Query and Bulk Type Extensions in Higher-Order Polymorphic Languages

Datenbankerweiterungen von polymorphen Sprachen höherer Ordnung

Diplomarbeit

vorgelegt
von
Dominic M. Juhász

März 1994

Betreuer:
Prof. Dr. Joachim W. Schmidt
Prof. Dr. Heinz Züllinghoven

Fachbereich Informatik
Universität Hamburg
Vogt-Kölln-Str. 30
22527 Hamburg

Contents

Chapter 1

Introduction and Motivation

A database system supports a database language consisting of a data definition language (DDL) for the definition of schemata and a data manipulation language (DML) for the access and manipulation of data stored in the database [Lockemann, Schmidt 87; Ullman 88a]. This scenario is sufficient for the formulation of ad-hoc queries since the DML includes a query language. Examples of database languages are SQL [Chamberlin, et al. 76], QUEL [Stonebraker et al. 76], POSTQUEL [Stonebraker, Rowe 86], and QBE [Zloof 77]. In order to enable a further processing of query results, an embedding of the query language into a programming language is necessary. The embedding of query languages into host languages is indivisible coupled with the embedding of the bulk types representing the database.

A widely utilized approach for embedding a query language into a given host language is the use of language preprocessors [Lorie, Wade 79]. Commands of the DML are syntactically more or less smoothly integrated into programs written in the host language. A preprocessor extracts these commands of the DML and processes them separately. Examples of database systems with this form of embedding are System/R (SQL into PL/I and Cobol) [Astrahan et al. 76], Ingres (QUEL into PL/I, Fortran, Cobol, Basic, and C) [Zook et al. 77; Ingres 89], and Oracle (SQL into Fortran, Cobol, and C) [Oracle 91].

Database programming languages represent an alternative approach to embedding. A conventional, Turing complete programming language is extended with bulk types for the persistent storage of the bulk data and database functionality as, for example, declarative constructs for the access of data and for iteration over data elements, and a transaction concept. The syntax, semantics, and implementation of the bulk type and the support for the database functionality are built into the language processor and the run-time system support. Database programming languages achieve a real integration of a database language into a programming language by eliminating the conceptual difference between the constructs of the database language and the programming language. An early representative of those languages is Pascal/R [Schmidt 77] extending Pascal [Wirth 71] and later evolved to Modula/R [Reimer, Diener 83; Koch et al. 83] and DBPL [Schmidt, Matthes 92] on the basis of Modula-2 [ModISO 91]. Other examples are Adaplex [Smith et al. 83], extending Ada [Ichbiah 83] with a data model DAPLEX [Adrion, Branstad 81], and Galileo [Albano et al. 85].

The development of programming languages as *Quest* [Cardelli 89], TL [Matthes 92b], ML [Paulson 91], Modula-3 [Nelson 91], and Eiffel [Meyer 92] supporting modern programming concepts as parametric polymorphism and higher-order functions discloses new opportunities. Bulk types are definable by the user employing parameterized type constructors; there is no need for built-in bulk types. If the language also supports an orthogonal persistency concept, the user-defined bulk types may be employed for data-intensive applications. In such an environment support for declarative access constructs for bulk types allowing iteration abstraction and the formulation of queries is desirable for all user-defined bulk types.

A possible solution is a library-based approach (add-on approach). The database functionality is supported by libraries as proposed in [Matthes, Schmidt 92; Niederée 92]. Query support and iteration abstraction are realized by higher-order functions. This approach is very flexible and easy to extend, but it lacks declarativity and a user-friendly syntax as found in systems following a built-in approach.

It is the aim of this work to combine the user-friendly syntax and declarativity of a built-in approach with the flexibility and extensibility of an add-on approach.

This thesis is concerned with the development of an extensible framework for a typed query language. In order to retain the flexibility of an add-on approach, the query language should not be built into a host language. It is attempted to support the syntax of the query language by syntax extension technology [Kohlbecker 86], i.e., by translating syntax specific to the query language into expressions of a host language. This work examines the feasibility of such an approach in a typed environment and investigates its flexibility and its extensibility to arbitrary bulk types.

1.1 Design Goals

It is the aim of this thesis to design a framework for a query language in a typed environment. The design of the query language and the choice of the supporting technology are driven by the goal to achieve extensibility, uniformity, flexibility, and optimizability. Comprehensions are chosen to form the basis of the query language. They promise to help achieving the four design goals [Trinder 92].

In order to improve the extensibility and flexibility of the approach, the comprehensions are augmented by an additional building block, called *context*. A comprehension together with a context forms a query. Intuitively speaking, the context defines a last processing step for the sequence of elements specified by the comprehension. A similar approach is found in the database programming language DBPL [Schmidt, Matthes 92]. DBPL supports *selective* and *constructive access expressions* allowing declarative formulation of query expressions on relations. The access expressions of DBPL are typed first-class language constructs that may be named and parameterized. It is possible to employ the access expressions in different contexts as in relation constructors, resulting in facilities for the definition of new relations, subrelations and views, and in *for*-loops, allowing the iteration over selected elements of a relation applying an operation to each of the elements. Another application of the access expressions is the update of selected subrelations.

The main ideas for the realization of the four design goals are summarized in the following parts of this section.

1.1.1 Extensibility

The proposed query language defines an extensible framework for the formulation of queries. The kernel of the language is based on comprehensions, a widely accepted notation for list manipulations in functional programming languages [Turner 82; Bird, Wadler 88; Field, Harrison 88; Peyton-Jones 87; Turner 87]. Rather than using the comprehensions directly as queries, as proposed in [Trinder 89; Trinder 92], they are augmented by the concept of *contexts*. Only a comprehension combined with a context forms a complete query.

Concepts similar to the *contexts* introduced here are implicitly present in many query languages. Examples are the application of aggregate functions to the query result [Bültzingsloe wen 87] or the choice of one/the only element of the query result supported for example in Fibonacci [Albano et al. 93], O_2SQL [Bancilhon et al. 92], and ADAPLEX [Smith et al. 81]. The specification of these "contexts" fixes a last processing step to be performed for the elements of the query result. In absence of special options for the context, the elements of the query result are collected in a bulk type (e.g., tables in SQL [Date 89]). This may be viewed as an implicit standard context for queries. The presented query language generalizes these ideas, and employs the notion of contexts explicitly.

The query language does not support a fixed set of contexts. However it offers an extensible framework for the definition and use of contexts. Concrete contexts are supported by an extensible library. Thus, special contexts may be defined according to the considered applications.

The designed query language is not restricted to one specific bulk type, as it is the case in most query languages and query facilities in database programming languages based on a built-in approach (Pascal/R [Schmidt 77], Plain [Wasserman et al. 81], Modula/R [Reimer, Diener 83; Koch et al. 83], DBPL [Schmidt, Matthes 92], E [Carey et al. 88], CO_2 [Lécluse, Richard 89]). The bulk types are not built-in. They are provided by a library that is extensible by the user. The query language is applicable to all those bulk types[1].

In addition, arbitrary element types are possible for the bulk types. The element types are not restricted in any way as, for example, to flat tuples in the relational model. The query language is independent of a specific data model. It is applicable in the context of flat relations as well as in the presence of complex objects. Support specific to a chosen data model, e.g., in the form of special contexts for the comprehensions can be provided by the libraries that are part of the environment for the query language.

1.1.2 Uniformity

A prerequisite for the implementation of a query language are access primitives for the bulk types that allow sequential iteration over the elements. A query language for an extensible set of bulk types requires a uniform protocol for these access primitives in order to enable a uniform implementation of the query language. In [Matthes, Schmidt 92] a protocol for access primitives described by an iterator type is proposed. Values of this type represent an iteration over the elements of a bulk type. The protocol supports three functions: a function *empty* to test if the iteration is empty, a function *get* to yield the first element of the iteration, and a function *rest* yielding another value of the iterator type representing the iteration over the

[1] Assuming they fulfil some implementation-dependent restrictions.

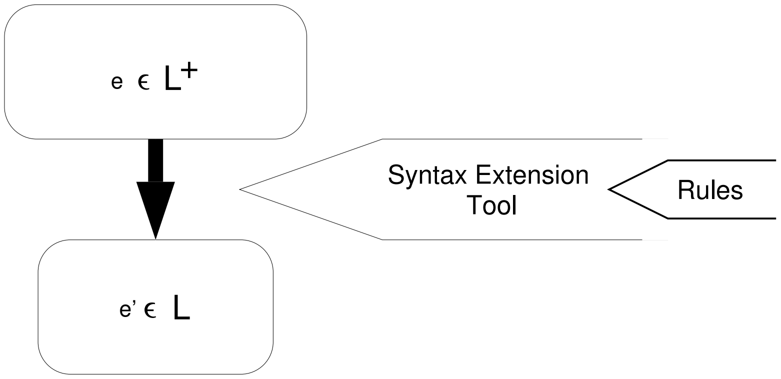

Figure 1.1: Flexible Language Extension

rest of the elements. The access primitives are supported for each bulk type by mapping it to a value of the iterator type. Functions for iteration abstraction and the definition of a query language for arbitrary bulk types may then be based on these access primitives.

A drawback of this approach is that for each step of iteration a set of three new function closures are produced resulting in unnecessary time and space overhead during the iteration of a bulk type [Liskov et al. 77]. It is more efficient to model the iterators by an object with a state. The state keeps track of the progress of the iteration. This approach comes close to scanning through data known from streams [Landin 64] in data processing.

On the basis of such a protocol a uniform query language for arbitrary user-defined bulk types is realized. It is even possible to specify different bulk types as an input to a single query. This enables the formulation of *mixed queries*, a novel concept to comprehension-based query languages. Additionally several forms of nested queries [Dayal 87] are possible in the presented framework. Queries may be employed as subqueries in all building blocks of this language. This feature is of special importance for complex object models.

1.1.3 Flexibility

The syntax of the query language is not built-in. It is provided by employing syntax extension technology [Cardelli 93; Kohlbecker 86]. This technology allows the introduction of application-oriented syntax into an existing language L. This leads to a new language $L+$, accepting valid expressions of the language L as well as constructs containing the additionally introduced syntax. Rules for the translation of the new syntax into constructs of the underlying language have to be provided. Figure 1.1 illustrates this approach. Expressions of the language $L+$ are mapped to expressions of the language L by the syntax extension tool. The syntax extension tool is driven by rules. The rules specify how expressions containing newly introduced syntax are implemented employing the language L. An advantage of this approach is that the syntax is adaptable to new requirements without touching (changing) the compiler.

So the chosen syntax for the query language is not fixed for all times; it can be adapted according to experiences made with the syntax and according to new requirements. It is also possible to change the implementation of the query language by developing new rules for the syntax extension tool. This might be necessary to improve the performance of the system.

As mentioned above the bulk type support for the query language is not built-in. It is provided by libraries. This introduces further flexibility to the environment of the query language: the implementation of the bulk types can be altered in order to fulfil new requirements towards the functionality and efficiency of the query evaluation.

1.1.4 Optimizability

The developed query language and its implementation have a prototypic character. It is the aim of this work to examine the feasibility of the chosen approach. For this reason it does not focus on efficiency and optimization. On the other hand these topics are very important for query languages. It is, therefore, notable that the query language offers promising starting points for optimizations.

Optimization rules for comprehensions, forming the kernel of the query language, are described in literature [Trinder, Wadler 89]. Further starting points for optimizations are the implementations of the query language and of the bulk types. Both are not built-in. The bulk types are provided by libraries that may be extended and changed by the user. Existing implementations are exchangeable by more efficient ones. The implementation of the query language is defined by the rules for the syntax extension tool. Other and probably more efficient implementations can be established by developing new rules.

The query language is realized employing syntax extension technology. Expressions containing query-specific syntax are transformed into code of the Tycoon language, or more precisely into abstract syntax trees of the language TL. After this transformation queries are subject to the code optimization performed by the Tycoon compiler. It has to be examined which options to the compiler improves the optimization of code typical for the query implementation.

1.2 Outline of the Text

In this chapter design goals for the developed query language together with some first ideas for the realization of the approach are presented. The rest of the thesis is split into five chapters.

The query language presented in this thesis is comprehension-based. Chapter 2 describes the origins of the comprehensions and the different application of this notation. One of the most significant application areas is the use of list comprehensions in functional languages. Comprehensions are proposed as query language in [Trinder 92; Abiteboul, Beeri 88]. The concepts of the comprehensions can be extended to different bulk types. Different approaches towards the notion of a bulk type are known from literature. Some of these approaches are summarized in chapter 2. First, attempts to give an intuitive semantics of bulk types are described followed by a more formal treatment of the notion of a bulk type. Among these approaches one can identify the monadic [Wadler 90; Trinder 92] and abstract data type [Beeri, Ta-Shma 94] definition of bulk types.

The query language developed in this thesis is embedded into a host language. This embedding is achieved by employing syntax extension technology. The language TL [Matthes 92b], a modern programming language supporting concepts as parametric polymorphism and higher-order functions, is used as host language for the query language and also as target language for the syntax extension. Relevant concepts of the language TL are described in chapter 3. Additionally a short overview of the employed syntax extension tool is given in this chapter. Both the language TL [Matthes 92a] and the employed syntax extension tool [Schröder 93] have been developed at the University of Hamburg.

Chapter 4 describes the designed query language. The query language consists of two parts: the kernel of the query language and an extensible framework for contexts. The kernel is mainly formed by comprehensions that accept arbitrary bulk types as input. First, the building blocks of the kernel of the query language and the proposed syntax for this part of the language is introduced. These considerations are completed by the presentation of a grammar. Furthermore, typing and scoping aspects are discussed.

Intuitively, contexts fix the last processing step for the sequence of elements specified by the comprehension that forms the kernel of the query. The query language provides an extensible framework for the definition of contexts. This framework together with the resulting extensions of the grammar of the query language are presented in chapter 4. The concrete contexts are supported by a library that is extendible and adaptable by the user. In order to give an impression of the wide variety of possible contexts a concrete set of representative contexts is described. It is a novelty of the developed query language that it supports *mixed queries*, i.e., different bulk types can be specified as ranges to a single query. Based on the theoretical results presented in chapter 2, a semantic for mixed queries is discussed in chapter 4. The last part of the chapter is devoted to more advanced aspects of the query language. First it is shown that naming and parameterization can be naturally introduced into the query language. Second an approach for the integration of recursive queries into the presented framework is outlined.

The query language presented in this thesis is realized employing syntax extension technology. Thus expressions containing query language syntax have to be implemented by expressions of the host language (here TL). This implementation fixes the operational semantics of the query language. Prior to the discussion of different implementation alternatives, an architecture for the extensible query language environment is presented in the first part of chapter 5. The implementation of the query language consists of a realization of the comprehensions and of a realization of the framework for the contexts. It is the design goal to keep the implementation of the comprehensions independent of the context in which they may appear. The environment for the query language and its implementation is constituted by libraries supporting bulk types, contexts, and iteration abstraction.

First, a functional implementation is examined. It is based on a translation scheme for list comprehensions presented in [Wadler 87]. The translation scheme is extended to work for arbitrary bulk types as input of the query and especially also for mixed queries. This is achieved employing a functional implementation of iteration abstraction. Furthermore the approach is generalized allowing the implementation of comprehensions in different contexts. For this reason a generic function *reduce* is introduced that is parameterized by functions in dependence of the chosen context. A further generalization leads to a generator for reduce functions. In accordance with approaches proposed in literature the employed bulk types are implementations of ringads [Trinder 92].

In the third part of the chapter an imperative implementation of the query language is described. The range variables of the comprehensions are realized by mutable variables. The elements of the according ranges of the comprehension are subsequently assigned to these variables resulting in a sequence of environments, called binding sequences or comprehension threads, for the evaluation specified in the target expression of the comprehension. Iteration abstraction and bulk types are realized by objects with a mutable state in this approach.

The first part of chapter 6 gives a summary of the results of the work. The efficiency of the presented query language depends on the chosen implementation for the comprehension and on the implementation of the bulk types. There are several methods to optimize the query language. Among these are the code optimization and the choice of more efficient bulk type implementations. Possible starting points for the optimization of the query language are described in the second part of chapter 6. Experience and further research have to show the effectiveness of the various approaches. In the third part of chapter 6 the experiences with the employed technologies are summarized. First the language TL used as host language for the query language is considered. Afterwards the experiences with the employed syntax extension tool are summarized. In the last part of the chapter some topics for further research related to the presented approach are proposed.

Chapter 2

Query Languages and Bulk Types

In this thesis an extensible framework for a query language is presented. This query language is based on comprehensions, known from functional programming. Comprehensions are used as declarative notation to express list manipulations. Comprehensions are proposed as query language in [Trinder 92]. An introduction to the origins and applications of comprehensions is given in this chapter. It is one of the goals of the proposed query language to allow querying different bulk types. In the literature different approaches for the definition of bulk types exist [Atkinson et al. 90; Breazu-Tannen, Subrahmanyam 91; Stemple, Sheard 91; Watt, Trinder 91; Beeri, Ta-Shma 94]. In the past, most work relied on a more intuitive understanding of bulk types [Atkinson et al. 90]. In recent work attempts are made to find a formal framework for the definition of them. Monads are taken as starting point in some approaches [Wadler 90; Trinder 92; Wadler 90] whereas other work propose a definition employing abstract data types for the specification of bulk types [Ross 92; Beeri, Ta-Shma 94]. There are also some approaches relying on a free algebra approach of bulk data types [Fegaras 94], or on other notions of bulk types as for example the *maps* presented in [Atkinson et al. 90]. This chapter gives a short description of some of these approaches.

An extension of the monads, the ringads, are taken as a basis to describe the semantics of comprehensions for other bulk types than lists. This approach is taken as a starting point for the definition of the semantics of the query language and especially of the mixed queries described in chapter 4. The functional implementation of the query language described in chapter 5 is also based on this approach. Further the restrictions from a monadic definition of bulk types are loosened in a more general theoretic framework for the semantic definition of bulk types. This generalization is a summary of the work of Beeri and Ta-Shma undertaken recently [Beeri, Ta-Shma 94].

2.1 Comprehensions: Origins and Applications

The most direct way to denote finite sets is to list their members, e.g., $\{ 1, 2, 3, 4 \}$. However, sets do not depend upon any ordering of their elements or upon duplication, so that $\{ 1, 2, 3 \}$ and $\{ 3, 1, 1, 2, 2, 3 \}$ denote the same set. A disadvantage of this notation is that it is not adequate for the representation of large sets and impossible for the representation of infinite sets. A more adequate and concise notation for sets are ZF-expressions named after Zermelo and Fraenkel [Zermelo 08]:

Suppose E is an expression and P is a predicate, then $\{ E \mid P \}$ denotes the set that contains x if, and only if, there are values for the variables in P and E that make $P = true$ and $E = x$.

In many functional programming languages lists are a central programming feature. They are used for data structuring as well as for programming, e.g., for defining functions. Lisp, Miranda, and Haskell [McCarthey et al. 65; Turner 85; Bird, Wadler 88] are typical representatives for this class of functional languages.

The heavy use of lists in these programming languages motivates the introduction of declarative constructs for list expressions. List comprehensions, an adaptation of the ZF-expressions, are often used for this purpose. List comprehensions in functional languages were first introduced in KRC [Turner 85], a functional language designed by Turner [Turner 82] and were further developed in Miranda.

2.1.1 Introducing the Comprehension Notation

The following considerations are based on the comprehension syntax presented in [Wadler 87, p.128]. Many other authors employ a similar notation referring to Wadler's proposal. Analogue to ZF-expressions, a list comprehension is mainly divided into two parts: an expression and a list of qualifiers.

[*<expression>* | *<qualifier>* ; . . . ; *<qualifier>*]

There are two forms of qualifiers, generators and filters.

Generators: A generator is a range expression ranging over a list. It introduces a range variable and a list as the domain of this variable. Syntactically, a generator is of the following form:

$$x \leftarrow A$$

In this expression x denotes the range variable and A the name of a list. Instead of a named list A, it is possible to have an expression denoting a list or a functional expression returning a list as a result. During the evaluation of the comprehension the members of list A are bound successively to the variable x. This mechanism is denoted by the arrow symbol pointing from the list to the variable.

Filters: A filter is a Boolean expression. All range variables introduced in generators preceding the filter can be part of the Boolean expression. The filter restricts the members of the lists, which can affect the result of the comprehension, to those which fulfil the given predicate.

The result of a list comprehension is by definition a list. The expression preceding the bar in the comprehension describes the construction of the elements of the resulting list. All introduced range variables can appear in this expression.

