block` syntactic construct. It is used to specify explicitly or implicitly whether the user wants to retrieve the optional gap or not. The default, when empty, is to assume that we do not want to retrieve the optional gap. Thus, if the user wants to retrieve the gap, they must explicitly write "`retrieve`".

The reason why we return "`false`" on "`retrieve`" and "`true`" otherwise is that the `boolean` really answers the question "do we *not* want to retrieve this gap?".

Note that this has different semantics from `retrieval` for the default. Here, the default is to assume that we do *not* want to retrieve the gap. For `retrieval`, the default is to assume that we *do* want to retrieve the `object_block(_first)`.

5.11.19 opt_gap_block

5.11.19.1 syntax

```
opt_gap_block  :  '['  "gap?"  opt_retrieval  ']'  ;
opt_retrieval  :  /* empty */  |  "noretrieve"  |  "retrieve"  ;
```

5.11.19.2 semantics

```
/*
 * R_opt_gap_block
 * Returns: (matched_object, noretrieve)
 * ON FAIL: Cannot fail.  Returns (EMPTY_mo,true) if there is no gap.
 *
 */
function R_opt_gap_block(U,Su,Sm,opt_gap_block) --> set_of_mo_nr ;
begin
    if (there exists monad m such that Sm..m is a gap in Su
        with respect to U) then
        return { ( [ pow_m(Sm..m) // {} ],
                     R_opt_retrieval(opt_retrieval)) };
    end
        return { (EMPTY_mo,true) };
end;
```

5.11.19.3 explanation

This function takes a universe `U`, a substrate `Su`, a Start Monad `Sm`, and the `opt_gap_block` syntactic construct which is to be treated. It returns a `set_of_mo_nr`, which is a set of `mo_nr`'s. An `mo_nr` is a pair `(matched_object,boolean)`, where the `boolean` part is inteded to mean "do we *not* want to retrieve this `matched_object`?". The result is one matching of an `opt_gap_block`.

The function always returns a singleton set. The reason why, then, we choose to return a `set_of_mo_nr` rather than an `mo_nr`, is that it seems more elegant: The `R_opt_gap_block` function is used in `R_block` in the same way as the `R_object_block` function, which returns a `set_of_mo_nr`.

If there is no gap in `Su` with respect to `U` which starts at `Sm`, we return `{ (EMPTY_mo,true) }`.

If, on the other hand, such a gap starting at `Sm` exists, we return a `matched_object` paired with the result of `R_opt_retrieval(opt_retrieval)`. The `matched_object` consists of the object id_m of a pow_m object consisting of the gap, coupled with the least `sheaf` which is not a failed retrieval ("`// {}`").

Since `U` is always a universe (see our definition in section 5.4.1 on page 48), we can calculate the existence of gaps in `Su` with respect to `U` from `Su` alone. Thus we need not even inspect the database, but can just look at Su. This should be easy, given the implementation of a set of monad_m's suggested in section 4.2 on page 32.

Note that this function cannot fail. Instead, when there is no gap, we return "{ (EMPTY_mo,true) }".

5.11.20 last

The last syntactic construct is used in object_block, which will be treated below.

5.11.20.1 syntax

```
last  :  /* empty */  |  "last"  ;
```

5.11.20.2 semantics

```
/*
 * R_last
 * Returns: a member of enum_first_last:
 *          - last if "last"
 *          - no_first_last otherwise
 */
function R_last(last) --> enum_first_last;
begin
    /* --- decide which syntactic incarnation to use --- */
    if (last = "last") then
        return last;
    else
        return no_first_last;
end;
```

5.11.20.3 explanation

The idea of the last syntactic construct has been explained under the firstlast syntactic construct in section 5.11.12.3 on page 79.