The following example of a list comprehension denotes a list of all members of a given list A of integers satisfying the condition that the members are all greater than five.

$$[\ x \mid x \leftarrow A;\ x > 5\ ]$$

The semantics of list comprehensions may be given by a set of five rewrite rules [Wadler 87]:

$$
\begin{array}{ll}
(1) & [\ E \mid v \leftarrow [];\ Q\ ] \rightarrow [\ ] \\
(2) & [\ E \mid v \leftarrow E' : L'\ ;\ Q\ ] \rightarrow [\ E \mid Q][E'/v] ++ [\ E \mid v \leftarrow L';\ Q\ ] \\
(3) & [\ E \mid \mathit{False};\ Q\ ] \rightarrow [\ ] \\
(4) & [\ E \mid \mathit{True};\ Q\ ] \rightarrow [\ E \mid Q\ ] \\
(5) & [\ E \mid\ ] \rightarrow [E]
\end{array}
$$

In the rules E denotes an expression, Q denotes a sequence of zero or more qualifiers and $E' : L'$ denotes a list with head E' and tail L'. The append function on lists is denoted by $++$. The used notation for substitution [<*expression*>][E'/v] expresses that all free occurrences of v in the expression are replaced by E'. The first two rules explain the impelling effect of the generators on the comprehension result. A comprehension with an empty list in the first generator results in an empty list (rule 1). In case of having a list $E' : L'$ in the generator all occurrences of the range variable v are replaced by the expression E' in the remaining qualifiers and the expression E. The result of the substitution is appended to the result of the rest of the comprehension. Rules (3) and (4) describe the effect of filters on the result. Rule (5) reduces comprehensions without qualifiers, which occur as temporary result during the reduction process. The following rule for the generators derived from rules (1) and (2) makes the effects of the generators in a comprehension more obvious:

$$(2')\quad [\ E \mid v \leftarrow [E_1, \ldots, E_n]\ ;\ Q] \rightarrow [\ E \mid Q]\ [E_1/v]\ ++\ \ldots ++\ [\ E \mid Q\ ]\ [E_n/v]$$

Employing these rules, the above example with the list $A = [2, 4, 6, 8]$ leads to the following reductions:

$$
\begin{array}{ll}
 & [\ x \mid x \leftarrow [2, 4, 6, 8];\ x > 5\ ] \\
 & \\
\rightarrow & [\ 2 \mid 2 > 5\ ] ++ [\ 4 \mid 4 > 5\ ] ++ [\ 6 \mid 6 > 5\ ] ++ [\ 8 \mid 8 > 5\ ] \\
\rightarrow & [\ 2 \mid \mathit{False}\ ] ++ [\ 4 \mid \mathit{False}\ ] ++ [\ 6 \mid \mathit{True}] ++ [\ 8 \mid \mathit{True}\ ] \\
\rightarrow & [\ \ ] ++ [\ \ ] ++ [\ 6 \mid\ ] ++ [\ 8 \mid\ ] \\
\rightarrow & [\ \ ] ++ [\ \ ] ++ [\ 6\ ] ++ [\ 8\ ] \\
\rightarrow & [\ 6, 8\ ]
\end{array}
$$

2.1.2 A Translation Scheme

On the basis of the five reduction rules describing the semantics of the comprehensions a translation scheme is evolved in [Wadler 87]. This scheme allows the translation of list comprehensions into an enriched lambda calculus. The translation uses a higher-order function *flatMap* which takes a list-valued function f and a list L as parameters. *flatMap* applies f to each element of the list and appends the resulting lists, yielding a single result list.

(1) **Translate** $([E \mid v \leftarrow L;Q]) \rightarrow$ *flatMap* λ *v* **Translate** $([E \mid Q])$ **Translate** (L)

(2) **Translate** $([E \mid B;Q]) \rightarrow$ *If* **Translate** (B) **Translate** $([E \mid Q])$ *NIL*

(3) **Translate** $([E \mid\,])\rightarrow$ *CONS* **Translate** (E) *NIL*

The given rules are expressed using a meta-function '**Translate**', to describe the different translation steps. A generator $v \leftarrow L$ is translated into a call of the function *flatMap*, where the function-parameter f is a lambda abstraction over the range variable v. The body of this lambda abstraction is formed by the translation of the rest of the comprehension $[E \mid Q]$. The consecutive application of the rules guarantees the translation of all existing generators into nested *flatMap* expressions.

The presented rules are part of a larger translation scheme for a language that incorporates comprehensions [Peyton-Jones 87]. The translation of list L is performed by further rules of the scheme. These rules are omitted here since they are not of interest for the considerations[1]. Rule 2 shows that a filter B is translated into a conditional expression. If the filter is fulfilled, the expression returns the translation of the rest of the comprehension $[E \mid Q]$, else an empty list is returned. A comprehension without qualifiers is translated into a list containing only an expression E (rule 3).

The efficiency of the expressions generated by the translation scheme may be improved by using methods from program transformation [Feather 87] known as *inlining* and *elimination*. The resulting rules are presented in [Wadler 87, p.135] are optimal in the number of *cons*-operations performed to construct the result.

2.1.3 Comprehensions as Query Language

Comprehensions have not only been used for functional programming as explained in the previous section, but have proved to be a successful notation for database applications. List comprehensions are used in persistent storage systems, in distributed database languages, and in languages for database programming [Heytens, Nikhil 91; Kato et al. 90; Trinder 92; Ghelli et al. 92]. In his thesis, Trinder introduced comprehensions as a query notation for database programming languages [Trinder 89, chapter8]. Trinder argues that the complexity having multiple kinds of bulk data can be reduced if the comprehension notation is used [Trinder 92]. It is stated that comprehensions can be defined in terms of a number of functions which are also definable over a dedicated set of collection types. In connection to this, the integration of bulk data constructors into database programming languages plays an important role.

According to Trinder the following properties make the comprehension notation a good candidate for a query language for different bulk types.

Clarity and Brevity: Comprehensions allow a brief, declarative specification of queries. It is obvious how to formulate intended queries and conversly queries formulated by comprehensions are also easy to understand.

[1]This is also true for the translation of the filter predicate B in rule 2 and for the expression E in rule 3.

Efficiency and Optimizability: Database literature identifies four algebraic and two implementation based improvement strategies [Ullman 88b]. For these strategies equivalent optimization rules can be developed for list comprehensions [Trinder, Wadler 89].

Extensibility: Comprehensions as query language are originally restricted to lists. It is shown in [Wadler 90] that they are easily made applicable to a larger class of bulk types, fulfilling certain structural restrictions.

Expressive Power: Relational completeness introduced in [Codd 72] is a measure for the expressive power of query languages. List comprehensions augmented by three auxiliary functions are proven to be relational complete in [Trinder 89]. It is shown that any relational calculus query can be mapped into an equivalent list comprehension. Since the semantic basis of comprehensions can be expressed on the basis of lambda calculus [Wadler 87], it is argued in [Trinder 92] that recursive functions added to comprehensions allow the formulation of recursive queries. As argued by Trinder, augmenting functions results in a query language that is computational complete and allows the formulation of recursive queries [Trinder 92].

For these reason comprehensions are chosen as basis for the query language developed in this work.

2.2 Bulk Types

In the database programming community there is a quite common understanding of the intuitive semantics of a bulk type. Bulk types are understood to be collections or values that have the properties:

Homogeneity: This is expressed as, "*Every collection is homogenous, i.e., all elements have a common type, termed the element type of the collection*" in [Watt, Trinder 91, p.3]. Others claim that, "*All elements are of a certain type*", have, "*certain required properties*", are of a "*subtype of some specified type*" [Atkinson et al. 90, p.6]. Moreover some argue that, "*data have uniform structure*" [Hull, Su 90].

Largeness: This is expressed as, "*the instances can be populated in a unbound fashion*", and "*instances of a bulk are potentially large*", in [Atkinson et al. 90]. Others make the statement that, "*arbitrary large numbers of elements*" are allowed [Beeri, Ta-Shma 94].

Further, a bulk type is understood to have the property of making all those elements accessible or retrievable which were inserted during the construction of a particular collection of entities [Watt, Trinder 91, p.17]. Examples of bulk types are *sets*, *lists* or *sequences*, *multisets* or *bags*, *relations*, *1NF relations*, *NF2 relations*, *directed acyclic graphs (DAGs)*, *lattices*, *trees* [Atkinson et al. 90, p.2,6,15] [Atkinson et al. 93, p.49] [Wong 92].

Atkinson et al. give a number of further requirements for bulk types [Atkinson et al. 90, p.6–7]. Some of them are on a conceptual level whereas others are more implementation-dependent. Since the conceptual understanding is emphasized here, the implementation-dependent requirements are omitted in the following list:

1. Data type completeness is expected for bulk types, i.e., arbitrary types are possible for the elements of a bulk;
2. Instances of a bulk may vary in size;
3. Means for iteration should be provided over the elements of a bulk type;
4. Bulk data values should be able to persist in a system;
5. A useful and succinct algebra with well-understood properties is required over the instances of the bulk type.

2.2.1 Monads and Monadic Bulk Types

The considerations presented in section 2.2 allow an intuitive understanding of the notion of a bulk type. A formal framework is needed to allow accurate, general, and formal propositions about bulk types. The monads are an early attempt to describe the formal properties of bulk data types [Moggi 89; MacLane 71].

Monads

The notion of a monad is taken from category theory and describes a specific structure employing mathematical methods. This section gives an overview of the monads. A detailed introduction can be found in [Lambek, Scott 86].

A category consists of two kinds of entities, namely, *objects* and *morphisms*. Among the morphisms there is an *identity* mapping for each object of the category and a facility for *composing* morphisms. For arbitrary categories monads may be defined. The monad used for the definition of bulk types is based on the following category: The objects of the category are types and the morphisms are functions. The identity functions id_α for each type α form the identities and function composition, denoted $\circ$, performs the composition of morphisms.

As in all other categories the following laws for the composition and the identities hold:

$$(h \circ g) \circ f \quad = \quad h \circ (g \circ f) \tag{2.1}$$

$$f \circ id_\alpha \quad = f = \quad id_\beta \circ f \qquad \text{for } f : \alpha \to \beta \tag{2.2}$$

A monad consists of a functor M and two special morphisms η and μ, called natural transformations. A functor is applicable to the objects and to the morphisms of the category, i.e., on types and on functions in the special category.

As an example let M be the functor for lists. The application of M to a type α yields $M\alpha$, the type of all lists with element type α. In case of passing a function $f : \alpha \to \beta$ to M a function $Mf : M\alpha \to M\beta$ is obtained[2], taking a list of elements of α and returning a list of elements of β. The returned list is constructed by mapping the function f to all elements of its argument. A functor M has certain properties. For the chosen category it maps the identity function of a type α to the identity functions of $M\alpha$.

$$Mid_\alpha \quad = \quad id_{M\alpha} \tag{2.3}$$

[2] Sometimes referred to as function *map*.

Further, M applied to the composition of two functions is equal to the composition of their images under M:

$$M(g \circ f) \quad = \quad Mg \circ Mf \tag{2.4}$$

The functor and the natural transformations have to satisfy the laws presented in the following diagrams:

$$\begin{tikzcd}
M^3 \arrow[r, "M\mu"] \arrow[d, "\mu M"'] & M^2 \arrow[d, "\mu"] & M \arrow[r, "M\eta"] \arrow[d, "\eta M"'] \arrow[dr, "id_M"] & M^2 \arrow[d, "\mu"] \\
M^2 \arrow[r, "\mu"] & M & M^2 \arrow[r, "\mu"] & M
\end{tikzcd}$$

The laws in the left diagram are the associative laws of a monad, whereas the right diagram depicts the unity laws of a monad, with id_M as the unit of M [Lambek, Scott 86, p.28]. Such diagrams are often used to represent relationships between families of functions. The edges are names of types and the arrows depict functions. The left diagram is said to commute, since the following equation holds:

$$\mu \circ M\mu = \mu \circ \mu M, \tag{2.5}$$

i.e., the function obtained by composing the right-hand side and the top of the square is identical to the function obtained by composing the bottom and left-hand side of the square. The right diagram depicts the following unity laws:

$$\mu \circ M\eta = id_M \tag{2.6}$$
$$id_M = \mu \circ \eta M \tag{2.7}$$

A well known alternative representation of monads are the Kleisli monads. Similar to the monads considered so far, the Kleisli monad consists of a functor M and two natural transformations. One of these transformations is again η, whereas μ is replaced by the following more general morphism:

$$\mathit{flatMap} : (\alpha \ \rightarrow \ M\ \beta) \rightarrow (M\ \alpha \ \rightarrow \ M\ \beta)$$

Three laws are sufficient to describe the semantics of a Kleisly monad:

$$flatMap\ \eta \quad = \quad id \tag{2.8}$$
$$flatMap\ f \circ \eta \quad = \quad f \tag{2.9}$$
$$(flatMap\ g) \circ (flatMap\ f) \quad = \quad flatMap((flatMap\ g) \circ f) \tag{2.10}$$

The Monad of Lists

As an example of a monad, the data type of lists is taken. It is shown that the given laws are satisfied within a theory of lists.

As mentioned above it is assumed that functions and types are the basic objects of interest. The function composition is associative and the identity functions of our types are left and right identities for composition with any function. Further there is a type constructor M which takes a type to the type of lists of elements of this type and a function f to the function map mapping f over the elements of a list and returning a list. The two above laws 2.3 and 2.4 hold since $map(id_x) = id_{Mx}$ and $map(f \circ g) = map\, f \circ map\, g$ are obviously true. As an example take the list $M\alpha = [1, 2, 3, 4]$ and α to be the integers then the following result is obtained:

$$
\begin{aligned}
M id_\alpha &= map(id_\alpha : Int \to Int \quad [1,2,3,4]) \\
&= [id(1), id(2), id(3), id(4)] \\
&= [1,2,3,4] \\
&= id_{[1,2,3,4]} = id_{M\ \alpha}
\end{aligned}
$$

Having the singleton function for lists, a function $f : \alpha \to \beta$, and the type operator M for lists, one can construct the list of alphas: $M\ \alpha$, the list of betas: $M\ \beta$, and the function mapping a list of alphas to a list of betas: $map\, f$. A well-known law can then be deduced which states that constructing a singleton list of alphas and mapping the function f to it is the same as applying the function f directly to the element and constructing the singleton of the result. Morphisms having this property are said to be natural transformations. The equation is:

$$map\, f \ \circ \ singleton_\alpha = singleton_\beta \ \circ \ f \tag{2.11}$$

A similar law can be constructed when looking at the flatten function for lists:

$$map\, f \ \circ \ flatten_\alpha = flatten_\beta \circ map(map\, f) \tag{2.12}$$

To show how the associative (2.1) and unity laws (2.2) hold for lists, consider the following examples:

$$
\begin{tikzcd}
M^3\alpha = [[[23][34]][[45][56]]] \arrow[r, "map\, flatten"] \arrow[d, "flatten"] & {[[2334][4556]]} \arrow[d, "flatten"] \\
{[[23][34][45][56]]} \arrow[r, "flatten"] & {[23344556]}
\end{tikzcd}
$$

The diagram shows how the associative law (2.1) is fulfilled for a given list. A list of list of lists is given and has the type M^3. Applying the flatten operator μ to a collection M which is flattening from the outside, a list of lists is obtained whose type is M^2. Applying the operator M to the function flatten ($M\mu$) performs mapping the flatten function to the elements of a given list — i.e., flattening from the inside of the list —. The result is again of type M^2. Applying the flatten function μ to both intermediate results then gives the final result of type M.

$$\begin{tikzcd}
M\alpha = [1234] \arrow[r, "map\,singleton"] \arrow[d, "singleton"'] \arrow[dr, "id_M"] & {[[1][2][3][4]]} \arrow[d, "flatten"] \\
{[[1234]]} \arrow[r, "flatten"] & {[1234]}
\end{tikzcd}$$

In case of the unity laws (2.2) the above example shows how a given list of type M is applied to the singleton function ηM and the result of type M^2 is then flattened in a further step leading to the initial list of type M. These two operations are identical to applying the identity function id_M to the list. The second law states that mapping the singleton function to a collection, and in a further step flattening the result is also identical to applying the identity function to the original list.

In this section a small theory over lists in a mathematic notation using functions and types was introduced. The results obtained from this approach are two: First, a set of laws is identified; i.e., identities for known operators of the data type list. Once recongnizing such identities they may be exploited for transformation purposes of a defined language using these operators on the manipulation of lists. Secondly, a set of abstract operators and operations were identified. These operations can also be defined for other data types, as 'bags' and 'sets'. In terms of the above, bulk types can be defined in terms of a monad.

Not all known bulks fit the description by monad operations and laws. Therefore, we talk of monadic bulks when we mean bulk data types in this framework. At least "sets", "bags", and "list" can be defined in terms of a monad.

2.2.2 Monadic Bulk Types and Comprehensions

Various approaches are described in the literature [Trinder 92; Watt, Trinder 91; Wadler 90; Wadler 92; Atkinson et al. 93] proposing monads as a basis for a formal framework for bulk types. [Wadler 90; Trinder 92; Watt, Trinder 91] take the (list)comprehensions as a starting point. The extension of comprehensions to other bulk types are considered. The requirements posed on a bulk type by the definition of comprehensions for these bulk types are examined.

Wadler's approach [Wadler 90] is based on the first representation for monads presented in the previous section. In this approach the functor M is modelled by a type operator M and a function *map*. The two natural transformations η and μ are called *unit* and *join*, respectively. The five laws for the monad presented in the previous section are extended by two further laws. These two laws are derived from the types of the functions *unit* and *join* according to results developed in Reynolds type theory [Reynolds 83; Wadler 89]:

$$mapf \circ unit = unit \circ f \tag{2.13}$$
$$mapf \circ join = join \circ map(mapf) \tag{2.14}$$

The monad operations *map*, *unit*, and *join* are used to define comprehensions. It is shown that an additional function $zero : \sigma \rightarrow \beta\ collection$ is needed to define the semantics of filters in comprehensions. This function takes an element and returns an empty collection. The element passed as parameter is discarded. Three laws are given for the function *zero*:

$$map\, f \circ zero = zero \tag{2.15}$$
$$join \circ zero = zero \tag{2.16}$$
$$join \circ map\ zero = zero \tag{2.17}$$

The following rules are presented for the definition of comprehensions in [Wadler 90]:

$$\begin{aligned}
[t \mid \lambda] &= unit\ t \\
[t \mid x \leftarrow u] &= map(\lambda x \rightarrow t)u \\
[t \mid p; q] &= join[[t \mid q] \mid p] \\
[t \mid b] &= \text{if } b \text{ then } unit\ t \text{ else } zero\ t \\
[t \mid b; c] &= [t \mid (b \wedge c)] \\
[t \mid q; b] &= [t \mid b; q]
\end{aligned}$$

where t and u are terms, b and c are filters, p and q are qualifiers, λ denotes the empty qualifier and x is a variable. The first two rules treat the cases of a comprehension without qualifiers and a comprehension with a single generator as qualifier. The third rule describes the semantics of composed qualifiers denoted by $p;q$. The semantics of a filter is described by the fourth rule. If the filter is fulfilled, a singleton collection with t as element is returned, else the empty collection is returned. Rule five describes the composition of filters (b and c are filters). For the last rule it has to be assumed that q does not bind variables used in b. Only in this case the interchange is legal. As described in section 2.1.3 comprehensions are proposed as query language in [Trinder 92]. The extension of comprehensions to different bulk types leads to the notion of a *ringad*. Ringads are monads augmented with a function *zero* and a function *combine*. Trinder uses Kleisli monads as basis using the names *single* and *iter* instead of η and *flatMap*, respectively. As in the approach of Wadler, a function *zero* is introduced to allow the implementation of filters. The laws specified for the *zero* are somewhat different since a different monad representation is used as basis:

$$zero \circ e = zero \tag{2.18}$$
$$iter\ zero = zero \tag{2.19}$$
$$iter\ f \circ zero = zero \tag{2.20}$$

where $f : \alpha \rightarrow \beta M$.

A monad with a *zero* is called a *quad*. In order to allow the construction of bulk values in a uniform manner a further function, $combine : M\alpha \times M\alpha \rightarrow M\alpha$, is introduced. The function *combine* merges two collection c and c' so that each element in c and each element in c' have a counterpart in the result. A monad with a *zero* and a *combine* fulfilling the following laws is termed a *ringad*:

$$combine\ (zero)\ f\ x \quad = \quad f\ x \tag{2.21}$$
$$combine(f\ x)\ (zero\ x) \quad = \quad f\ x \tag{2.22}$$

A third law describes how *iter* distributes through the *combine* function:

$$iter\ f\ (combine)\ c\ c') \quad = \quad combine(iter\ f\ c)(iter\ f\ c') \tag{2.23}$$

Wadler [Wadler 92] discusses the impact of monads to modelling impure programming concepts such as exceptions and states within functional programming languages. Additionally the definition of comprehensions using monads is described. The considerations made are based in the Kleisli monad with natural transformations *unitM* for η and *bindM* for *flatMap*.

The following table gives an overview over the different notations used in the literature concerned with the notion of a monad:

Lambek, Scott86	Wadler87	Wadler90	Trinder92	Wadler92
M		M	Collection	M
		map		
η		$unit$	$single$	unitM
μ		$join$		
	$APPEND$		$combine$	$plusM$
	$flatMap$		$iter$	$bindM$
	NIL	$zero$	$zero$	$zeroM$

The table shows vertically grouped axiom systems with their operators and gives the authors for each of the identified group in the first row of the table. The authors Lambek and Scott discuss monads in a pure categorial/mathematical sense, whereas all the other authors view monads as structures to implement functional programming.

To summarize, monads with a *zero*, called *quad* in [Trinder 92], are identified as sufficient for the definition of comprehensions. The function *zero* is required to define the semantics of a filter. Quads are extended to ringads by a *combine* function to support the construction of bulk values. Ringads can be used as a basis for a formal definition of a bulk type.

2.2.3 An Initial Algebra Approach

Not all bulk types which are constructible are captured by the notion of a monad, e.g., trees with data at internal nodes [Ross 92, p.5] [Trinder 92, p.50] [Trinder 92, p.59] [Watt, Trinder 91, p.16–17] relations with keys [Watt, Trinder 91, p.16]. The aim of the work of Beeri and Ross [Ross 92; Beeri, Ta-Shma 94] is to provide a basis to allow a formal definition of the notion of a bulk type. Criteria are given to investigate whether a specified data type is a bulk data type or not. Relationships between bulk types and the role of the data type set are investigated. Algebraic properties are thus consequently utilized to investigate the effect of operations defined over bulk types. The work of defining bulk types as ringads is generalized by capturing types which are difficult to view as monads or ringads. The following sections give an overview of this framework. A detailed presentation can be found in [Ross 92] and [Beeri, Ta-Shma 94].

Formalizing Bulk Type Properties

In section 2.2 intuitively defined properties for bulk types, namely, homogeneity and largeness are introduced. As argued in [Ross 92] bulk types are described as container types which admit finite type definitions, while their instances can be populated in a unbound fashion. Integers and arrays are, for example, not considered as bulk types since they can not be regarded as container types and can not be populated in a unbound fashion, respectively.

Ross makes an approach to describe bulk types in terms of abstract data type specifications. The concept of a bulk data type is given through parameterized algebraic specifications [Ross 92]. These abstract data type specifications consist of a formal parameter specification and a target specification. A sort represents the formal parameter specification and describes a set of minimal requirements which all actual parameters of the target specification must satisfy. Hence the notion of homogeneity is naturally captured in this framework. In a standard construction [Gougen et al. 76] the initial algebra is associated with each of the specified abstract data types. The notions of largeness and conservativity are employed to single out only those abstract data types that meet the conditions which have to be fulfilled by bulk data types. In the following these conditions are explained in detail:

Largeness: To capture the intuition that bulk data types are types with arbitrary large numbers of elements, a formal criterion is given. A bulk is considered large if, and only if, the directed hypergraph derived from its definition, especially from its constructors, is large, i.e., has cyclic structures and the formal parameter specification for the bulk admits infinite algebras [Ross 92, Def. 3.10, Th. 3.11]. Admitting infinite algebras means that no constraint limiting the actual size of constructible bulks is imposed on the target specification by the formal parameter specification. An example of such a constraint would be the equation '$insert(x, B) = false \quad \text{if } size(B) > 1000$', restricting the number of elements which can be inserted into a collection B.

The construction algorithm for the directed hypergraph on the constructors of an abstract data type specification is given in [Ross 92, Def. 3.9].

Conservativity: The formal notion of conservativity expresses the property of a bulk type to preserve all data items added to it. This criterion focuses on the equations found in the specification of the target type. Only the constructors are of relevance to this criterion, the destructors are excluded from these considerations, since their nature is by definition not information preserving and, therefore, the conservativity property does not hold for them.

A bulk is considered to have the conservativity property, if for all suitable parameters the equivalence, constructed from the initial algebra approach, of any pair of data terms (t_1, t_2) implies that the contents of the bulks from data terms t_1 and t_2 are equal [Ross 92, Def. 3.17].

In general, conservativity is not recursive [Ross 92, Th. 3.19], but constraints that have to be satisfied by the equations of the specification can be named [Ross 92, p.19]. These constraints are sufficient to guarantee conservativity of the specified bulk type.

In summary, abstract data types are bulk types, when proven to be large and conservative, where *largeness* is a property of the constructors and *conservativity* is a property satisfied by the equations of an abstract data type.

Bulk-Morphisms

Of further interest are the relationships between bulk types. Ross investigates them by constructing a category whose objects are bulk type specifications and whose relationships are morphisms [Ross 92]. The notion of a bulk-morphism is used to describe the relationship between two bulks. For example, the bulk-morphism from lists to sets represents a type conversion of a list collection into a set collection. A bulk-morphism has to specify how to map a given bulk type specification (abstract data type) to Another bulk type specification; i.e., the sorts, function symbols, and data terms of one data type have to be mapped according to constructs of the target specification. Again certain properties have to hold for morphisms to identify them as bulk-morphisms. Ross defines a formal definition of a morphism for a given parameterization [Ross 92, p.21]. Three mappings h_s, h_F, and h^A_{DT} are defined to map the sorts, the function symbols, and the data terms of one bulk data type into the other, respectively. The mappings have to prove holding the *b-morphism criterion*:

$$f(h^A_{DT}(t_1), \cdots, h^A_{DT}(t_n)) = h^A_{DT}(h_F(f)(t_1, \cdots, t_n));$$

for all function symbols f of the source type of the defined mapping with signature $s_1, \cdots, s_n \to s$ and for all data terms $t_1 \in s_1, \cdots, t_n \in s_n$.

Having parameterized specifications as bulk types, a family of morphisms is definable from one bulk type to the other. One morphism for each of the actual parameter types is constructed and the resulting family denotes the bulk-morphism between the considered types. These morphisms have to satisfy certain properties to constitute a bulk-morphism.

First, it has to be guaranteed for the obtained family of morphisms that they act in a uniform way over all suitable actual parameters, giving similar results for similar inputs [Ross 92, p.25]. *Well-definedness* and *complete-definedness* [Ross 92, p.25] of the morphism guarantee this uniform behavior. Second, the family of morphisms from one bulk into the other has to be *content preserving*, i.e., the content of the image of the morphism applied to a bulk has to be the same as the contents of the original bulk. This is always true for all possible bulks subsumed in one bulk data specification [Ross 92, p.27]. Thirdly, the family of morphisms has to satisfy the *conservativity condition*. This states that the equality of two data terms under the morphism implies the equality of the contents of their images unser the morphism [Ross 92, p.27].

A Bulk-Morphism

This section gives an example of a bulk-morphism taken from [Ross 92]. A bulk-morphism from lists to binary trees is constructed and all the properties named in the previous section are discussed. First, the target specifications for the bulk types 'list' and 'binary tree' are given:

<u>**LIST**</u> =
import : data, nat
sorts : list
cons : nil : → list
insert : data, list → list
func : length : list → nat
eqns : x ∈ data, L ∈ list
length(nil) = 0
length(insert(x,L)) =
1 + length(L)

<u>**BINTREE**</u> =
import : data, nat
sorts : bintree
cons : empty : → bintree
leaf : data → bintree
node : bintree, bintree → bintree
func : breadth : bintree → nat
height : bintree → nat
eqns: x,y ∈ bintree, d ∈ data
breadth(empty) = 0
breadth(leaf(d)) = 1
breadth(node(x,y)) =
breadth(x) + breadth(y)
height(empty) = 0
height(leaf(d)) = 0
height(node(x,y)) =
1 + max(height(x),height(y))

where *nat* are the natural numbers, zero included. The morphism from $LIST(char)$ to $BINTREE(char)$ defines a mapping for the sorts (h_s), for the function symbols (h_F), and for the data terms (h_{DT}):

h_S(BINTREE) = LIST	h_F(breadth) = length	h_{DT}^{char}(c) = c
h_S(char) = char	h_F(height) = length	h_{DT}^{char}(n) = n
h_S(nat) = nat		h_{DT}^{char}(nil) = empty
		h_{DT}^{char}(insert(c,L)) = node(leaf(c),h_{DT}^{char}(L))

The mapping of the sorts is obvious. h_F maps the function symbols *breadth* and *height* of $BINTREE$ to the function symbol *length* of $LIST$. The morphism maps lists to "right-handed" trees (see the definition of h_{DT}). For this kind of tree the height and the breadth is always equal.

In order to show that the defined morphism is a bulk-morphism the following properties have to be proven.

1. The *b-morphism criterion*;
2. The reasonability of recursion;
3. The well-definedness and complete-definedness of h_{DT};
4. The morphism h_{DT} is content preserving;

5. The conservativity of the mapping.

To prove the b-morphism criterion, one has to show that for all data terms of the list type the following equations must hold:

(a) breadth($h(m)$) = h_{DT}(length(m)) = length(m)
(b) height($h(m)$) = h_{DT}length(m) = length(m)
This is proven by structural induction over the data type $LIST$. The above morphism can be viewed as defining a family of morphisms. It has to be shown that the family obtained forms a natural transformation in the categorial sense.

Let d $\in$ data, L $\in$ list. We define
h(nil) = empty
h(insert(d,L)) = node(leaf(d),h(L))

The proofs of the remaining properties for the given example can are given in [Ross 92].

Set is Terminal

An object T of a category is called *terminal* if for every other object of the category there exists a unique mapping to this object T [Lambek, Scott 86, p.19]. The bulk type theory presented by Ross forms a category with the bulk type specifications as object and the bulk-morphisms as morphisms. It is shown in [Ross 92] that the bulk type set is a terminal object in this category, i.e., for all bulk data types, there is a unique bulk-morphism from B to the bulk type set. Further, it is known from results of category theory that all terminal objects of a category are isomorphic. Sets are, therefore, upto isomorphism, the only object in the category of bulk types with this property.

The bulk-morphisms and especially the terminal property of sets are taken as a starting point for the discussion of the semantics of the query language developed in this work. Query languages with different bulk types as inputs and different bulk types as results can be interpreted as bulk-morphisms in a given category. The discussions related to this are given in chapter 4.

Chapter 3

Language Technology

The syntax of the developed query language and its implementation are not built-in. The query language is realized by employing syntax extension technology. A syntax extension tool is driven by rules that associate the chosen syntax of the query language with an implementation of the constructs in a target language. In order to allow the implementation of a query language a concrete target language has to be chosen. In this approach, the target language TL is employed [Matthes 92b], a modern programming language supporting enhanced features as parametric polymorphism and higher-order functions. TL is also used as host language to embed the query language: constructs of the host language can be used inside the query, and queries are valid parts of programs written in the host language. The features of the language TL relevant for the chosen approach are described in the first part of this chapter.

A syntax extension tool, developed at the University of Hamburg [Schröder 93], is employed to realize the query language. The second part of the chapter gives a summarized overview of this syntax extension tool. The concrete rules needed for the implementation of the query language are described in chapter 5.

3.1 Introduction into the Tycoon Language

The programming language TL evolved from the languages *Quest* [Cardelli 89; Cardelli 90] and *P-Quest* [Matthes 91; Müller 91; Niederée et al. 92]. Abstracting from some syntactical differences, *Quest* is mainly a subset of TL.

Considering the syntactical structure and the module concept, the language TL is similar to the languages of the Modula family (Modula-2 [ModISO 91] Oberon [Wirth 87], Modula-2+ [Rovner et al. 85], Modula-3 [Nelson 91] and Ada [Ichbiah 83]). On the other hand, its semantic is closer related to polymorphic functional languages of the ML language family [Cardelli 89; Cardelli 90; Mauny 91; Field, Harrison 88; Hudak 89]. The basic semantic concept is formed by the model of the language *Fsub* [Cardelli et al. 91], a commonly accepted formal basis for the study of newer type systems. In TL this explicitly and strictly typed second-order lambda calculus with subtyping is embedded into a complete, modular programming language. Supporting functional and imperative concepts, TL allows functional as well as imperative programming. It supports an orthogonal persistency concept. TL programs may be interpreted as well as compiled.

The evaluation of expressions is strict and deterministic. TL supports many concepts of modern programming languages:

- functions and types as first class objects
- subtyping for all type constructors and for type operators
- parametric polymorphism and subtype polymorphism
- abstract data types
- modules, interfaces, and libraries
- exception handling
- dynamic typing

This part of the chapter describes the features of TL that are relevant to this work. It is structured as follows: the first section presents two basic concepts of TL, naming and binding. Functions are an important concept in TL. They are first-class values of the language. The definition and use of simple functions, recursive functions, and higher-order functions is considered in the second section.

Section 3.1.3 is dedicated to value and type constructors. For all type constructors a subtype relationship is defined. These subtype relationships are described in section 3.1.4. Parametric polymorphism is an attracting feature in TL. It allows the definition of polymorphic functions as well as the definition of type operators. Parametric polymorphism and its applications are discussed in section 3.1.5. TL is not restricted to functional programming. It also supports imperative programming features as mutable variables, loops, and a powerful exception mechanisms. These features are the topic of section 3.1.6. Finally an overview of the structuring facilities for TL programs is given in section 3.1.7.

The following syntactic rules and conventions are important for the examples presented in this chapter and in chapter 5. Keywords are emphasized by writing them in bold face. Reserved identifiers referring to types or used to define types are capitalized. All other identifiers start with small letters. This rule is adopted as a convention for user-defined identifiers whenever TL code is presented in this work. This convention improves the readability of the programs. Comments are enclosed in (* and *). They may be arbitrarily nested.

3.1.1 Naming and Binding

Important concepts in TL, on the level of values as well as on the level of types, are *naming* and *binding*. User-defined names may be bound to semantic objects of the language and afterwards used to denote the bound objects.

Binding of Values

The binding of names to values is accomplished by constructs of the following form:

let a = *3*

The variable a is not mutable[1]. Every subsequent use of such a variable evaluates to the bound value. If the right-hand side of a binding is an expression, this expression is evaluated in advance, and the result of the evaluation is bound to the specified identifier.

let x = a + *4*

binds the identifier x to the value 7.

Groups of bindings can be written in one program block without the use of limiting characters as comma or semicolon. A sequence of bindings denotes a sequential binding and they are evaluated in the order of their appearance. Bindings may refer to identifiers bound earlier in the sequence:

let a = *3*   **let** x = a + *3*

Simultaneous bindings are also possible in TL. In this case the bindings are connected by the keyword **and**. The identifier *b* is bound to the value 6 by the following bindings:

let a = *4*
let a = *100/4* **and** *b* = a + *2*

Recursive bindings must be characterized as such by the use of the keyword **rec**. The expressions permitted in recursive bindings have to obey certain constraints concerning their construction (for details see [Matthes 92b]). Mutual recursive bindings are expressed by the combination of the concepts *recursive* and *simultaneous binding*. An important application of recursive bindings is the definition of recursive functions (see section 3.1.2).

The types of variables may be inferred by the TL compiler from the inferred types of the subexpressions and the types of the values used to construct the expression bound to the variable. The types can be given explicitly by the programmer to improve the readability of the program. In recursive bindings, the type of the variable has to be given explicitly.

Binding of Types

Most of the concepts presented for the value level can also be found on the type level. Type bindings are established as follows:

Let *A* = *Int*

This construct binds the identifier *A* to the base type *Int*. Recursive bindings on type level are marked by the keyword **Rec** in analogy to the keyword **rec** on the value level. An example for a recursive type definition is presented in section 3.1.3. Simultaneous bindings are achieved by connecting two or more bindings by the keyword **and**, as it is the case for simultaneous value bindings.

[1] The definition of mutable variables will be presented in section 3.1.6.

3.1.2 Functions

Since TL supports functional programming, functions are a central concept in TL. Functions are first-class values. This permits the definition of higher-order functions and the inclusion of functions as components into constructed value, e.g., tuples. TL also supports polymorphic functions. Since they represent an application of parametric polymorphism, their presentation is postponed to the section describing parametric polymorphism (section 3.1.5).

Function Definition and Binding

The definition of a function consists of defining a function abstraction and binding a name to it. In contrast to many programming languages, as Pascal [Wirth 71] for example, where definition and naming are inseparably coupled, TL allows the definition of a function abstraction without binding a name to it. These unnamed functions can be used to be passed as parameters to higher-order functions. The definition of a function abstraction is introduced by the keyword **fun** followed by a list of signatures which describe the formal parameters and an expression which forms the function body. The parameters of the function are separated by a blank.

fun(*x :Int*) *x + 1*
fun(*x :Real y :Real*) *:Real x ++ y*

The function body can refer to the formal parameters, to global variables in the static scope of the function definition, and to local variables introduced in the function body.

Functions are first-class values in TL, so names can be bound to functions by the standard binding mechanism for values:

let *succ =* **fun**(*x :Int*) *:Int x + 1*
let *add =* **fun**(*x :Real y :Real*) *:Real x ++ y*
let *succ2 = succ*

The first function takes a parameter of type *Int* and returns a value of type *Int*. It calculates the successor of the integer value passed as parameter. The second function *add* adds two given real values[2]. It takes two parameters of type *Real* and returns a value of type *Real*. For convenience, TL also offers the following short form for the definition of a named function:

let *succ(x :Int) = x + 1*
let *add(x, y :Real) = x ++ y*

When the function is applied, the actual parameters are dynamically bound to the formal parameters. They may be explicitly bound, using the *let*-construct, but it is also sufficient just to list the actual parameters. Since the parameter list is an ordered sequence, the assignment to the formal parameters is unique. The following applications of the function *add* yield the same result:

[2] Note that there is no operator overloading in TL; therefore there are different operators for the addition of integer and for the addition of real values.

```
add(1.0  2.0)
add( let x = 1.0  let y = x ++ 1.0)
```

A function call *f(D)* is evaluated as follows: First, *f* is determined. Then, sequential evaluation of the bindings *D* yields the actual parameters. The sequential evaluation allows parameter definitions to reference to earlier bound parameters, as it is seen in the second example of applying *add*. The result of the function is yielded by evaluating the body of the function, where the formal parameters are bound to the actual parameters.

TL distinguishes two kinds of identifiers: Alphanumeric identifiers and infix symbols. Alphanumeric identifiers are formed by a sequence of characters and digits, infix symbols contain only special symbols. The use of infix symbols as names of functions allows the definition of binary infix operators. For example, the function *string.concat* defining string concatenation may be renamed employing an infix symbol:

```
let <> = string.concat
```

Functions bound to symbolic identifiers may be applied using the infix notation as well as the standard notation for function application:

```
<>("concat" "enation")
"concat" <> "enation"
```

So, the renaming of the function *string.concat* results in a more convenient notation for applying it. This technique is employed to yield infix operators for the predefined operations on the basic types which are normal functions imported from modules in TL .

Recursive Functions

TL allows the definition of recursive functions. This is accomplished by the use of recursive bindings. The example shows the well-known function computing the factorial of a number:

```
let rec fac(n :Int) :Int =
  if n == 0 then 1
  else n * fac(n - 1)
  end
```

To avoid operator overloading the equality test in TL, is not performed by the symbol =, which is reserved to bind names, but a special operator == is used instead.

By combining the concepts of recursive and simultaneous binding, it is possible to define mutual recursive functions.

Function Types

A function type describes the names and types, i.e., the signatures of the formal parameters and the type of the result of the function. Function types are introduced by the keyword **Fun**. Here are the types of the functions which are used as examples in the preceding sections:

succ, succ2 :**Fun**(*x* :*Int*) :*Int*
fac :**Fun**(*n* :*Int*) :*Int*
add :**Fun**(*x* :*Real* *y* :*Real*) :*Real*

Function types are needed to define higher-order functions and types with functions as components unless the short forms are used (see below). They may be used by programmers wherever functions are defined to improve the readability. Names of formal parameters can be omitted in function types. This yields a more general function type: functions with arbitrary names for this formal parameter are included in this type. For example the functions *succ*, *succ2* and *fac* are all of the type:

succ, succ2, fac :**Fun**(:*Int*) :*Int*

Higher-Order Functions

Functions are first-class values in TL. It, therefore, is possible to define higher-order functions, i.e., functions which take functions as parameters and/or return functions as results:

let *twice* = **fun**(*f* :**Fun**(:*Int*) :*Int* *a* :*Int*) :*Int* *f*(*f*(*a*))
let *newInc* = **fun**(*x* :*Int*) :**Fun** (:*Int*) :*Int* **fun**(*y* :*Int*) :*Int* *x* + *y*

Function valued parameters and results have to be specified by function types in the signature. The following short forms for the definition of higher-order functions omit the explicit use of function types:

let *twice*(*f*(:*Int*) :*Int* *a* :*Int*) = *f*(*f*(*a*))
let *newInc*(*x* :*Int*) (*y* :*Int*) = *x* + *y*

twice is a higher-order function that accepts a function and a value as parameter. It applies the function passed as first parameter twice to the second parameter:

twice(*succ* 3)
 ⇒ 5 :*Int*

newInc is a higher-order function that takes an integer value as parameter and returns an integer function as result. If *newInc* is applied to a parameter *b*, it returns a new unnamed function of type **Fun**(:*Int*) :*Int*. If this new function is applied to an integer value *c*, it returns the sum of the values *b* and *c*:

let *plus2* = *newInc*(2)
plus2(5)
⇒ 7 :*Int*
let *plus3* = *newInc*(3)
plus3(5)
⇒ 8 :*Int*
newInc(3)(5)
⇒ 8 :*Int*

The function *newInc* may be viewed as a function generator generating different functions in dependence of the first passed parameter, e.g., *plus2*, *plus3*. The short form of *newInc* proposes an alternative view. *newInc* may be interpreted as function with two separated parameters. The application of *newInc* to a single parameter may then be viewed as partial application of the function (currying of functions).

3.1.3 Value and Type Construction

TL supports a set of value constructors that may be arbitrarily combined with each other. For each constructor c on the value level there exists a corresponding constructor C on the type level, that is used to construct the type of a value constructed with c. Examples are the definition of functions and function types as described in section 3.1.2. The operator == is provided for equality tests for values of base types as well as for constructed values. For simple values it checks the equality, for constructed values an identity test is performed: *v1 == v2* only holds if *v1* and *v2* are synonyms for the same value.

Tuples

TL offers two constructors to define aggregate types: *tuples* and *records*. Tuples represent the labeled cartesian product type. The components of a tuple are ordered. A tuple type defines an ordered sequence of signatures:

Let *Person* = **Tuple** *name :String age :Int* **end**

Values of tuple types are ordered lists of bindings. The bindings may be anonymous.

let *peter* = **tuple let** *name = "Peter"* **let** *age = 25* **end**
let *peter :Person* = **tuple** *"Peter" 25* **end**

Since functions are first-class values in TL they may be built in as components of aggregate types, for example for tuples. Selection of components of a tuple is denoted by using the dot notation, e.g., *peter.age*.