5.11.21 object_block

5.11.21.1 syntax

```
object_block  :  '['  type  retrieval  last
                 features  variable_assignments  my_blocks  ']'
                 ;
type  :  IDENTIFIER
retrieval  :  /* empty */  |  "noretrieve"  |  "retrieve"  ;
```

```
last  :  /* empty */  |  "last"  ;
```

5.11.21.2 semantics

```
/*
 * R_object_block
 * Returns: set_of_mo_nr
 * ON FAIL: { (NIL_mo,true) }
 * On success:  There are no (NIL_mo,true) in the set_of_mo_nr!
 *
 */
function R_object_block(U,Su,var S,Sm,object_block) -->
         set_of_mo_nr ;
var
    O : object;
    Inner : sheaf;
    fl : enum_first_last;
    U' : set_of_monad_ms;
    Straw1 : straw;
    Result : set_of_mo_nr;
begin
    /* --- Initialize Result. --- */
    Result := {};

    /* --- Get all O's beginning at Sm (Start Monad). --- */
    Straw1 := (all O of type "type" such that O.first() = Sm);
    for (each O in Straw1) do
    begin
        /* --- Note that we might not get here, since Straw1
           might be empty. --- */

        /* --- Calculate last/no_first_last.  It is deemed better
           to have it here rather than outside the "for" loop,
           since here it is only executed O(n) times, where n is
           the number of O's beginning at Sm.  If we put it
           outside, we will execute it _every_ time we call
           R_object_block, which is deemed to be more times than
           if we place it here.  There might not be any objects
           starting at Sm! --- */

        fl := R_last(object_block);

        /* --- Make sure that the last/no_first_last rule is
           obeyed. --- */
```

```
        if (((fl = last) and (O.last() = U.last()))
            or (fl = no_first_last)) then
        begin
            /* --- So, the last/no_first_last rule was obeyed.
               --- */
            if (O part_of Su) then
            begin
                /* --- So, we are part_of Su. --- */
                if (R_features(S,O,features)) then
                begin
                    /* --- So, the features matched. --- */

                    /* --- Assign variables. --- */
                    R_variable_assignments(S,O,
                      variable_assignments);

                    /* --- Make new universe to be
                       O.first()..O.last(). --- */
                    U' = hat(O,U);  /* O^U */

                    /* --- Calculate inner. --- */
                    Inner := R_my_blocks(U',O,S,my_blocks);

                    /* --- Only add object if Inner was a
                       success. --- */
                    if (Inner <> "// ")
                        Result := Result "union"
                                  { ( [ O.id Inner ],
                                      R_retrieval(retrieval) ) };
                end;
            end;
        end;
    end;

    if (Result = {})
        return { (NIL_mo,true) };
    else
        return Result;
end;
```

5.11.21.3 explanation

The `R_object_block` function takes a universe `U`, a substrate `Su`, a call-by-name state `S`, a start-monad_m `Sm`, and the `object_block` syntactic construct for treatment. It returns a set of pairs of `(matched_object,boolean)` (`set_of_mo_nr`), whose intended meaning is, "a `matched_object` paired with a `boolean` being "`true`" if we are *not* to retrieve this `matched_object`".

The function returns the `set_of_mo_nr` corresponding to all those objects of type type which start at `Sm`, and for which a number of constraints hold. The reason why we return a `set_of_mo_nr`. rather than a single `mo_nr`, is, of course, that more than one object of type `type` may start at `Sm`.

On failure, the function returns the singleton set `{ (NIL_mo,true) }`.

The `Sm` parameter is a "Start Monad" parameter. It is used to get all those objects of the given type which start at the given monad. *This is why it is handy to store in a `monad_d` a list of those objects which start at a given monad_m.*

This function is actually quite simple, despite its cluttered look. The steps are as follows:

1. Initialize `Result` to be the empty set of `matched_objects` (a `straw`).

2. Get in `straw Straw1` all those objects of type `type` which start at `Sm`. This is done by inspection of the monad_d corresponding to `Sm`.