Records

The second alternative to define aggregate types in TL are records. In contrast to tuples the components of records are not ordered. Record types are sets of signatures.

Let *Point* = **Record** *x :Int y :Int* **end**

Accordingly, values of a record type are sets of bindings. As in the case of tuple values component selection is performed by the dot notation.

Tuples allow a more efficient implementation of component selection since the components are ordered. The advantage of records compared to tuples is a higher flexibility in defining subtype relationships. In addition, record values may be extended by additional components without loosing the record identity. These two topics are discussed in section 3.1.4.

Variants

The variant type of TL is closely related to the tuple type. The syntax of variant types is presented in an example together with the recursive types in the next part of the section.

Recursive Types

TL supports the definition of recursive types. A recursive type is introduced by the keyword **Rec**. An example is the definition of the type of lists of integers:

```
Let Rec IntegerList <:Ok =
  Tuple
      case nil
      case cons with head :Int  tail :IntegerList end
  end
```

IntegerList is defined by a variant type. The different variants of such a type are introduced by the keyword **case**. There are two variants for a list. A list can be empty (**case** *nil*) or it consists of a head of type *Int* and a tail of type *IntegerList* (**case** *cons*).

The different variants of a variant type can be inspected by a case-expression as illustrated by the following example:

```
let rec sum (l: IntegerList) :Int =
  case l
    when nil then 0
    when cons with x then x.head + sum(x.tail)
  end
```

The function *sum* sums up all integers contained in the list. The *with*-construct allows the introduction of a identifier that takes the value of the considered variant. The introduced identifier *x* is of type:

Tuple *head :Integer tail :IntegerList* **end**

3.1.4 Subtyping

A subtype relationship (<:) is defined on the types of TL. The intuition about the subtype relationship is set inclusion between supertype and subtype, i.e., each value of a type T is also a value of each supertype *T1* (T <:T1) of this type T. There are no subtype relationships between the basic types, except the trivial ones (e.g., Int <:Int). For structured types the subtype relationship is defined inductively. The type **Ok** plays a special role in the subtyping hierarchy. **Ok** is the top type, i.e., it is the supertype of all non-parameterized types[3].

[3] There is also a subtype hierarchy defined on type operators. This aspect is not discussed here.

The subtype relationship specifies a partial ordering on the types. It is transitive (A <:B ∧ B <:C => A <:C), reflexive (A <:A), and the subsumption principle (a :A ∧ A <:B => a :B) holds for it.

The notion of subsignatures are needed to define subtype relationships on function types: Signatures S' are called *subsignatures* of signatures S, if (1.) the ordered sequences S and S' have the same length, (2.) the types A_i of the signature components of S are subtypes of the types $A_{i'}$ in the same position of S', and (3.) if both variable names x_i and $x_{i'}$ are not anonymous then $x_1 = x_{i'}$ has to hold. The subsignature relationship is denoted by S' <::S.

Subtype Polymorphism

Signatures in TL are partial specifications. A signature *x :T* does not state that *x* has to be of type *T*, but that it has to satisfy at least the specification, i.e., *x* is at least of type *T*. An actual value for *x* may be of type *T* or of any subtype of *T*. This gives rise to a special form of polymorphism, the so-called *subtype polymorphism*: a function that expects a value of type *T* as parameter also accepts values of an arbitrary subtype of *T* as parameter. So the function is not restricted to parameters of one type, and therefore polymorphic.

Subtyping on Aggregate Types

A subtype of a tuple type can be constructed by restricting one or more of its component types A_i to a subtype $A_{i'}$ <:A_i and/or by appending additional components at the end of the tuple. In the following example *Employee* is a subtype of *Person*, but *Student* is not.

Let *Person* = **Tuple** *name :String age :Int* **end**
Let *Employee* = **Tuple** *name :String age :Int company :String* **end**
Let *Student* = **Tuple** *name :String matrNr :Int age :Int* **end**

TL offers a special construct that allows the definition of subtypes of tuple types without repeating the components of the supertype in verbatim: Signatures of named tuples, records or functions may be repeated using the keyword **Repeat** followed by the name of the signature(s). Employing this construct the type *Employee* may be defined as followed:

Let *Employee* = **Tuple Repeat** *Person company :String* **end**

This construct is also useful in other situations than the definition of tuple subtypes (see [Matthes 93]).

The subtyping relationship between record types is similar to the one on tuple types, but there are some differences: Additional components may be inserted at arbitrary positions in a type T and not only be appended at the end to construct a subtype of T. In addition, the components of the subtype do not need to be in the same order as the components in T. This enables the representation of multiple inheritance hierarchies. Subtype hierarchies over record types can result in acyclic directed graphs, not only in trees as in the case of tuple types.

The Rule of Contravariance

The subtyping relationships between function types is defined by the contravariance rule: A function type with signature S of the formal parameters and result type A is subtype of a function type with signature $S1$ and result type B, if A <:B and $S1$ <::S holds.

For the following function types, the subtype relationship *PersonToString* <:*EmployeeToString* holds:

Let *EmployeeToString* = **Fun**(*:Employee) :String*
Let *PersonToString* = **Fun**(*:Person) :String*

According to the subtype polymorphism the function

let *nameOfPerson(p :Person) :String = p.name*

can be used whenever a function of type *EmployeeToString* is expected.

3.1.5 Parametric Polymorphism

TL has a polymorphic type system. Beside subtype polymorphism, already presented in the previous section, it supports parametric polymorphism: It is possible to introduce type parameters into function and type definitions. The use of type parameters in function definitions yields polymorphic functions, whereas type definitions containing type parameters are type operators. Parametric polymorphism represents universal quantification over type variables, because the definitions are valid for arbitrary types. TL also supports a restricted form of the parametric polymorphism: the bounded parametric polymorphism. In this special case, the definitions are not valid for arbitrary types, but only for subtypes of the specified type, which is understood as the *bound* of a type parameter.

Polymorphic Functions

Polymorphic functions are functions including type parameters in their parameter list. The further parameters of the function may refer to this type. A simple example of a polymorphic function is the identity function that is presented in two versions:

let *identity1 (E <:***Ok***) (e :E) = e*
let *identity2 (E <:***Ok** *e :E) = e*

In the above examples E <:**Ok** introduces a type parameter. E is specified to be subtype of **Ok**, which is the supertype of all non-parameterized types. Thus arbitrary non-parameterized types may be passed as parameters to the functions. The second parameter of the functions is of the type E passed as first parameter.

The two versions of the identity function in the examples differ in the way they can be parameterized. The first function, *identity1*, may be parameterized in two steps (currying). It is a higher-order function and parameterizing it with a type in the first step yields an identity function over this type.

let *intId = identity1(:Int)*

Note that type parameters are preceded by a colon. In a second step this function can be parameterized with values of the type passed as first parameter.

The second version has to be parameterized in a single step with a type and a value of this type. The type parameter is omitted in most cases since it can be inferred from the type checking algorithm of the TL compiler. The following different applications of the above identity functions evaluate to the same result and show the different possible parameterizations:

intId(4)
identity1(:Int)(4)
identity2(:Int 4)
identity2(4)

A cunning aspect of TL is the possibility to combine the concepts of polymorphic and higher-order functions. This can be illustrated by a function to sort a list with arbitrary element type. The sorting algorithm may be implemented independent of the element type (polymorphism). The function to compare two elements needed for sorting is passed as parameter. For a list with elements of type *E* it has the following signature:

sort(e1, e2 :E) :Bool

Type Operators

Type operators are functions on the level of types, they take types as parameters and map them to a type. An example of a type operator is the operator *BinaryFun*, that may be used to define the type of binary functions over arbitrary types:

Let *BinaryFun* = **Oper**(*E* <:**Ok**) **Fun**(*:E :E) :E*

As in the case of functions, TL offers a short form for the definition of type operators:

Let *BinaryFun(E* <:**Ok**) = **Fun**(*:E :E) :E*

This operator maps every type *E* to the type *BinaryFun(E)*. *BinaryFun(Real)* is the type of the function *add* presented in section 3.1.2 and *BinaryFun(Int)* is the type of the predefined operators '+' and '–'.

Further, it is possible to define recursive type operators. An important application of recursive type operators is the definition of bulk data types, since the use of type operators allows abstraction from the element type of the collection. It is, for example, possible to define lists over arbitrary element types as generalization of *IntList*:

```
Let Rec List(E <:Ok) <:Ok =
    Tuple
      case nil
      case cons with head :E  tail :List(E) end
    end
```

The operations *new* and *cons* can be defined as polymorphic functions:

```
let new(E <:Ok) :List(E) =
  tuple case nil of List(E) end

let cons(E <:Ok  head :E  tail :List(E)) :List(E) =
  tuple case cons of List(E) with head  tail end
```

This shows that polymorphic functions complement the concept of type operators. The combination of both concepts allows the employment of generic code in the description of structure as well as of behavior.

Bounded Parametric Polymorphism

Bounded parametric polymorphism restricts the introduced type parameters to subtypes of a specified type. It may be used to overcome restrictions of parametric polymorphism as well as restrictions of subtype polymorphism. As motivating example the function *chooseOlder* is considered. *chooseOlder* is intended to compare the *age* component of two values $p1$ and $p2$ of an arbitrary subtype of type *Person* and to return the value with the higher age. This may be implemented using subtype polymorphism:

```
let chooseOlder1(p1, p2 :Person) :Person =
  if p1.age > p2.age then p1 else p2 end
```

The result of this function is of type *Person*, independent of the type of the parameters passed. When the function is applied to values of type *Employee*, this results in a loss of type information: the salary field of an employee passed as parameter may no longer be accessed in the function result, although it is the same tuple. The more specialized type information, that it is of type *Employee* not only of type *Person* is lost.

This loss of type information may be avoided by employing parametric polymorphism: Introducing a type parameter *P* allows to specify that the result of the function is of the same type as the parameters *p1* and *p2*:

```
let chooseOlder2(P <:Ok  p1, p2 :P) :P = ...
```

The type parameter *P* is specified to be subtype of **Ok**. Since no further assumption are made about *P*, the function has no information about the structure of *p1* and *p2*. For this reason no operations as, for example, field selections are possible on these parameters. So the comparison of ages may not be defined in the body of the function.

Bounded parametric polymorphism allows the restriction of type parameters to subtypes of a certain type. Now the function may be restricted to subtypes *P* of the type *Person* and the intended function *chooseOlder* can be defined:

```
let chooseOlder(P <:Person  p1, p2 :P) :P =
  if p1.age > p2.age then p1 else p2 end
```

Bounded parametric polymorphism is also permitted in definitions of type operators to restrict the possible types passed as parameter to the operator. A possible application is modelling relations. Relations permit only tuple types as element types of the relation. This restriction may be modelled in TL as follows:

Let *Relation(ElementType <:***Tuple end***)* = ...

3.1.6 Imperative Programming

TL offers functional as well as imperative programming features. The previous sections of this chapter are dedicated to the basic concepts and the functional aspects of the language. Concepts supporting imperative programming are presented in this section. Many of these concepts can be found in a similar form in common imperative programming languages as Modula-2 [ModISO 91]. They are, therefore, only mentioned here. A detailed description can be found in [Matthes 93].

Variables do not allow updates, if they are not explicitly defined as mutable variables in TL. The only exception are *arrays*. The components of arrays are always mutable. The definition of mutable variables is described in the first part of this section.

TL supports different constructs for the definition of loops: *while*-loops, *for*-loops, and the most general construct *loop* which is left with the *exit* statement. A further way determining the control flow of a TL program are exceptions. The definition and use of exceptions and exception handlers is described in the second part of this section.

Mutable Variables

Variables in TL are not mutable, if they are not explicitly defined as such. Value bindings established by the *let*-construct (see section 3.1.1) represent constant definitions. A repeated binding of the same identifier employing the *let*-construct does not change the original binding, but produces a *new* constant with the same name.

A mutable variable has to be introduced by the keyword **var**:

let var *x = 3*

This binding defines a mutable variable x and initializes it to the value 3. A mutable variable can be updated by destructive assignment using the operator *:=*.

x := 4

The assignment operator has the following signature:

*:=(A <:***Ok var** *lValue :A rValue :A) :***Ok**

An assignment, therefore, evaluates to the value **ok**. The operator *:=* is not built-in, it may be redefined by the user.

Exception Handling

In addition to the handling of exceptions raised by exceptional conditions as arithmetic overflow and division by zero, TL supports user-definable exceptions. A special value constructor **exception** is provided for this purpose:

```
let noCredit = exception "No Credit!"
```

The string "No Credit!" is used for an identification of the exception if it is propagated to the top level of the system. An actual exception can be raised at all points in a program signaling the related exceptional condition:

```
let withdraw(var account :Int  amount :Int) =
  if amount <= account then
    account := account - amount
  else
    raise noCredit
  end
```

An exception terminates the actual execution (evaluation) and propagates along the calling hierarchy of the enclosing functions until an adequate handler is found or the top level of the system is reached. In TL an exception handler is defined as follows:

```
try
  withdraw(petersAccount  300)
  print.string("Transfer succeeded")
when noCredit then
  print.string("Overdrawn")
else
  print.string("Unexpected exception occurred")
end
```

In order to enable exception handling, the operation that raises the exception has to be enclosed in a *try*-construct. The *when*-branches of this construct are dedicated to special exceptions. The specified handler is only evaluated for the named exception. An exception not explicitly captured by one of the *when*-branches is propagated further, if no *else*-branch is provided by the *try*-constructs. Otherwise, the handler defined in the *else*-branch is evaluated.

3.1.7 Modules, Interfaces, and Libraries

The possibility to define modules is one of the most important structuring facilities in modern programming languages. As in the language Modula-2 [ModISO 91] large programs are dividable into several modules. In TL each module consists of an interface and an implementation module. The interface contains names and supertypes of abstract data type definitions, names and signatures of functions and type operators, and names and types of variables that are defined and exported by the module. Additionally, it is possible to define types in an interface.

Modules and interfaces can import other modules and use the types, type operators, functions, and variables exported by these modules. References to the imported objects have to be preceded by the name of the exporting object. A function *cons* exported by a module *list*, for example, is referenced by the name *list.cons*.

TL offers a further structuring facility for larger programs. Interfaces and modules are groupable into libraries. A library may also contain other libraries.

3.2 Syntax Extension in the Tycoon Environment

The syntax extension tool presented in [Schröder 93] (see also [Cardelli et al. 94; Kohlbecker 86]) is based on a two phase compiler model. The first phase is the *frontend* taking source code to an intermediate machine independent representation. The second phase is the *backend* taking the machine independent representation to the target code of an existing machine. The syntax extension is solely concerned with the frontend of this compiler model. The syntax extension changes the frontend of the compiler for a base language. The frontend can be divided into five phases: the first three phases are scanning of the source code, the parsing with generation of a parse tree, and the generation of an abstract syntax tree derived from the parse tree. Since the used syntax extension tool leaves the abstract syntax of the base language unchanged the syntax extension tool is only concerned with these three phases.

As mentioned above the syntax extension modifies the frontend of an existing compiler. Changing the code implementing the frontend is a very complex and error prone task. For this reason a different approach is proposed by [Schröder 93]. The frontend is specified in an abstract description language that allows automatic generation of a frontend (by scanner-, parser-generators). To modify the frontend, it is sufficient to change this description of the frontend. It is even possible to generate the frontend incrementally using incremental parser generators.

Syntax extensions consists of two phases. In the first phase (definition phase) the new syntax extending the syntax of a given language is defined. For each new syntactic construct a semantic interpretation has to be specified. The use of these new syntactic constructs initializes the second phase (expansion phase) of the syntax extension: the syntactic constructs are expanded according to the specified semantic interpretations.

A system for syntax extension has to support two languages. The first language, extension language TLExt, allows the formulation of the syntax extensions. The extensions of the syntax refer to an existing language, the *base language* of the syntax extension. Figure 3.1 taken from [Cardelli et al. 94] illustrates this scenario. An extensible grammar package consisting of a grammar checker and a parser generator extends the given base language TL. For this, object languages $OL_0 \cdots OL_n$ are defined providing a grammar specifying the syntax of a language. Parse tables are generated from the syntax description by the parser generator and resemble an extensible parser from which internal parse tree information is mapped to abstract syntax trees of the language TL. Programs of the object languages are valid semantic actions denoted by the corresponding syntax and TL expressions exist for these. The TL expressions are valid

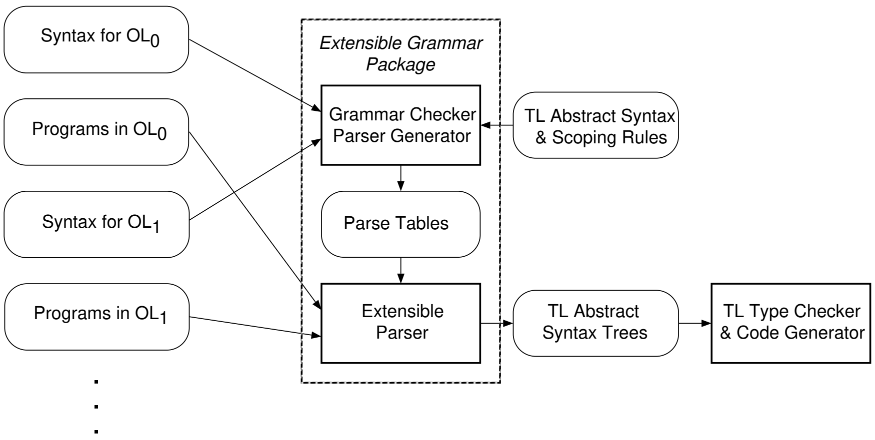

Figure 3.1: Syntax Extension

programs and hence subject to the underlying type checker and code generator for different target languages.

Description of a Syntax Extension

The description of a syntax extension consists of two parts. A syntactic construct and a semantic interpretation of this construct. The syntactic constructs are described by so called *patterns*. Patterns are similar to productions of context free grammars except that they allow placeholders in addition to the terminal and non-terminals of a grammar. The placeholders are identifiers that are assigned to non-terminals in the pattern. They introduce names for the constructs generated by the associated semantic interpretations. The introduced names for these constructs allow a reference to them in the semantic interpretation. The following example is taken from [Schröder 93]:

"for" "each" id = ideG "in" table = valG ":" pred = valG "do" act = bndsG "end"

and shows how the placeholders *id*, *table*, *pred* and *act* are introduced in a syntactic expression for the use in a syntax extension tool. The placeholder *id* is for any valid expression resembling an identifier in the base language. *table* and *pred* are placeholders for value expressions, and *act* is a placeholder for a binding expression in the base language. Such augmented syntactical patterns are a central construct to a valid description of a syntax extension.

Abstract syntax trees are used to specify the semantic interpretation. Instead of describing the syntax trees by constructors, the base language extended by the placeholders is employed for this purpose. This allows a user-friendly definition of the semantic interpretations in a syntax extension.

A syntax extension consists therefore of a set of declarations. There are three possible forms of declarations: a declaration can introduce a new production, or it can replace an existing production, or it can extend a production by further alternatives. Each production must have at least one alternative and each alternative is of the following form:

patterns $\Rightarrow$ *semantic interpretation*

A kind of parameterization of the productions is possible by introducing inherited attributes. The attributes can be referenced in the semantic interpretation analogous to the placeholders introduced in the pattern. If inherited attributes are used in a production a sort is assigned to each of the attributes. This enables the system to check whether attributes used in the semantic interpretation are conform to their sort.

For each production the according derived attributes belonging to the production have to be determined. The derivation of such attributes is defined by the semantic interpretation of the rule. A sort is assigned to each derived attribute of a production. A detailed discussion of this can be found in [Schröder 93].

Variable captures are one of the most crucial problems in connection with syntax extension technology. They occur during the expansion of the rules where local bindings can occur as placeholders and therefore conflicts from preexisting local/global bindings in the semantic interpretations of a rule are possible. In [Schröder 93] a novel approach to resolve the inherent in the expansion process of syntax extensions is described. Further a classification of occurring binding problems on syntax extensions is given (see also [Kohlbecker 86] and [Cardelli et al. 94]).

Chapter 4

A Typed Comprehension-Based Query Language

List comprehensions are known from functional programming as a concise notation expressing list manipulations [Bird, Wadler 88; Field, Harrison 88; Peyton-Jones 87] (see also chapter 2). As described in [Trinder 92], comprehensions may be employed as a query language. The query language developed in this work is based on comprehensions. It relaxes the restrictions of (list) comprehensions in the following ways:

Arbitrary bulk types as input: In a list comprehension one or more lists may be specified by a generator as input for the list manipulation. When the comprehension is considered as query, these ranges have the role of the queried bulks. In list comprehensions, as in most other query languages, the ranges are restricted to one type of bulk, namely, lists. The presented query language permits arbitrary bulk types as ranges. It is even possible to use different bulk types as inputs inside a single query (*mixed queries*). To achieve this generality, a common supertype for all bulk types is defined, and iteration abstractions are used as uniform intermediate representations.

Arbitrary bulk types and non-bulk values as result: The result of a list comprehension is in every case a list. Similarly, in many query languages query results are fixed to be of a specific bulk type, e.g., tables in SQL. Even if the result is a single value, e.g., after application of an aggregate function, the result is a bulk containing this single value. It would be more adequate in this situation, to allow non-bulk values as results. This is the case in the proposed query language. Additionally, the user may choose arbitrary bulk types as container for the elements of the query result.

Trinder proposes to use the comprehensions directly for the formulation of queries [Trinder 92]. In order to augment the flexibility, a different approach is chosen in this thesis. Each query consists of a comprehension construct, forming the kernel of the query. This kernel determines the flow of control, especially the order of iteration over the elements of the ranges. The sequence of elements defined by this kernel is called *comprehension result*. The query is completed by an additional construct which is called *context* in the sequel of the text. The context determines the actions to be performed on the elements of the comprehension result.

This may, for example, be the insertion into a bulk type or the formulation of quantifiers and aggregate computations for a comprehension. The idea of a context is implicitly used in many other query languages, e.g., by offering aggregate functions. A more explicit treatment is found in the database programming language DBPL [Schmidt, Matthes 92]. This language supports *selective* and *constructive access expressions* that may be used in different contexts. An overview of the treatment of contexts in DBPL and other query languages is given in the appendix.

Many contexts require a complete evaluation of the comprehension. The number of the elements in the query result, for example, may only be computed by evaluating the complete comprehension. The presented language, on the other hand, supports the definition of a context allowing lazy evaluation of the comprehension result. The comprehension result is evaluated element-by-element on request. Such contexts enable querying bulk types that are potentially infinite, or not completely determined in the moment of query evaluation. An example for such bulk types are input streams. This feature is also useful in the case that only few elements of the query result are requested, or if the result is extremely large (calculation of the whole result exceeds given time and/or space limits).

The query language is independent of a specific data model. Since arbitrary bulk types with arbitrary element types are possible inputs to the queries, the query language is not restricted to a bulk type or element structure (as for example flat relations) characteristic for a concrete data model. Special requirements of data models can be supported by contexts defined for this purpose. Furthermore, several kinds of nesting queries are possible. This is especially useful in the presence of nested structures.

The query language is integrated into a host language. This is accomplished by employing syntax extension technology. The integration has a two-fold effect towards the query language: On the one hand, the query result has to represent a valid expression of the host language. On the other hand, arbitrary expressions of the host language may be used inside the comprehension, e.g., for the formulation of filters. This introduces computational completeness into the query language.

The language TL [Matthes 93] is used as host language[1]. This language may be interpreted as well as compiled. This is also true for the extended language. It, therefore, is possible to formulate ad-hoc queries that are interpreted instantly on the top-level of the system. However, queries may also be embedded into programs and compiled as part of them. Since TL is a typed language the query language is realized in a typed environment. This implies that a type has to be assigned to the query result. This assigned type depends on the ranges, the target expression, and the context.

The proposed framework for the query language does not manifest a fixed number of specific contexts; instead it supports a small number of different classes of contexts. The classes may be parameterized to yield a specific context. Suitable parameters are supported by a special context library. This library may be extended and adapted to support new requirements and applications of the query language.

This chapter starts with an introduction to the syntax of the query language. First, the building blocks of the kernel of the query language are presented, followed by the definition of a grammar for the query language. In the second section typing and scoping aspects of the

[1] A short description of this language is presented in chapter 3.

comprehension which form the kernel of the query language are considered. The third section is devoted to the contexts. Contexts are classified and an extensible framework for contexts is presented. In order to illustrate the variety of possible contexts, a set of representative contexts is described. Finally, a short overview concerning typing aspects of the contexts is given.

It is a novelty of the presented query language to allow different bulk types within a single query (called *mixed queries* in this thesis). Additionally, arbitrary bulk types can be specified as contexts. The semantic of mixed queries in different contexts is examined in section 4.4. This discussion is based on the theoretical approaches presented in chapter 2. The last section (4.5) of the chapter is dedicated to more advanced features of the query language. Naming and parameterization of queries may be introduced into the presented framework in a rather natural way. The support of recursive queries is also discussed.

4.1 Introduction of the Syntax

Comprehensions are chosen as notation for the kernel of the query language. The design decisions in developing the syntax for the query language are driven mainly by the following requirements:

Concise and Uniform Notation: List comprehensions are a well-understood notation used in functional programming. They are a concise, declarative notation for list manipulations that may be used as query language [Trinder 89; Trinder 92]. Comprehensions may be easily extended to other bulk types resulting in a uniform notation for queries over different bulk types (see chapter 2).

Expressive Power: It can be shown that any relational calculus query can be translated into an equivalent list comprehension expression [Trinder 89, pp.124–131]. A language based on list comprehensions is at least relational complete.

The chosen embedding of the query language via syntax extension allows the use of arbitrary expressions of the host language. This introduces computational completeness into the query language.

Clean Semantics: List comprehensions are a well-understood concept with a good theoretical foundation. The extension of comprehensions to other bulk types and their semantics has been examined in several publications (see chapter 2). A set of rewrite rules exists that translate a comprehension into an expression of an enriched lambda calculus. This leads in a natural way to an implementation in the Tycoon language, since it is also based on the lambda calculus.

Syntactic Integration: An important point for the syntax of the query language is to fit well into the syntax of the embedding language. The proposed language in this chapter is comprehension-based and uses a classical comprehension notation. The use of a syntax extension tool enables the user to adapt the syntax to his preferred style, e.g., a keyword-fashion style [Ghelli et al. 92]. The syntax of the language can be changed easily by altering the grammar driving the syntax extension tool.

Figure 4.1: Syntax Chart with Components of a Query

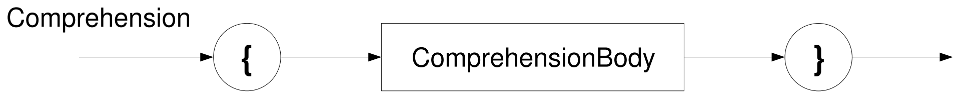

Figure 4.2: Syntax Chart of the Comprehension

Seamless Embedding: The aim of the design process is an embedding of the query language into a host language. This is accomplished by using a syntax extension tool. The query language is embedded into the host language by extending the grammar of the Tycoon language. The extended front-end of the host language accepts comprehension queries as well as valid host language expressions.

The first part of this section introduces the building blocks of a query. A query is split into a comprehension and a context. Each of the building blocks of the comprehension is discussed in detail in one of the following subsections. The contexts are considered in section 4.3. The examples used for illustration of the discussions are all based on a common schema. The presentation of this schema together with the first simple query examples precedes the detailed discussion of the buliding blocks.

4.1.1 The Building Blocks of a Query

The query language is comprehension-based. The comprehensions form the kernel of the query language. For every query the comprehension is augmented with a context (see figure 4.1). Contexts are considered in the next section. The syntax of the comprehensions used in the query language is very similar to the one presented in [Trinder 92]. A comprehension consists of a comprehension body enclosed in curly brackets (see figure 4.2). The first part of the comprehension body is a *target expression* followed by a vertical bar ("|") and one or more qualifiers (see figure 4.3). A qualifier can be a *generator* or a *filter*. The first qualifier is always a generator. Filters are separated by a colon from the previous building blocks, further generators by a comma (see figure 4.4).

A generator introduces a *range variable* and a *range*; these components are connected by an arrow pointing from the range to the range variable (see figure 4.5).

Example Database

For the examples presented in the next sections, the following type definitions are assumed.

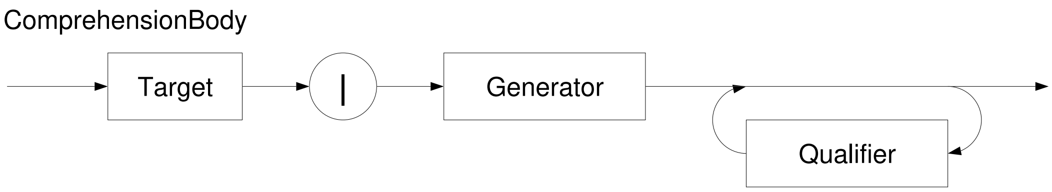

Figure 4.3: Syntax Chart of the Body of a Comprehension

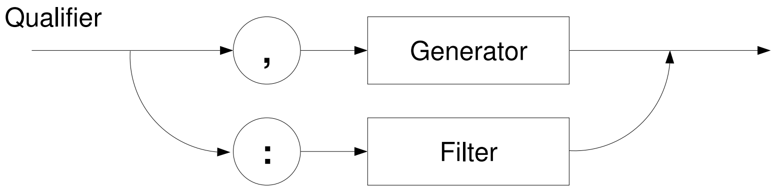

Figure 4.4: Syntax Chart of a Qualifier

Let *Person* =
    Tuple
        name :String
        var *age :Int*
    end

Let *City* =
    Tuple
        name :String
        country :String
        habitants :Int
    end

Let *Family* =
    Tuple
        mother :Person
        father :Person
        children :Set(Person)
        home :City

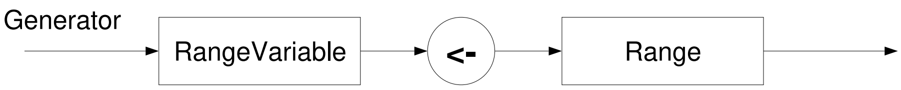

Figure 4.5: Syntax Chart of a Generator

end

Additionally, the bulk values *persons*, *cities*, and *families* are used, where *persons* is a set of elements of type *Person*, *cities* is a list of elements of type *City*, and *families* is a set of elements of type *Family*.

The following example is a simple query in the described syntax notation:

get *list* $\{ p.name \mid p \leftarrow persons : p.age < 18 \}$

The example is a one-range query with a context *list*. Contexts are introduced by the keyword **get** in the query language. The comprehension body contains a target expression *p.name*, a generator $p \leftarrow persons$, and a filter $p.age < 18$. The query computes the list of the names of all persons younger than 18 years. The same query without filter looks as follows:

get *list* $\{ p.name \mid p \leftarrow persons \}$

It returns the list of the names of all persons.

Generators

A generator introduces a new identifier, a so called *range variable*. This variable is bound to a collection constituting the range for the variable. Syntactically, the range variable and the range are connected by an arrow pointing from the range to the range variable.

During the evaluation of the query, all elements of the range are successively assigned to the range variable. This identifier holds one single element of the range at a time and amounts to an iteration over the range.

It is the aim of the query language to allow queries over different bulk types using a uniform syntax. In contrast to most common query languages the ranges are not restricted to one kind of bulk type (e.g., tables in SQL [ISO9075 92]). Any structured homogenous bulk type is permitted as range, presuming it allows iteration over its elements[2]. Additionally, arbitrary expressions evaluating to bulk types are accepted inputs to a query. The element type of the range is not restricted in any way. It may be an arbitrary type of the language TL. The generality of this approach admits the following enhanced forms of queries:

Mixed queries: Having a query with several ranges, it is possible to employ different bulk types as ranges inside a single comprehension. This kind of queries is called *mixed queries* in this thesis. The impact of mixed bulks as ranges on the query result is discussed in section 4.4, presenting a clean semantics for mixed queries. The considerations are related to the bulk-morphisms presented in [Ross 92; Beeri, Ta-Shma 94] (see also chapter 2). The following is an example of a mixed query:

[2] For implementation reasons this restriction is later narrowed to the constraint that the bulk type of the ranges have to be in subtype relationship to a common supertype; see section 5.3.1.

$$\textbf{get } \textit{list} \ \{ \ \textbf{tuple } c.name\ c.age \textbf{ end} \mid f \leftarrow \textit{families},\ c \leftarrow f.children \ \}$$

computing the list of all 'children' of all 'families'. Note that '*families*' is a list and *f.children* is a set.

Nested queries: With the choice of certain contexts for the comprehension, the result of a query evaluates to a bulk type. Such queries are valid ranges for other queries forming subqueries. If comprehensions are used as ranges they form subqueries. This allows the formulation of nested queries, as illustrated in the following example:

$$\textbf{get } \textit{list} \ \{ \ f \mid c \leftarrow \textbf{get } \textit{bag} \ \{ \ c \mid c \leftarrow cities : c.country == \text{"}I\text{"} \ \}$$

$$,\ f \leftarrow families : f.home == c \ \}$$

The subquery computes all cities in Italy. The entire query returns the list of all families living in Italy.

Thus nested queries are naturally integrated into the presented framework. As described in [Cluet, Moerkotte 94] for a SQL-like query language, there are several other kinds of nesting queries by using queries in other parts of the comprehension, as for example in the filter or in the target expression. These kinds of nesting are discussed together with the associated building blocks.

Queries over nested collections: Since arbitrary TL types are valid element types of the queried bulk types, it is possible to define nested bulk types as ranges. The scope of a range variable starts just behind the introducing generator. This admits subsequent generators to refer to this identifier, employing bulk-valued components of this range variable as their ranges. This is illustrated by the following example:

$$\textbf{get } \textit{count} \ \{ \ c \mid f \leftarrow families,\ c \leftarrow f.children \ \}$$

where *count* is a context calculating the number of elements in a comprehension result. The query computes the total number of all children for each family. The set-valued component *children* from the value *f* of type *Family* is used as range of the second generator. Note that the second range depends on the first range variable.

Filters

Filters are predicates restricting the range of the preceding generator. Only those elements of the ranges of the preceding generator that fulfil the predicate specified by the filter are taken into account for the rest of the comprehension.

The filter may refer to range variables introduced in generators preceding the filter and to identifiers defined in the environment of the query (global variables). The filter predicate may construct arbitrary expressions using these identifiers and literals. The only restriction is that the expression has to yield Boolean expressions, when evaluated for concrete elements of the ranges. Especially the filter may use functions and operators defined in the environment of the query. Therefore the expressive power of the filter strongly depends on the environment of the query in contrast to many other query languages with a fixed set of operators for the construction of filters; e.g., SQL.

Arbitrary TL values can be employed in the construction of filters. Since query results are TL values, they may participate in the filter construction. This yields a further kind of nesting queries illustrated by the following example:

$$\textbf{get}\ \mathit{list}\ \{\ f \mid f \leftarrow \mathit{families} : 3 < \textbf{get}\ \mathit{count}\ \{\ c \mid c \leftarrow \mathit{f.children}\ \}\ \}$$

where *count* is defined as above. The example query computes the list of all families with more than three children.

According to the syntax charts in figure 4.3 and 4.4 it is possible that a filter directly succeeds another filter. This is interpreted as a conjunction of the filter predicates.

Target Expressions

The *target expression* is the first part of the comprehension. The elements of the Cartesian product of all ranges in the comprehension are potential candidates for the comprehension result, but only the tuples fulfilling all filters specified in the comprehension are taken into account. The target expression is evaluated only for these elements. The sequence of elements resulting from these evaluations form the comprehension result.

The target expression may refer to all range variables introduced in the comprehension and to all identifiers visible in the environment of the query. From these variables and arbitrary literals, the result elements may be constructed. For this purpose the value constructors of TL and functions, which are defined in the environment of the query, are applicable. Of special importance are the dot notation and constructors for tuples and records allowing the selection of components of range variables and the combination of values and/or components of different range variables, respectively.

As in the case of filter construction, it is possible to employ queries in the target expression. This is still another method to introduce nesting into queries. The resulting kind of nesting may be used to yield nested collections as a query result. This is illustrated by the following example[3]:

$$\begin{array}{l}\textbf{get}\ \mathit{list}\ \{\ \textbf{tuple}\ \mathit{c.name}\ \textbf{get}\ \mathit{list}\ \{\ f \mid f \leftarrow \mathit{families}\ : \mathit{f.home} == c\ \ \}\ \textbf{end} \\ \qquad\qquad\quad \mid c \leftarrow \mathit{cities}\ \}\end{array}$$

The query computes a list of tuples, each consisting of the name of a city and the list of all families living in the city.

4.1.2 Grammar of the Query Language

The query language is to be understood as an embedded language in a given host language. The syntax of the host language is extended by the syntax of the query language. The resulting

[3] A similar example may be found in [Cluet, Moerkotte 94].

Value ::=		...
	|	Query
Query ::=		Context Comprehension
Comprehension ::=		"{" *Value* "|" Generator { Qualifier } "}"
Qualifier ::=		"," Generator
	|	":" *Value*
Generator ::=		*Ide* "<-" Range
Range ::=		*Value*
Context ::=		...

Figure 4.6: Grammar of the Query Language

syntax describes a language of all valid expressions of the host language and, in addition, the expressions of the query language. The chosen host language is TL. The grammar of the query language (see figure 4.6) extends the grammar of TL [Matthes 93, appendix A]. The production rules of the query language introduce new terminal and non-terminal symbols, but there are also non-terminals shared by both languages. To facilitate distinction between the non-terminals of the host language and those of the query language, the non-terminals of the host language are italicized.

The production *Value* of TL is extended by an additional alternative production *Query*, since query results are values of the language TL. The production rules of the grammar may be directly derived from the description of the building blocks of the query language (see page 46). Target expressions, ranges, and filter predicates are values of the Tycoon language. So they are described by the non-terminal *Value*. The grammar rules for the context will be presented in section 4.3.2.

The syntax presented for the query language is based on the classical mathematical comprehension syntax. It should be viewed as a proposal for one possible syntax rather than as the only syntax of the query language. It is easy to offer a keyword-based syntax instead, for example, by replacing the symbols of the language keywords according to figure 4.7. When employing this syntax the example query from page 48 looks as follows:

get *list* **of each** *p.name* **from** *p* **in** *persons* **where** *p.age* < *18* **end**

The proposed keyword-style syntax does not affect the structure of the language and the statements made about it. It would even be possible to offer both syntax styles in a single grammar by adding an alternative right side for each production which is a copy of it with

Symbols	Replace by
$\{ \cdots \}$	**of each** $\cdots$ **end**
$\mid$	**from**
$\leftarrow$	**in**
$:$	**where**

Figure 4.7: Mathematic vs. Keyword Style

the replacements given in table 4.7. In this setting it would however not be possible to avoid mixing of keyword and mathematical notation in a single query.

4.2 Typing and Scoping in a Comprehension

The considerations in this section are based on an abstract representation of queries. It is assumed that queries are of the general form:

$$context \ \{ \ t \mid rv_1 \leftarrow r_1 : f_1, rv_2 \leftarrow \ r_2 : f_2, \cdots, rv_n \leftarrow \ r_n : f_n \ \}$$

The rv_i are the range variables, the r_i are the ranges and the f_i are the filters for i ranging from 1 to n ($n \geq 1$). t is the target expression. It is assumed that each generator is followed by a filter. Filters have no influence on the typing of the comprehension and do not introduce variables relevant to the comprehension. So this assumption does not restrict the generality of the considerations.

The first generator introduces a range variable rv_1 and a range r_1. Range r_1 is a bulk with elements of type E_1. The type of r_1 depends on the element type E_1. This is expressed by

$$r_1 : Bulk_1[E_1]$$

where $Bulk_1$ is the type operator representing the bulk type of r_1. The range variable rv_1 is used to iterate over the elements of r_1. It is therefore of type E_1. The scope of rv_1 starts behind the generator and ends with the closing bracket of the comprehension. Additionally it includes the target expression t. The range r_2 also represents a bulk. Since it is in the scope of rv_1 it may depend on the value of this variable. Therefore r_2 must be represented as a function

$$r_2 : E_1 \rightarrow Bulk_2[E_2]$$

in the general case. Pursuing this scheme yields the type

$$r_i : E_1 \ \times \ E_2 \ \times \ \cdots \ \times \ E_{i-1} \rightarrow Bulk_i[E_i] \qquad i = 2, \cdots, n$$

for the general case of range r_i and the type

$$rv_i : E_i$$

for the range variables. As in the case of rv_1 the scope of each range variable rv_i consists of the target expression and of the comprehension starting behind the generator introducing rv_i.

Now the types of the filters are considered. Filters are Boolean expressions. Filter f_1 may depend on the range variable rv_1 introduced by the first generator. This fact results in the type

$$f_1 : E_1 \rightarrow \mathit{Bool}$$

Because of the scoping rules for the range variables, filters may depend on all range variables already introduced, i.e., a filter f_i may depend on the range variables $rv_1, \cdots, rv_i$ which have element types $E_1, \cdots, E_i$, respectively. So, every filter f_i may be expressed as function of the following type

$$f_i : E_1 \times \cdots \times E_i \rightarrow \mathit{Bool} \qquad i = 1, \cdots, n$$

The type of the elements of the comprehension result is determined by the target expression. The target expression t may depend on all range variables defined in the comprehension. Since the range variables are of types $E_1, \cdots, E_n$, respectively, t is expressible as function of the following type:

$$t : E_1 \times \cdots \times E_n \rightarrow E_r$$

where E_r is the element type of the comprehension result.

A query consists of a comprehension and a context. The type of the query result is influenced by the context. In most cases it also depends on the type E_r of the elements of the comprehension result as well. The static semantic of the query regarding the type of the contexts and the type of the comprehension and their interaction is discussed in section 4.3.

A comprehension defines a sequence of elements. Comprehensions do not represent values of the language TL. Since comprehensions and contexts are implemented seperately, comprehension results appear as intermediate results in the implementation. For this reason a type has to be assigned to comprehensions. Since this type depends on the element type E_r, a type operator [Cardelli, Wegner 85] has to be used to describe the type. The concrete type operator depends on the implementation.

4.3 Comprehensions in Different Contexts

The building blocks of the comprehension — generator, filter, and target expression — may be identified in a similar form in most query languages. They may be viewed as the essence or kernel of a query language[4]. Generally those building blocks determine a sequence of elements in the following way: The target expression is applied to all those elements of the Cartesian product of the ranges that pass all filters. In case of the comprehensions this sequence is called

[4] This kernel can express the three primitives of relational calculus: selection σ, projection π, and join $\bowtie$ [Beeri 92b].