3. For each `O` in `Straw1`, do the following:

    (a) IF (the rules about `last` are obeyed) AND (`O` part_of `Su`) AND (the features all match) THEN

        i. Make variable assignments (call `R_variable_assignments`).
        ii. Calculate `U'` = `O^U` (see section 5.11.13 on page 79)
        iii. Calculate the result of the inner `my_blocks` (call `R_my_blocks` with `U'` as the universe and `O` as the substrate)
        iv. IF (the call to `R_my_blocks` was a success) THEN add the matched object "`[ O.id Inner ]`" paired with the result of `R_retrieval(retrieval)` to `Result`.

4. If `Result` is empty, return `{ (NIL_mo,true) }`.

5. Else, return `Result`.

Note that the "union" when adding the new object can be just a concatenation to a list, since we are sure that the new member is not there.

5.11.22 features

5.11.22.1 syntax

```
features   :  /* empty */  |  ffeatures  ';'  ;
ffeatures  :  fterm  |  ffeatures  "or"  fterm  ;
```

```
fterm  :  ffactor  |  fterm  "and"  ffactor  ;
ffactor  :  "not" ffactor  |  '('  ffeatures  ')'  |  feature  ;
feature  :  feature_name  '='  value  ;
feature_name  :  IDENTIFIER  ;
value  :  enum_val  |  INTEGER  |  STRING  |  variable  ;
enum_val  :  IDENITIFIER  ;
```

5.11.22.2 semantics

```
/*
 * R_features
 * Returns : nothing
 */
function R_features(S,O,features) --> boolean;
begin
  if features empty, then
      return true.
  else,
      traverse tree represented by features,
      deciding truth-value with respect to O.
      Return the result.
end;
```

5.11.22.3 explanation

The `R_features` function takes a call-by-*value* state `S`, an object `O`, and the `features` syntactic construct which we wish to treat. It returns a `boolean`. The result of `R_features` is the answer to the question, "do the feature-constraints in `features` match with the features of the object `O`, given the state `S`?".

The `ffeatures` syntactic construct is assumed to be represented as an Abstract Syntax Tree (AST). We can then traverse the tree in depth-first order, deciding truth-value recursively. We may use short-circuit evaluation, since there are no side-effects.

5.11.23 variable_assignments

5.11.23.1 syntax

```
variable_assignments  :  variable_assignment  |
                         variable_assignments
                         variable_assignment  ;
variable_assignment  :  variable  ":="  feature_name  ';'  ;
```

5.11.23.2 semantics

```
/*
 * R_variable_assignments
 * Returns : nothing
 * NOTE: updates S
 */
function R_variable_assignments(var S,O,variable_assignments);
begin
  for (each variable_assignment "r := feature_name") do
     S.r := O.feature_name();
end;
```

5.11.23.3 explanation

The `R_variable_assignments` retrieval function takes a call-by-name state `S`, an object `O`, and the `variable_assignments` syntactic construct which we wish to treat. It returns nothing, only updating `S`.

The function updates the state, `S`, in such a way that all variables in `variable_assignments` get assigned the value of their corresponding feature, taken on `O`.

5.12 Conclusion

In this chapter, we have given most of the framework or design for a query engine for EMdF databases, as embodied in the syntax and semantics of MQL. Even though some concepts were "stolen" from Doedens, most of the chapter has been original.

Among the major points of this chapter can be mentioned the description and explanation of the sheaf, and the description and explanation of the MQL grammar and MQL variables. The majority of the chapter has dealt with an operational, syntax-driven specification of MQL.

As it stands, an MQL query engine should be relatively straightforward to implement.

Chapter 6

Conclusion

6.1 Conclusion

This bachelor thesis has been about one approach to text databases, namely a modification and extension of the MdF model by Crist-Jan Doedens. We have detailed and explained the MdF model as given by Doedens in his PhD dissertation. We have built a framework of concepts on top of the MdF model, obtaining the EMdF model. We have given many hints on how to implement the EMdF model on top of an actual OODBMS. Lastly, we have developed, motivated, and explained a query language for EMdF databases, MQL.