'comprehension result' (see section 4.1). The obtained sequence is not considered as the result of the entire query.

In most query languages the elements are collected in some specific kind of bulk, for example a sequence in FIBONACCI ([Albano et al. 93]) or a table in SQL ([ISO9075 92]). This implicit fixing of the last processing step for the result elements is avoided in the proposed query language in order to support an extensible set of possible last processing steps. This leads to the introduction of a further abstract building block into the query language, called *context* in the sequel of this thesis. The context of a comprehension defines an environment in which the comprehension is evaluated. It is the aim of this environment to fix the final processing steps. The insertion of the elements into a collection may then be viewed as one possible concrete form of the context as abstract building block. Other contexts are aggregate functions as *sum* or *count* which are supported by many query languages, e.g., SQL, and FIBONACCI. The language DBPL [Schmidt, Matthes 92] is an example of a database programming language with an explicit treatment of a concept similar to the contexts presented here. It supports *selective* and *constructive access expressions*, that are applicable in different contexts allowing, for example, the construction of a new relation and the iteration over selected relation elements.

In the first part of this section an overview over common contexts used in query languages and a classification of such contexts is given. In the second part of this section it is shown how the contexts are integrated into the proposed query language. This leads to an extensible and flexible framework for contexts. Although the presented query language does not have a fixed set of contexts, examples of contexts are needed for illustration. A set of representative contexts illustrating the wide variety of possible contexts is presented in the third part of this section. Typing aspects of contexts and especially of the chosen set of contexts are discussed in the last part of this section.

4.3.1 A Classification of Contexts

A systematic overview of some contexts supported by other database and query languages, namely, FIBONACCI [Albano et al. 93], O_2SQL [Bancilhon et al. 92], ADAPLEX [Smith et al. 83], FAD [Bancilhon et al. 87], Napier [Morrison et al. 89], Machiavelli [Ohori et al. 89], and DBPL [Schmidt, Matthes 92] is given in the appendix (see A). The contexts found in literature are also used as basis for the following classification of contexts:

Bulk types are the implicit standard context in many query languages. The elements of the comprehension result are collected in the according bulk type if no explicit context is declared.. In most query languages this class of contexts is restricted to one bulk type, for example, tables in SQL, sequences in FIBONACCI.

Aggregate functions accumulate a single value from all elements in the comprehension result. In each step the already accumulated result of the aggregation is combined with the next element. The kind of the aggregate function is determined by the operator chosen for the combination (e.g., "+" for the summation of elements). In addition, in some cases a bottom value is needed. It is used as return value if the comprehension result is empty. Examples for language constructs describing aggregating functionality are *SUM* and *COUNT* in SQL, *count* in FIBONACCI.

Quantifiers check whether a specified predicate holds for all elements or at least one element of the comprehension result, respectively. The query result is a single Boolean value. Quantifiers may be viewed as special aggregate function with $\wedge$ and $\vee$ as combining operators and *true* and *false* as bottom elements, respectively.

Selections of single elements return one element of the comprehension result. There are two variations: a) The comprehension result may contain several elements, and an arbitrary element is chosen non-deterministically. An example is the operator *pick* in FIBONACCI. b) It is assumed that the comprehension result consists of a single element: This element is returned as query result. If the number of elements in the comprehension result is not equal to one, this leads to a failure. Examples are the operators *the* in FIBONACCI and *element* in O_2SQL.

Combinative contexts are contexts that have to be (or may be) combined with other contexts. This class contains contexts specifying duplicate elimination, sorting, grouping, and flattening to be applied to the comprehension result. Examples are *flatten* in O_2SQL and *setof* in FIBONACCI. The contexts of this class are used in combination with other contexts, frequently with the implicit standard bulk type context.

Iterations with side-effect apply some operation to each element of the comprehension result. This class of contexts leads to constructs that are no longer queries in the strict sense. Their intention is to perform side-effecting operations and not to calculate a result as in the case of a query. They are not excluded from the discussion, since they need the same kernel constructs, i.e., may be expressed by employing comprehensions. Examples are operations that update, insert, or delete the sequence of elements specified by the comprehension. This kind of contexts is supported in the languages FAD, SQL and COOL. An alternative approach is to allow the definition of loops where the loop body is evaluated for each element of the comprehension result. In the loop body, an arbitrary side-effecting operation may be specified. An example are *constructive access expressions* in DBPL. They can be embedded into *for*-loops.

4.3.2 An Extensible Framework for Contexts

It would be possible to support a fixed set of contexts for the presented query language, but in order to achieve an extensible and more flexible framework for contexts, a different approach is chosen. It is based on the following more general classification of contexts. Contexts may be

result oriented: The intended effect of the query is to compute a value. The value may be non-bulk or bulk. Contexts of this class are introduced by the keyword **get** in the presented query language. The concrete context is determined by an object which is provided by a special context library.

side-effect oriented: The result of the query is the application of some side-effecting operations to the elements of the comprehension result (Iteration with side-effect). The keyword **do** followed by the operation to be performed for the specified elements is used to define contexts of this class.

Query	::=	Context Comprehension
Context	::=	**get** Value
	|	**do** Value

Figure 4.8: Extension to the Grammar of the Query Language

The keywords **get** and **do** introducing the different classes of contexts are part of the syntax of the query language (see figure 4.8).

The result oriented queries are separable into some fine-granular classification in regard to the evaluation of the comprehension. Some contexts require a *complete evaluation* to yield the query result. Examples of these contexts are the summation of the comprehension elements *sum* and the computation of the cardinality *count*. For some contexts it is sufficient to evaluate a part of the comprehension (*partial evaluation*). An example is the context implementing the existential quantifier *some*: the evaluation of the comprehension can terminate if the first element fulfilling the predicate of the quantifier is found. A special group of contexts is introduced in this work. It performs an evaluation of the comprehension on demand. These contexts allow to query inputs that are potentially infinite as for example input streams. Lazy evaluation is supported facilitating the treatment of extremely large comprehension results. This further classification is of special interest for the implementation of the contexts (see section 5.3.4).

A comprehension augmented by a side-effect oriented context is not a query in the strict sense. The side effecting function specified for the context is applied to each element determined by the enclosed comprehension; operations performing output are possible applications:

$$\textbf{do}\ \textit{print.string}\ \{\ c.name \mid c \leftarrow cities : c.habitants < 10000\ \}$$

The construct prints the names of all cities with less than 10.000 habitants. Another interesting utilization are update operations:

$$\textbf{do}\ \textit{incrementAge}\ \{\ p \mid p \leftarrow persons\ \}$$

More enhanced applications of the side-effect oriented contexts are also possible, e.g., the visualization of the query result. The graphical editors presented in [Kirch, Müßig 92] can be utilized for this purpose.

$$\textbf{do}\ \textit{editCities}\ \{\ c \mid c \leftarrow cities : c.habitants < 10000\ \}$$

where *editCities* is a function that produces a graphical presentation of the data records *city* on the screen. The browsing through the elements of the comprehension result can be controlled interactively by the user.

4.3.3 A Set of Representative Contexts

As described in section 4.3.2 the presented query language does not support a fixed set of contexts. Instead it offers an extensible framework for contexts. The concrete contexts are defined in a supporting library that may be extended and adapted by the user of the query language.

A set of representative contexts is described in this section. For this reason they are grouped into the classes presented in section 4.3.1. They are chosen to show the flexibility of the approach and to give an impression of the wide variety of contexts that may be realized in the framework. The implementation of these representative contexts described in sections 5.2.4 and 5.3.4 may be used as template for the implementation of new similar contexts.

Bulk Types

There exists a large number of bulk types that may be chosen as contexts for comprehensions, distinguishable by properties as duplicate elimination and order and by the operations for access and update defined on them (see section 4.4). The contexts *list*, *bag*, *set*, and *relation* are taken as representative examples for bulk type contexts.

Lists and bags represent bulk types with and without respecting the order of their elements, respectively. Both bags and lists may contain duplicates. Sets and relations are chosen as examples for bulk types with duplicate elimination. Prerequisite of duplicate elimination is a notion of equality that defines in which case two given elements are regarded as equal and therefore are duplicates. Identity is the notion of equality (for constructed values) directly supported by the host language TL (see section 3.1.3). In order to allow other forms of equality, as for example value equality, to be defined for the duplicate elimination, the context *set* is equipped with a parameter *equal*. This parameter is a function of type $\mathbf{Fun}(: E_r : E_r) : Bool$ specifying the notion of equality to be used to enforce duplicate elimination.

In comparison with sets, relations introduce the additional feature of a unique key. The key is used as basis for the duplicate elimination and for the definition of access operations. In order to enforce the uniqueness of keys, two elements have to be considered as duplicates if their keys are equal.

If a relation is specified as context, the resulting relation must not contain elements with equal keys. If the comprehension result contains elements with equal keys, there are two possible ways to handle this situation. The simplest way is to reject the query since the uniqueness constraint for the keys is not fulfilled. The alternative solution requires a case analysis. If e_1 and e_2 have the same key two cases can be distinguished:

e_1 and e_2 are equal: In this case only one of the elements is included in the result relation. The duplicate is ignored;

e_1 and e_2 are not equal: i.e., e_1 and e_2 are *conflicting* elements (compare to [Watt, Trinder 91]). In this case the query is rejected, because the query result of not well-defined.

The second alternative is chosen for the context *relation*. For this reason three functions are needed as parameters for the context: a function *key* to extract the key from a given element, a function *keyEqual* to compare two keys, and a function *equal* to check whether two elements are equal.

Aggregating Contexts

As mentioned in section 4.3.1, the different aggregate functions mainly vary in the choice of the function that is used to combine the elements of the comprehension result. Additionally there are differences in the effect of applying them to an empty comprehension result. The computation of the maximum element, for example, is not a legal operation in this case, whereas the sum of an empty collection is well-defined (equals zero). The aggregate functions that may be applied to empty comprehension results provide a bottom element that is returned as aggregate value in the empty case, and is also used to initialize the aggregate value computed by the function.

The aggregate functions *sum*, *max*, *min* and *count* are taken as example contexts. The context *sum* adds up the elements of the comprehension result. It may only be used for comprehensions with a numeric element type, or more precisely, an element type for which addition may be defined. Since TL supports no overloading, there is no unique operator "+" that is used for different numeric types as *Int* and *Real*. In order to allow the *sum* context to be applied for different numeric types, the add-function specific for the type (e.g., *int.add*, *real.add*) and the zero value of the type (e.g., 0 and 0.0) have to be passed as parameters to the context *sum*. The context is applicable to other types than *Int* and *Real* allowing the definition of a function *add* and a value *zero*.

The computation of the maximal or minimal element of the comprehension result needs an order that is specified by the elements. This order is expressable by a function *less* that takes two parameters and returns *true*; if, according to the order the first parameter is less than the second parameter. This function *less* is a parameter for the contexts *max* and *min* used for the computation of maximal and minimal elements, respectively. The following query example computes the oldest person that is younger than fifty years:

get *max(older)* { *p* | *p* ← *persons* : *p.age* < *50* }

where *older* is a function of type $\mathbf{Fun}(p1, p2 : Person) : Bool$ that compares the ages of persons *p1* and *p2*.

The context *count* does not depend on the element type of a comprehensions result. It computes the number of elements in the comprehension result.

The contexts representing aggregate functions may be classified by the properties (laws) that are fulfilled for the combining functions used. Commutativity and idempotence of these operators are considered in this discussion. The contexts *count* and *sum* are commutative, but not idempotent. The contexts *max* and *min* are idempotent and commutative. A context that is neither commutative nor idempotent would be a context *concat* that concatenates elements of type *String*. The above classification of contexts is of interest for the combination with contexts that restructure the comprehension result, namely, sorting and duplicate elimination. The application of an aggregate function that is not idempotent may yield a different result, if a duplicate elimination is enforced on the comprehension result before the aggregate function is applied. Duplicate elimination may not affect the query result if the aggregate function is idempotent. There exists a similar relationship between commutativity of the aggregate function and sorting. Sorting the elements of the comprehension result before applying the aggregate function may only change the query result, if the aggregate function is not commutative. Theses considerations are of interest for the use of combinative contexts together with aggregate functions.

Quantifiers

Existential and universal quantification over the elements of a comprehension are included in the representative set of contexts (*some* and *all*). Both existential and universal quantification depend on a predicate. It may be defined as part of the context or as part of the comprehension. In the first case, the predicate is a parameter of the context, as is illustrated by the following query example checking whether there is a person older than 100 years in the set of *persons*:

$$\textbf{get}\ \textit{some(older100)}\ \{\ p \leftarrow \textit{persons}\ \}$$

where *older100* is a function of type **Fun**(: *Person*) : *Bool* that tests if a person's age is greater than one hundred years.

In the second case, when the predicate is part of the comprehension, the predicate is represented by the target expression of the comprehension. Employing this approach the above query may be formulated as follows:

$$\textbf{get}\ \textit{some}\ \{\ p.\textit{age} > 100 \mid p \leftarrow \textit{persons}\ \}$$

The advantage of the second approach is the possibility to formulate the predicate by employing the range variables introduced in the comprehension. The definition of a function representing the predicate is avoided. For this reason, the quantifiers are included in this parameterless version into the representative set of contexts.

Selective Contexts

In section 4.3.1 two variations of this class of contexts are described. A representative for each variation is contained in the set of contexts considered here. The context *the* requires that the comprehension result consists of exactly one element and returns this element (variation b), whereas the context *any* chooses one of the elements contained in the comprehension result (variation a). Both contexts cause an exception, if the comprehension result is empty. The context *the* also raises an exception, if the comprehension result includes more than one element.

Combinative Contexts

In the scenario considered so far, every query consists of a comprehension and a single context. In contrast to this, some contexts presented in section 4.3.1 are combinable with other contexts. Examples are sorting of the comprehension result and the elimination of duplicates.

Duplicate elimination may, for example, be combined with the context *sum*, resulting in a summation that does not take duplicates into account when accumulating the sum.

There are two approaches to integrate such contexts into the given framework: other contexts can have these combinative contexts as parameters and yield combined contexts. Since the combination with contexts is optional, and since a context is combinable with several combinative contexts, a context must accept an arbitrary number of contexts as parameters. This leads to difficulties in the implementation. It is not possible to define a function that accepts arbitrary numbers of parameters.

Another approach is, therefore, applied. The combinative contexts are defined as a special class of contexts called *combinators*. Structurally, combinators are mappings from iterators to iterators. Since comprehension results are iterators, combinators as all other contexts can be applied to comprehensions. The result of this application is again an iterator and, therefore, further contexts are applicable. By this definition, queries can have more than one context, where the second and all further contexts have to be combinators. This approach requires an extension of the grammar of the query language. The alternative describing the result-oriented context has to allow more than one TL value for the specification of the context.

The representative set of contexts considered here includes the following combinators: *sortBy*, *uniqueOn*, *fromTo* and *flatten*. It is possible to distinguish different kinds of combinators. For some kinds of combinators a complete materialization of the comprehension result is necessary. An example for this kind is the combinator *sortBy* which computes a sorted iterator from a given iterator (comprehension result). Sorting of the elements of the comprehension requires the complete materialization. The elements may be inserted into a structure that facilitates sorting, as for example a search tree. A function *before* specifying the intended order of the elements has to be passed as parameter.

For other combinators, as for example *uniqueOn* (duplicate elimination), it is sufficient to keep track of the set of elements that have already been visited during the iteration. This set allows the detection of duplicates. The elements have to be stored in a structure that allows efficient member tests. The combinator *uniqueOn* expects a function *equal* as parameter that defines the notion of equality underlying the duplicate elimination (compare with the context *set*).

The combinator *fromTo* allows a position-dependent selection of a part of the comprehension result. Its two parameters *lowerLimit* and *upperLimit* specify the starting and ending position of the selection. The following query illustrates the application of the combinators *fromTo* and *sortBy*. It returns the list of the ten largest cities:

$$\textbf{get}\ \textit{fromTo (1,10) sortBy(size)}\ \{\ c \mid c \leftarrow \textit{cities}\ \}$$

where *size* is a function of type **Fun**$(c_1, c_2$ *:City) :Bool* that returns *true* if the city c_1 has more habitants than the city c_2. The combinator *flatten* can only be applied on comprehensions whose result elements are bulks. It flattens the structure. If it is, for example, applied on a comprehension result whose elements are lists the combinator returns an iterator over all elements in all lists[5]. The combinator *flatten* can be used to yield the bag of all children of all families living in cities with more than 50.000 habitants:

[5]This combinator is especially useful in the case of complex object databases.

get *bag flatten* { *f.children* | $f \leftarrow$ *families* : *f.home.habitants* > *50000* }

The combinators *fromTo* and *flatten* need no materialization of the comprehension result for the computation of the resulting iterator.

Evaluation on Demand

As mentioned above contexts enabling an evaluation of the comprehension on demand are necessary for querying very large and potentially infinite inputs. The representative set of contexts includes two contexts for this kind, namely *iter* and *range*.

The context *iter* allows a lazy evaluation of the comprehension. The computation of each element of the comprehension result has to be explicitly requested by the user. For this purpose the context *iter* returns an iterator that allows to iterate element-by-element over the comprehension result[6]. A comprehension enclosed in a bulk type context can be used as range to another comprehension, forming subqueries. A bulk type context requires complete evaluation in order to compute the query result[7]. In some situations lazy evaluation of subqueries is desirable, for example, if the context *any* is chosen for the outer query. In this case a complete evaluation of the comprehension is unnecessary and inefficient. The context *range* enables a lazy evaluation of comprehensions used as ranges for other comprehensions. It is very similar to the context *iter*, but additionally it fulfils the requirements, specified for bulk types used as ranges. A comprehension enclosed in the context *range*, therefore, is a valid range for a query.

4.3.4 Typing Aspects of Contexts

In section 4.2 typing aspects of comprehensions are considered. It is examined in which way the element type E_r of the comprehension result depends on the building blocks of the comprehension. The embedding of a comprehension into a specific context leads to several new typing aspects.

Firstly, there are contexts that may not be used for arbitrary comprehensions. They impose certain restrictions on the element type of the enclosed comprehension. For the context *flatten*, for example, the elements of the comprehension result have to be bulks[8].

Secondly, the result type of the complete query is considered. It is determined by the chosen context, and in many cases it also depends on the element type E_r of the comprehension.

Finally, if the contexts have one or more parameters, the typing of these parameters is of interest. Generally, the types of these parameters depend on the type E_r. In some cases the choice of the parameters have influence on the type of the query result.

Table 4.9 lists the typing aspects for the representative set of contexts presented in the previous section. It consists of five columns. The first column names the different contexts. The second and third column list the parameters of the contexts and their types. The fourth column

[6] The concept of an iterator is discussed in more detail in section 4.4. Possible implementations are presented in chapter 5.

[7] All elements of the comprehension result have to be inserted into the bulk.

[8] This is described by the resriction $<: Bulk(E)$ in table 4.9.

Context	Parameters	Parameter Type	Result Type	Restriction on E_r
		Aggregating Contexts		
sum	zero add	$:E_r$ **Fun**$(:E_r\ :E_r)\ :E_r$	E_r	<:**Ok** (Numeric)
max	less	**Fun**$(:E_r\ :E_r)$:Bool	E_r	<:**Ok**
min	less	**Fun**$(:E_r\ :E_r)$:Bool	E_r	<:**Ok**
count	–	–	Int	<:**Ok**
		Quantifiers		
some	–	–	Bool	<: Bool
all	–	–	Bool	<: Bool
		Bulk Types		
bag	–	–	Bag(E_r)	<:**Ok**
list	–	–	List(E_r)	<:**Ok**
set	equal	**Fun**$(:E_r\ :E_r)$:Bool	Set(E_r)	<:**Ok**
relation	key keyEqual equal	**Fun**$(:E_r)$:K **Fun**(:K :K) :Bool **Fun**$(:E_r\ :E_r)$:Bool	Relation(E_r K)	<:**Ok**
		Selection of Single Elements		
any	–	–	E_r	<:**Ok**
the only	–	–	E_r	<:**Ok**
		Combinators		
flatten	–	–	Iterator(E)	<: Bulk(E)
uniqueOn	equal	**Fun**$(:E_r\ :E_r)$:Bool	Iterator(E_r)	<:**Ok**
sortBy	before	**Fun**$(:E_r\ :E_r)$:Bool	Iterator(E_r)	<:**Ok**
fromTo	lowerLimit upperLimit	:Int :Int	Iterator(E_r)	<:**Ok**
		Evaluation on Demand		
iter	–	–	Iterator(E_r)	<:**Ok**
range	–	–	Range(E_r)	<:**Ok**

Figure 4.9: Typing Aspects of the Contexts

specifies the type of the query result. The last column shows the restrictions on the element type E_r of the comprehension result. Note that the entry <:**Ok** imposes no restriction on E_r since **Ok** is the supertype of all non-parameterized types in Tycoon. In all columns E_r represents the element type of the result of the comprehension which is embedded into the considered context.

4.4 Iteration Abstraction and Bulk-Morphisms

The developed query language is comprehension-based. The semantics of list comprehensions and its extension to comprehensions for other bulk types, namely, for ringads is given in chapter 2. All these semantic definitions are restricted to a special case: all inputs and the result of the comprehension are of the same bulk type. In contrary to this, different bulk types can be used as ranges of a comprehension in the presented query language. It is even possible to define mixed queries, i.e., to specify different bulk types for the ranges of a single query. An example illustrating such a case is given below:

A :List(Int)
B :Set(String)

get *bag* { **tuple** x y **end** | $x \leftarrow A$, $y \leftarrow B$ }

The ranges A and B are of type *List(Int)* and *Set(String)*, respectively. The type of the query result is *Bag(***Tuple** *:Int :String* **end***)*. Thus, the result of the query is not compatible with one of the types of the ranges.

In order to enable a uniform implementation of the queries, independent of the bulk types, a uniform intermediate format for the bulk types is needed.

Beeri and Ta-Shma present a category of bulk types where the objects are bulk types and the arrows are bulk-morphisms [Beeri, Ta-Shma 94]. In this category sets are identified as the terminal objects, i.e., for all other bulk types a unique bulk-morphism to the set bulk type exists. The notion of a bulk-morphism is discussed in section 2.2.3. Intuitively, a bulk-morphism may be used to map an instance of one bulk type to a target instance of another bulk type. Since set is terminal, every bulk type may be uniquely mapped to a set.

This approach leads to a first attempt for the definition of the semantics of mixed queries. Arbitrary ranges of the query may be uniquely mapped to sets. The sets are used as intermediate representation during the evaluation of the query. The result of the query is a set constructed from these intermediate representations. Since mappings of the bulk types to the sets are unique, the result is also unique. The approach leads, therefore, to a well-defined semantics for mixed queries. Unfortunately, this is not the intended semantic. Duplicates and order information present in the ranges are lost by the mapping to sets. In dependence of the chosen context, some of this information may be necessary to construct the query result. For the general case, an intermediate representation is therefore needed that preserves duplicates and order information present in the ranges.

In chapter 2 rewrite rules defining the semantics of list comprehensions are described. These rules are taken as a starting point for the examination of effects of ordering aspects to the comprehension/query result. In the following example the rewrite rules are used to transform a comprehension containing two lists A and B as ranges with elements $a_1, \cdots, a_n$ and $b_1, \cdots, b_n$, respectively:

$$\textbf{get}\ \mathit{list}\ \{\ \textbf{tuple}\ x\ y\ \textbf{end}\ |\ \ x \leftarrow A,\ \ y \leftarrow B\ \}$$

$$\begin{array}{lll} \rightarrow & [(a_1\ b_1)\ (a_1\ b_2) \cdots (a_1\ b_n) \cdots & \\ & (a_2\ b_1) \cdots & (a_2\ b_n) \\ & \quad \cdots & \\ & (a_n\ b_1) \cdots & (a_n\ b_n)\ ] \end{array}$$

The result of the comprehension is determined by iteration over the lists given as ranges. The rules specify the following iteration order: all elements of the second range are visited for the first element of the first range before the second element of the first range is considered, and so on. This can easily be extended to arbitrary numbers of ranges. The result of the list comprehension depends on the iteration order of the elements in the lists as well as on the order of the ranges in the comprehension. Exchanging ranges in the above list comprehension yields a different result, since the elements are iterated in a different order.

The semantic of a comprehension strongly depends on the iteration order over the elements for the involved bulk types. For this reason iterators as presented in [Liskov et al. 77] seem to fit well as an intermediate representation. Iterators are a kind of data abstraction, accessing all objects in a given bulk sequentially without exposing irrelevant details. If iterators are employed as intermediate representation, mappings from bulk types to iterators are needed (see figure 4.10). For the specific bulk type *list*, it is quite obvious how to iterate over the elements in a given list. For other ordered bulk types the iteration over their elements can be uniquely determined by the order of the elements in the bulk type. This is not true for unordered bulk types. To generalize the presented semantic definition, the iteration order for other bulk types than lists has to be considered.

For unordered bulk types more than one iteration order is possible. A mapping from such a bulk type to an iterator is, therefore, not uniquely determined. It is, for example, possible to iterate in different ways over the elements of a bag. This is clear from theoretic results given in the theory of Beeri and Ross (see chapter 2). The mapping from a bulk type to an iterator can be regarded as a morphism. Sets and iterators do not constitute an isomorphism: iterators preserve order information and duplicate elements of a bulk, whereas sets do not. Only those objects which are isomorphic to sets can be a terminal object in the considered category. Iterators are, therefore, not a terminal object of the defined category in the theory of Beeri and Ta-Shma, i.e., there are bulk types having no unique mapping to an iterator. Next, the consequences of this ambiguities towards the semantics of the comprehensions are examined.

It is possible to distinguish the following cases for the bulk types occurring as ranges of a comprehension:

(i1) the comprehension contains only ordered bulks

(i2) the comprehension contains only unordered bulks

(i3) the comprehension contains at least one ordered and at least one unordered bulk

The cases (i2) and (i3) involve iteration orders on unordered bulk types, resulting in possibly different sequences of elements for the comprehension result. The result of a comprehension can also be viewed as an iterator that allows the iteration of the mentioned sequence of elements. Following this argumentation, comprehensions form a mapping from one or more iterators to a target iterator (see figure 4.10). For the cases (i2) and (i3) this mapping is not uniquely defined. In these settings there is no well-defined semantics for the result of a given comprehension. The resulting iterator of this intermediate mapping does not represent the result of the query. The query is completed by setting the comprehension into a context which determines the result of the query. A context can specify the result of a query as a bulk type (see section 4.3.3). With respect to the ordering property two cases can be distinguished:

(r1) the result collection is ordered and

(r2) the result collection is unordered.

The following table shows the dependencies between the ordering aspects of the ranges, the ordering aspects of the resulting bulk and the uniqueness of the query result:

	r1	**r2**	**a1**	**a2**
i1 (ordered)	unique result	unique result	unique result	unique result
i2 (unordered)	no unique result	unique result	unique result	no unique result
i3 (mixed)	no unique result	unique result	unique result	no unique result

In general the order of the elements in an ordered bulk type depends on the order on which the elements are inserted[9]. It is assumed that the elements of the comprehension result are inserted into the bulk type chosen as context in the order they are visited by the iteration. For ordered result collections (r1) the order of the elements, therefore, is determined by the order of the elements in the result of the comprehension. Thus the different comprehension results in the cases (i2) and (i3) lead to different query results. The query result is not well-defined in these cases. If an unordered bulk type is chosen as context all possible comprehension results in the cases (i2) and (i3) yield semantically equal bulk values, since the order of the insertion of the elements has influence on the result. If all ranges are ordered bulks the comprehension result, and as a consequence, the query result is well-defined. Similar considerations can be made if aggregate functions are chosen as context. Referring to the examinations on page 58, the aggregating function can be commutative (case a_1) or not (case a_2). For a commutative aggregating function the order in which the elements are aggregated does not make a difference to the result. It is comparable to a unordered bulk type where the order of insertion of elements into the bulk does not make any difference (is commutative). For unordered bulk types and commutative aggregating functions the same results concerning well-definedness hold (see table). Aggregating functions that are not commutative are sensitive towards ordering aspects. For this reason they are comparable to ordered bulks as result. Note that the order of the ranges has an impact on the result of the query. This is true for the case (i1, r1)

[9] An exception are, for example, sorted list where the order is independent of the insertion. It is determined by the sorting. These cases are not considered here.

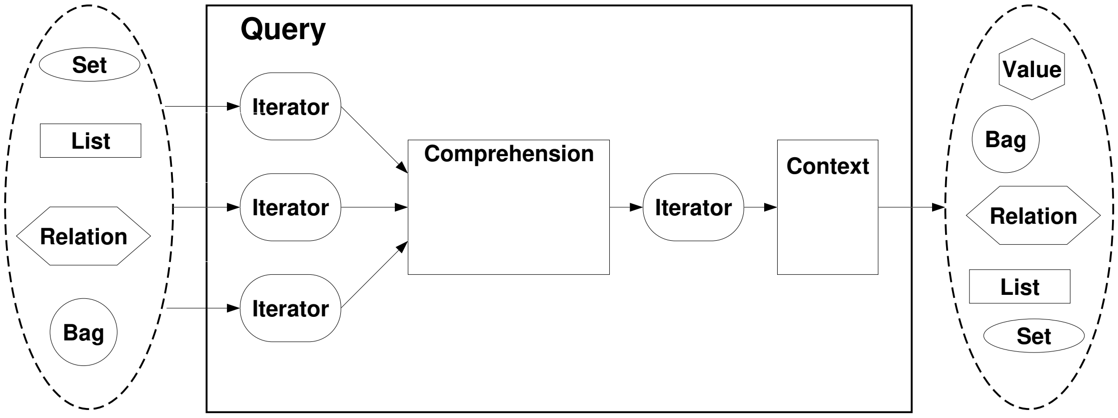

Figure 4.10: Mapping Semantics of the Query Language

for example. This aspect has been discussed for list comprehensions at the beginning of this section. Exchanging ranges can lead to different results.

A well-defined semantic for the queries (comprehensions) can be achieved by the following approach: while implementing a bulk type, a function that maps an instance of this bulk type to an iterator is defined. This function fixes the order of iteration over the elements of instances of this bulk type and as a consequence of this a unique comprehension result. If these functions are applied on the ranges of a comprehension used to determine the comprehension result a unique, well-defined result can be achieved for each case. Note that for unordered bulk types more than one definition of the mapping function exists. If more than one possible comprehension result exists for a comprehension, it is also possible to chose one of them by specifying an order predicate fixing the order of the elements in the result of the query.

4.5 Naming, Parameterization, and Recursion

Naming and parameterization of queries are important features for an enhanced query language. With the chosen approach for the realization of the query language, these features can be smoothly and naturally integrated into the presented framework for a query language. This is also true for recursive queries. The integration of parameterization, naming, and recursion into the query language is described in this section.

4.5.1 Naming of Queries

It is desirable that a query, like all other constructs of the language, can be bound to a name. There are two distinct ways for binding an identifier to a query. A name may be bound to

the result of the query: subsequent use of the name refers to the value calculated as query result. This way of binding allows the further processing of query results.

the query as an executable unit: each subsequent use of the name causes an execution of the query in the context of the actual state of the database. This kind of naming is useful for the definition of frequently used queries. The query is defined once and bound to a name. Reevaluation of the query may then be caused by employing the bound name. Another important application of this kind of naming is the definition of views.

Since queries are values of the language TL, no special constructs for the binding of identifiers to queries is needed. The binding mechanism defined for values in TL is applicable; the binding of a name to a query result may be accomplished by the use of the *let*-construct:

$$\textbf{let } \textit{bigCities} = \\ \quad \textbf{get } \textit{list} \ \{ \ c.name \mid c \leftarrow cities : c.habitants \geq 500000 \ \}$$

The identifier *bigCities* is bound to the list of all cities with more than 500000 habitants. In order to define a query as an executable unit, it is enclosed into a function closure before binding a name to it:

$$\textbf{let } \textit{bigCities} = \textbf{fun}() \\ \quad \textbf{get } \textit{list} \ \{ \ c.name \mid c \leftarrow cities : c.habitants \geq 500000 \ \}$$

An evaluation of the query is initiated by a call *bigCities()*.

4.5.2 Parameterization of Queries

A natural extension of naming a query as an executable unit is to allow parameterized queries. Parameterization means the introduction of one or more parameters into a query. Actual values for this parameters have to be passed at execution time of the query. As with all other parameterizations this approach allows to express a variety of similar queries by a single query. The differences are captured by the parameters.

Parameterized queries may easily fit into the presented framework. Queries as executable units are represented by parameterless functions. An obvious extension of this approach is to represent parameterized queries by functions with parameters:

$$\textbf{let } \textit{bigCities(lowerLimit :Int)} = \\ \quad \textbf{get } \textit{list} \ \{ \ c.name \mid c \leftarrow cities : c.habitants \geq lowerLimit \ \}$$

The parameters of the query are unbound identifiers inside the comprehension.

Since filters specified in the comprehension are values of the language TL (see section 4.1.2) a query can be parameterized with a filter predicate. This is illustrated by the following example:

$$\textbf{let } \textit{selectedPersons(predicate :}\textbf{Fun}\textit{(:Person) :Bool)} = \\ \quad \textbf{get } \textit{list} \ \{ \ p.name \mid p \leftarrow persons : predicate(p) \ \}$$

In this case the query is a higher-order function. It is also possible to pass an entire query as parameter. The query passed as parameter can be employed as subquery in any building block of the query (compare to section 4.1.1).

Copy Semantic vs. Referential Semantic

Mutable values which are referenced during the processing of the query have an identity, and, therefore, they can be referenced and altered from other locations. The consequence is a possible change of the query result during processing, since such an element could possibly change the result of a predicate referencing it. This can be omitted with deep copy semantics, but the costs for such an operation can be very expensive.

4.5.3 Recursive Queries

Naming and parameterization of queries are not supported by constructs of the proposed query language. As described in the previous section the binding facilities and the facilities for the definition of functions of the language TL are used for this purpose.

A straightforward extension of this approach is to realize recursive queries by recursive functions. A similar approach is presented in [Smedt et al. 93]. Following this idea the query to calculate the cost of a part p in a bill-of-material database may look as follows:

let rec *cost(p :Part) :Int =*
    p.cost + **get** *sum* { *sp.qty * cost(sp)* | *sp* ← *p.subparts* }

On the first sight the identification of recursive functions with recursive queries conceptually fits well into the presented framework. But a closer look shows some grave problems. A consequence of this approach is that recursive queries are evaluated with the same strategy used for the evaluation of recursive functions. This strategy lacks some features that are desirable or even necessary for the evaluation of recursive queries.

1. Query evaluation may run into cycles that are caused by cyclic data. These cycles have to be detected in order to avoid non-terminating evaluations.
2. It is crucial for the efficiency of query evaluation that intermediate results that are needed several times, are computed only once.

It is, therefore, desirable that the algorithm for the evaluation of recursive queries is not indivisibly coupled with the evaluation strategy for recursive functions.

Recursive queries in the considered representation are very similar to functions. They have a name and a signature. The following description is restricted to queries with one parameter in order to facilitate illustration. It is easily extended to queries with more than one parameter.

As mentioned above the detection of cycles and the memorization of intermediate results are important aspects of the evaluation of recursive queries. A structure is required that allows the detection of cycles and the storage and retrieval of intermediate results. This can be accomplished by a kind of dictionary having calls of the query as entries and parameters of the call as keys. For each parameter three states can be distinguished:

new: a call with this parameter did not yet occur during evaluation, no entry exists for this parameter;

ready: a call for this parameter has been completly evaluated; the structure contains the according intermediate result;

evaluating: a call for this parameter is actually in progress, a cycle has occured during evaluation.

The structure has to support the following operations: It has to be possible to request the state for a parameter *p* and to insert and retrieve intermediate results in dependence of a parameter *p*. Additionally, the structure has to keep track of the state of evaluation for a parameter *p*. For this reason functions to signal the start and termination of a evaluation towards the structure are desireable.

A recursive query *rq* with signature *rq(p :T):R* is transformed into a recursive function with the same signature preceded by the creation of a structure as described above. The body of the function mainly consists of the query expression *qe* of *rq* augmented by the following functionalities: for each parameter *p* the state is requested. The further processing depends on the state. In case of state *new* the query expression is evaluated for the parameter. If the case is *ready*, the already computed result for this parameter is retrieved and returned as result of the call. The state *evaluating* signals the occurence of a cycle. Actions taken upon the occurence of such a cycle depend on the query and the chosen evaluation strategy. If the evaluation of a call is completed the result has to be entered into the structure.

In order to distinguish them from recursive functions a special keyword for the definition of recursive queries is introduced. There are different approaches for the introduction of such a keyword. It is possible to introduce a new keyword **recq** that is used in the same position as the keyword **rec** for the definition of recursive bindings. The following example illustrates the use of the keyword **recq** in a recursive query.

> **let recq** *cost(p :Part) :Int =*
> *p.cost +* **get** *sum { sp.qty * cost(sp) | sp ← p.subparts }*

For the introduction of this facility the grammar rule describing value binding of TL has to be extended.

An alternative approach is to introduce a keyword that directly precedes the query expression and introduces the recursively used name. With this approach the above query looks as follows:

> **recQuery** *cost(p :Part) :Int*
> *p.cost +* **get** *sum { sp.qty * cost(sp) | sp ← p.subparts }*

For this approach it is sufficient to extend the rules for values in TL, as it has been done for the introduction of queries as values in the grammar in section 4.1.2. Note, that the name *cost* is introduced only for internal use in the query in this approach. It may not be used outside of the recursive query.

For the formulation of many recursive queries a union operator is needed. As an example the query calculating all subparts of the part *p* is considered. The set of these parts is the union of the direct subparts *sp* of *p* with the set of the subparts of these parts *sp*. A similar pattern may be identified in most queries calculating transitive closures. For the languages SQL2 and

SQL3 [Melton 93] an operator *recursive union* is proposed for this purpose. In [Smedt et al. 93] an operator *or* in the predicate is used to express the union. If comprehensions are used as query language there is also a need for this kind of operator. It is employed to express a union over ranges. In [Trinder 89] a *cons*-operation for lists is proposed for this purpose. Whereas [Beeri 92a] introduces an operator "∪". A similar approach is proposed in the presented query language: An operator "<+>" is introduced into the query language. Employing <+> as union-operator the subpart query may look as follows:

let recq *subparts(p :Part) : Set(Part) =*
  get *set* { *ssp* | *sp* ← *p.subparts, ssp* ← *subparts(sp)* <+> *sp* }

There are two different ways to support the union-operator. First, it may be introduced as additional syntactical construct into the query language. In this case the implementation of the operator is part of the syntax extension. On the other hand the operator may also be supported by the environment. A generic function allowing the union of arbitrary bulk types with an element can be defined. The function does not insert the specified element into the bulk type. It causes iteration over the bulk type and the specified element during query evaluation. If a symbolic name as <+> is chosen for the function it is even possible to use the operator in the intended infix notation (compare section 3.1.1). The approach to define the union operator as a function in the environment is more flexible since its implementation is adaptable. Additionally it avoids the introduction of new keywords into the language. This approach, therefore, is chosen here.

The considerations are restricted to recursive queries that may be expressed by linear recursive functions [Bancilhon, Ramakrishnan 86, p.20] [Smedt et al. 93, p.145]. This class of queries is powerful enough to express classical application problems such as computing ancestors, bill-of-materials, and shortest paths. It covers a broad range of practical applications involving recursion.

In the following a set of classical applications for recursive queries and their realization in the proposed query language is given [Ceri et al. 90]. First, a computation of a transitive closure having non-cyclic data is considered. The query returning all ancestors of a given person may be formulated as follows:

let recq *ancestor(a :Person) :Set(:Person) =*
  get *set* { *anc* | *p* ← *persons, c* ← *p.children : c == a, anc* ← *ancestor(p)* <+> *p* }

The next query computes an aggregate with a recursive query. The cost of a given part is computed from the cost of its subparts [Atkinson, Buneman 87].

let recq *costPart(p :Part) :Int =*
  p.cost + **get** *sum* { *sp.qty * costPart(sp)* | *sp* ← *p.subparts* }

The last class are queries computing the transitive closure over data structures having cycles. Assuming a set of connections between cities with the following structure

Tuple
from :City
to :City
end

a query computing all cities reachable from a given city may look as follows:

let recq *reachable (s :City) :List(City) =*
get *list* { *dest* | *c* ← *connections : c.from == s, dest* ← *reachable(c.to)* <+> *c.to* }

Note that *connections* may describe cyclic data.

4.6 Expressive Power

The relational calculus is a query formalism which underlies several database systems [Codd 72]. A query language is said to be relational complete, i.e., adequately expressive, if it is at least as powerful as the calculus. Any relational calculus query can be translated into an equivalent list comprehension [Trinder 89, pp.124–131]. Since the proposed query language also incorporates arbitrary functions and the naming and parameterization allows the formulation of recursive queries one can expect an even greater power than in the relational calculus.

The approach allows one to define a variety of models with rich data structuring facilities, including, in particular, all those that use constructors, or that allow abstract data types. Abstract data types are used to define add-on user definable bulk types and further they are used to define operations as generalized aggregates, restructuring operations, quantifiers, iterations, and generalized selection operations. The proposed query language utilizes comprehensions as its kernel construct and allows the lazy evaluation of queries [Vuillemin 74; Wadsworth 71], thus interpreted functions within a query expression are permitted. The user has the ability to evaluate potentially infinite query results to be materialized in an on-demand fashion. By this approach safety [Beeri, Milo 92; Abiteboul, Beeri 88] of a query can not be guaranteed, but the gain in flexibility and expressiveness is very large. Complete materialization of queries is also possible, but has to be handled with care since present functions may cause the query to compute the complete database. "Queries over infinite domains and complex types can be hard and their result may not be recursive enumerable" [Hull, Su 90]. The queries are typed and in the system they show to be syntactic sugar for a static-typeable expression.

The underlying model for the proposed query language uses a general data model generalizing the nested relations/complex object models that support the construction of complex values using tuple, set, etc. and those that support abstract data type definitions.

In the following a set of examples are given which support the above argumentation. A set-collapse and a powerset query is expressible in the proposed language. Further, the nesting abilities of the language allow the expression of queries performing grouping on query results.

A set-collapse query takes a collection of sets and returns their union [Beeri 92a, p.9]. It is expressible using *pump* and *union* [Abiteboul, Beeri 88].

let *collapse (E <:***Ok** *sets :Set(Set(E))) :Set(E) =*
get *set* { *element* | *set* ← *sets*, *element* ← *set* }

The collection of sets in the above query is *sets*. The first generator takes each set of the collection of sets (implicitly flattening). The second generator takes all elements of each set and returns the single elements. All elements are inserted into the resulting set.

A powerset query takes a set collection and returns the powerset of the set.

let recq *power (E <:***Ok** *s,* s_1 *:Set(E)) :Set(Set(E))=*
get *set* { s_p | s_p ← **get** *set* { e_r ← s : $e \notin s_1$, e_r ← *power (s, union(e* s_1*))* } <+> s_1 }

The query is parameterized with two sets s and s_1 and returns the powerset of the set s. To compute the powerset for a concrete set b, it has to be called with the following parameters *power(b {})*. With the set b as first and the empty set as second parameter. The operator *union* is the set union. It can be expressed by the following query:

let *union(E <:***Ok** *e :E s :Set(E)) :Set(E) =*
get *set* { r | r ← s <+> e }

The filter condition $e \notin s_1$ is only included to eliminate unnecessary evaluation steps. Note that the recursive query is introduces with a **let recq** binding construct which allows the introduction of an appropriate temporary data structure for the detection of cycles during the evaluation of the recursive query.

The next illustrating query returns all families having more than 4 children grouped by their townships. And only those groups are taken into account which have at least 5000 such families. A seperate query *children_lover* that computes all families with more than four children in a city is defined for this purpose:

let *children_lover(c:City) =*
get *list* { *f* | *f* ← *families* : **get** *count* { *c* | *c* ← *f.children* } > 4 ∧ *f.home = c* }

The named query *children_lover* can be utilized to formulate the intended as follows:

get *list* {*l* |
l ← **get** *list* { **tuple let** *c = city* **let** *cl = children_lover(city)* **end** |
city ← *cities*
: **get** *count* { *cl* | *cl* ← *l.cl* } > *5000* }

Note that it is not necessary to define the subquery seperately. It could also be directly incorporated, but the resulting query would be harder to understand.

Chapter 5

The Operational Semantics of the Query Language

In the previous chapter, a framework for a comprehension-based query language is presented. The discussion is focused on the conceptual design and the syntax of the query language. Furthermore, typing and scoping aspects are considered and a first intuition of the intended semantics is given.

Instead of defining an interpreter for the proposed query language, it is the aim to design the query language in an add-on approach using a syntax extension tool. For this reason, the queries have to be implemented by constructs of the underlying language. This process of fixing an appropriate implementation defines an operational semantics for the introduced query language. TL, a typed polymorphic language based on a second-order lambda calculus is used as target language[1]. The chosen language supports procedural and functional programming, allowing a functional, as well as a procedural implementation of the query language. These two alternatives are examined separately; mixed forms are also possible.

In the presented framework a query consists of two parts: a context and a comprehension, forming the kernel of the query. In order to facilitate the realization of the approach by a syntax extension tool, the attempt is made to keep the implementation of the comprehensions and the implementation of the contexts independent of each other.

Before the implementations are considered, the architecture of the environment of the query language is described in the first section of this chapter. The second section is devoted to a functional approach for the implementation. The described solution is based on the translation scheme for comprehensions presented in section 2.2.2. Alternatively, the query language can also be implemented using imperative programming features. An imperative approach to the operational semantics of the query language is presented in section 5.3. The last part of the chapter treats aspects of a concrete implementation of the query language by syntax extension. Rules for the implementation of the query language employing the syntax extension tool described in chapter 3 are presented. The rules realize the implementation developed in section 5.3.

[1] Chapter 3 introduces the concepts of this language that are relevant for this work.

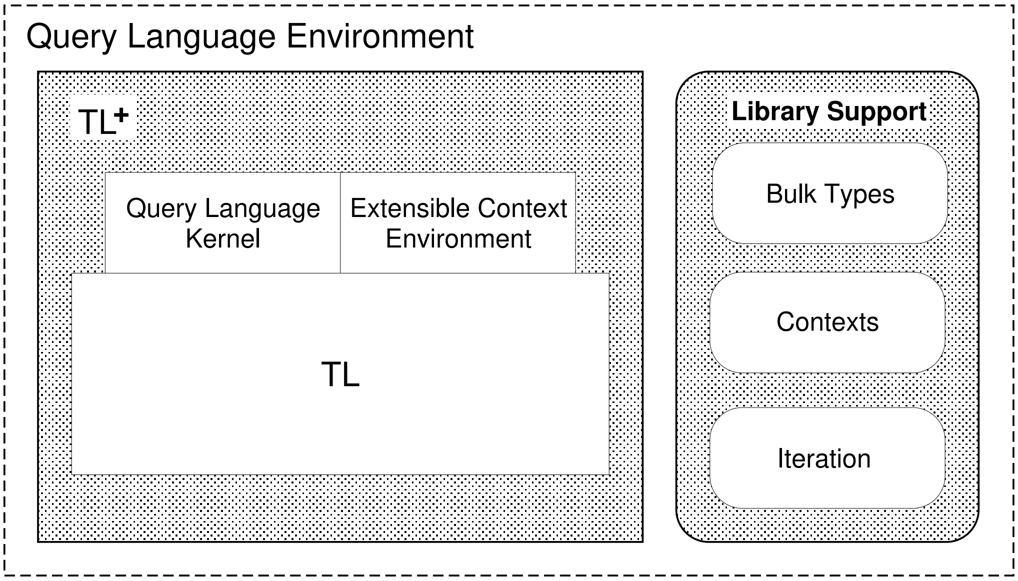

Figure 5.1: Architecture of an Environment for Query Languages

5.1 An Extensible Environment for Query Languages

Figure 5.1 shows the architecture of the query language environment. It consists of two parts: The language TL^+ and a set of supporting libraries. TL^+ is an extension of the language TL, used as host language for the presented query language. TL is extended by the syntax necessary for the declarative formulation of queries. The syntax extension consists of the syntax for the kernel of the query language, mainly a comprehension-based syntax (see section 4.1.1). Additionally, constructs for an extensible environment for contexts are included into the language TL (see section 4.3). The resulting language TL^+ accepts expressions of the language TL as well as expressions containing the constructs of the query language. Constructs containing the newly introduced syntax have to be implemented in the host language. The type checker of TL verifies the correct typing of the queries[2]. In order to realize TL^+, an implementation of the query language kernel and of the context environment is needed. A functional and an imperative implementation for these two components of the language are presented in the following sections.

The environment of contexts included in TL^+ is a framework for the implementation of contexts. Concrete contexts are supported by a library that may be adapted and extended in order to meet the requirements of special applications. In section 5.3.4 an example library of contexts is presented.

It is the aim of this chapter to present a uniform framework for querying different bulk types. The bulk types are not built-in in TL. They have to be supported by a library. The environment of the query language contains a library of bulk types. Similar libraries are provided in Modula-3 [Nelson 91; Mehlhorn, Näher 92] (reader, writer, and tables), Eiffel

[2]Typing aspects of the queries are discussed in section 4.2.

[Meyer 90] and Smalltalk [Goldberg, Robson 83] (collections), and in functional languages as ML [Paulson 91] and CAML [Mauny 91] (lists).

For the functional approach, a functional implementation of bulk types is employed based on ringads. The support of bulk types for the imperative approach represents bulk types as objects with a mutable state. The bulk types are provided in a generic form, as type operators, allowing arbitrary element types and key types (for relations).

The bulk types employed as input for the presented query language have to meet certain requirements. Especially, each bulk type has to provide a facility to iterate over its elements (compare to section 4.4). For the implementation of the query language a uniform format for iteration abstraction is needed. This format is described by a type operator in a separate interface (Iteration). It is used to iterate over the elements of the ranges. In the presented implementations each bulk type supplies a function that maps a collection to the required uniform format for iteration abstraction.

The environment for the query language is extensible: a library of bulk types may be extended and adapted to new requirements, e.g., for the introduction of new data models (object-relationship model [Albano et al. 91]). The query language may also be extended by extending the library of contexts.

5.2 A Functional Approach

Wadler introduces a translation scheme for list comprehensions [Wadler 87, p.132]. This scheme is described in chapter 2. It is taken as starting point for a functional implementation of the comprehensions. For the translation Wadler uses an enriched lambda calculus as target language. This facilitates the adaptation of the solution since the used target language TL is based on a lambda calculus as well. The translation scheme is based on a higher-order polymorphic function *flatMap*. This function takes a list and a list-valued function *f* as argument[3]. *flatMap* applies *f* to each element of the list and appends the lists resulting from the successive applications of *f*. An implementation of the function *flatMap* in TL may look as follows:

```
let rec flatMap (E1, E2 <:Ok   f(e :E1) :list.T(E2)   l :list.T(E1)) :list.T(E2) =
        if list.empty(l) then
           list.new(:E2)
        else
            list.append(f(list.head(l)) flatMap(:E1 :E2   f   list.tail(l)))
        end
```

This implementation reveals the dependencies between the signature of the function *f* and the element type of the input list and the resulting list. These dependencies are implicitly included in the implementation presented by Wadler.

The solution in [Wadler 87] is restricted to lists. In [Wadler 90; Trinder 92] it is shown that the translation scheme may be extended to arbitrary ringads (see also chapter 2). For the

[3] The application of the function *f* to an element returns a list as result.

considered query language this approach has to be adapted to allow the implementation of mixed queries (see section 4.4) and of comprehensions in different contexts (see section 4.3).

A direct adoption of Wadler's approach would require the definition of a function *flatMap* for each bulk type (ringad). Alternatively, the definition of a generic function *flatMap* working on arbitrary ringads is proposed. By further generalizing this function *flatMap* to a generic function *reduce*, a uniform implementation of the comprehensions in different contexts is possible. For the implementation of the different contexts it is sufficient to parameterize the function *reduce* with appropriate functions. An even more general approach is obtained by defining a generator for the reduce functions which are necessary for the different contexts. The generic reduce function and the generator are presented in section 5.2.2.

The implementation of queries employing a reduce function is divided into two parts. In section 5.2.3 a context independent implementation of the comprehensions is presented, whereas in section 5.2.4 the parameterizations of the reduce generator necessary for the realization of different contexts is described.

As discussed in section 5.1, the implementation of the queries imposes requirements on the environment in which the query is formulated and evaluated. Especially, certain assumptions about the bulk types defined in the environment have to be made to allow a uniform implementation of the comprehensions on arbitrary bulk types. Since the implementation relies on the environment and the definition of bulk types, the discussion of those precede the discussion of aspects of the rest of the implementation.

5.2.1 The Environment: Iteration and Bulk Types

In order to admit a uniform treatment of queries over different bulk types, a unifying framework for bulk types is needed. This topic has been investigated elsewhere [Trinder 92] and leads to the ringad theory of bulk types. Although the number of bulk types satisfying the ringad laws is restricted, most bulk types intuitively classified as such are ringads.

In section 5.1 a facility for iteration abstraction [Liskov et al. 77] is identified as further rudimentary building block of the environment for the query language. A functional realization of iterators is presented here.

A Functional Approach to Iterators

When aiming to support a uniform protocol for a sequential iteration over the elements of an arbitrary homogenous bulk structure, one can define the following recursively defined type constructor [Matthes, Schmidt 92]:

```
Let Rec Iter(E <:Ok) <:Ok =
  Tuple
    empty() :Bool
    get() :E
    rest() :Iter(E)
  end
```

It is parameterized by the element type E of the iteration. The components *empty*, *get*, and *rest* of the type constructor *Iter* are parameterless functions. The function *empty* yields the Boolean value *true*, if the iteration contains no elements to iterate over. Application of the function *get* returns the first value of the iteration and *rest* yields an iterator without the first element. This iterator is again of type *Iter(E)*. Both *get* and *rest* are only defined for non-empty iterators[4]. An application of the function *get* leaves the iterator unchanged. The second element of an iterator *it* is obtained by the following expression: *it.rest().get()*.

Iterations over sets, bags, and lists and other bulk types can be represented using this recursive representation of an enumerative data structure. For each bulk type a function *elements* has to be defined that maps an instance of this bulk type with element type E to an iterator of type *Iter(E)*. In [Matthes, Schmidt 92] the implementation of such a function *elements* for arrays is presented as an example. In a similar way this function can be defined also for other bulk types, as sets, bags, and lists and for more complex types as trees. As discussed in section 4.4, one might not find a unique solution in each case: binary trees and lists for example provide a criterion for their iteration order, whereas sets and bags do not specify any order at all.

An Implementation of Ringads

A monad may be defined by a constructor M and two functions *flatMap* and *unit*. The expressive power of a quad, i.e., of a monad with a *zero* is sufficient to implement comprehensions (see chapter 2). Therefore comprehensions may be implemented for bulk types having the functions *flatMap*, *unit*, and *zero*. This approach is adopted here. In order to avoid the definition of an own function *flatMap* for each bulk type, it is attempted to define one generic function *flatMap* that may be used for the implementation of comprehensions over arbitrary bulk types. Aiming to explore in which way the function *flatMap* must be generalized to support other bulk types than lists, the structure of *flatMap* is examined (see page 75).

The function is defined by structural recursion [Burstall 69]. It consists of two branches, one for the empty list and one for lists consisting of at least one element. In case of an empty list as input the function *flatMap* returns an empty list. Regarding lists as a ringad, the returned list is the *zero* of this ringad. In case of a non-empty list the result of the application of the function f to the head of the list is appended to the result of a recursive call of *flatMap* applied to the rest of the list. Again taking the point of view of a ringad, the list append is the ringad operation *combine*.

Consequently, the bulk type has to be augmented with a function *combine* in order to allow an implementation of the generic function *flatMap*. This leads to an implementation of a ringad[5]. The functions *zero* and *combine* of the ringad are utilized to construct the result of the function *flatMap*. Every bulk type defined as ringad, therefore, may be specified as result type (context).

Additionally, a facility to split the bulk type into a first element and the rest of its elements has to be provided for each bulk type. This may be realized by providing a function *elements* for each bulk type mapping instances of this bulk type to iterators (compare to page 76). The operations of the iterator *get* and *rest* are applied to split the bulk type specified as

[4] Their application to empty iterators causes an exception.

[5] A ringad is a monad with a *zero* and a *combine* (see chapter 2).

input to the generic function *flatMap*. This solution allows arbitrary bulk types as input to a comprehension. It is even possible to specify different bulk types as input to a single query, thus, enabling the implementation of mixed queries.

A ringad can be realized in TL by an abstract data type of the following form:

```
Let Ringad (E <:Ok) =
  Tuple
    M <: Iterative(E)
    zero() :M
    unit(:E) :M
    combine (:M :M) :M
  end
```

where *Iterative* is defined as:

```
Let Iterative (E <:Ok) =
  Tuple
    elements() :Iter(E)
  end
```

M represents the constructor of the ringad. Syntactically M is a type variable that has to be defined in a concrete implementation. The specified subtype relationship guarantees that it is at least possible to iterate over the elements of a value of type *Ringad*. More precisely, a value r of type M can be mapped to an iterator by application of the function *r.elements*. The functions *zero*, *unit*, and *combine* are defined on the type M.

To enable the implementation of comprehensions, a bulk type has to be defined as ringad (cf. above). In the implementation this restriction is realized by a subtype relationship: each bulk type has to be represented by an abstract data type that is in subtype relationship to the operator *Ringad*. As an example, the abstract data type for a list is considered:

```
Let List (E <:Ok) <:Ringad(E) =
  Tuple
    M <:Iterative(E)
    zero() :M
    unit(:E) :M
    combine (:M :M) :M
      ...
  end
```

The subtype relationship ensures that at least the functions *zero*, *unit*, and *combine* are defined for the abstract data type. Further operators may be appended at the end of the tuple. A concrete implementation of the abstract data type *list* implements the functions specified in the tuple. The ringad laws must hold for this implementation. For many common bulk types as sets, lists, and bags implementations fulfilling the ringad laws are discussed in literature [Watt, Trinder 91]. For a ringad implementation of sets the *combine* function has to be idempotent (see [Watt, Trinder 91, p.5]), i.e., duplicates have to be eliminated. As discussed in section 4.3.3, an equality function has to be defined as basis for the duplicate elimination. Different notions of equality are possible. The proposed solution is to provide a Boolean function for a set that specifies the intended equality. There are two approaches to incorporate this parameter into a set ringad. An additional parameter can be defined for the functions *unit* and *zero* that are used to construct new sets. By this approach the uniformity of the signatures of the ringad operation is lost. The subtype relationship to the type *Ringad* does not hold. Additionally, the semantics of combining two sets with different definitions of equality is not clear. Therefore, a different approach is taken. The equality is specified as a component of the abstract data type. A concrete implementation of the abstract data type *set*, fixes the notion of equality. All values (sets) of type *set.M* are based on the same equality function. In this framework it is only possible to combine sets carrying the same notion of equality. It is possible to write a polymorphic, higher-order function *generateSet* that generates an implementation for the abstract data type *Set*. It is parameterized by the element type and the equality function:

*generateSet(E:***Ok** *equal(:E :E) :Bool) :Set(E)*

The implementation of relations with unique keys requires special treatment. This is discussed in the next part of the section.

Relations as Ringads

Watt and Trinder discuss the problem of representing relations with a unique key as ringads [Watt, Trinder 91]. More precisely, they examine a concept *finite mappings* that is very similar to relations with keys. It is shown that the function *combine*, denoted as $\oplus$ in this text, has to be defined as partial function in order to fulfil the ringad laws. For this reason, the notion of *conflicting* elements is introduced. Two elements are conflicting if they are not equal in their values, but have equal keys. The function *combine* has to fail, if an attempt is made to combine two collections containing conflicting elements ($[x] \oplus [x'] = \perp$ if $\text{same\_key}(x, x') \wedge x \neq x'$ [Watt, Trinder 91, p.16]). This specification of the *combine* function raises the question for equality of two elements and two keys of a relation.

Contrary to flat relations, the answer to this question is not obvious in presence of nested structures for key and element types. Again different notions of equality are applicable (see section 4.3.3). Similar to the set implementation, equality functions may be employed to solve the problem of defining key equality and value equality for a given relation. Typically relations provide associative access to elements employing the unique key concepts. With a function *lookup* for the associative access the abstract data type for the relations may look as follows:

Let *Relation (E, K <:***Ok***) =*
  Tuple
    M <:Iterative(E)
    zero() :M
    unit(:E) :M
    combine(:M :M) :M
    lookup(:M :K) :E

end

The abstract data type does not only depend on the element type of the relation, but also of the type K of the keys. As described in section 4.3.3, functions *key* and *keyEqual* are needed to support the concept of keys. Additionally, a function *equal* of type **Fun***(:E :E) :Bool* is needed for the recognition of conflicting elements. This leads to the following signature for a generator function associated to the abstract data type *Relation*:

*generateRelation(E, K <:***Ok**
key(:E) :K
keyEqual(:K :K) :Bool equal(:E :E) :Bool) :Relation(E K)

5.2.2 A Generic Function Reduce

The function *flatMap* can be generalized to arbitrary bulk types employing ringads and iteration abstraction (see previous section). This generic function is used to implement comprehensions with arbitrary bulk types (ringads) as input and in arbitrary bulk type contexts. A further generalization of the function *flatMap* to a generic function *reduce* is presented in this section. As a motivation the result of the application of Wadler's translation scheme to a query

get *list* { $x \mid x \leftarrow listA : odd(x)$ }

is considered, where $listA$ is the list [1, 2, 3, 4, 5] and *odd* is a Boolean function that checks if an integer value is odd. This query will be transformed into the following expression by the translation scheme:

[1] ++ ([ ] ++ ([3] ++ ([ ] ++ ([5] ++ [ ]))))

where ++ denotes the list append operation and [ ] denotes the empty list.

If the context *sum* is chosen the expected result of the transformation could be written as follows:

$1 + (0 + (3 + (0 + 5)))$

or even as

$id(1) + (0 + id(3) + (0 + id(5)))$

The same translation scheme would work for the context *sum* if we replace *combine* by *add* (+), [ ] *(zero)* by 0 and *unit* by *id*. If this idea is generalized the translation scheme is applicable for different contexts, if it is parameterized by the following context-dependent functions: a function *zero*, a function *unit* to be applied to each element which takes part in the result, and a function *combine* to unite two intermediate results.

If this idea is applied to the function *flatMap*, it leads to a generic function *reduce* parameterized by the functions *f*, *zero*, and *combine*:

```
let rec reduce(E, R <:Ok   it :Iter.T(E)   f(:E) :R   zero() :R   combine(:R :R) :R) :R =
  if it.empty() then
  zero()
  else
    combine(f(it.get())   reduce(it.rest()   f   zero   combine))
  end
```

Choosing *zero* as *list.zero* and *combine* as *list.combine* the application of the reduce function returns the same result as a *flatMap*. In contrast to the function *flatMap* the result type R of *reduce* does not have to be a bulk type. It may as well be a scalar value as in the case of *sum*.

If the translation scheme is applied to queries with more than one range, there is a call of *reduce* for each range[6]. Each call of *reduce* has to be parameterized with the result type R of the query, the element type E of the actually considered range, and the functions *f*, *combine*, and *zero*. Since the parameters R, *zero*, and *combine* depend only on the context of the query they are equal for each call of *reduce* within one query. In order to reduce the number of parameters a generator for reduce functions is defined:

```
let reduceGen (R<:Ok zero() :R combine(:R :R) :R)
            :Fun(E:Ok  it :Iter.T(E)  f(:E) :R) :R =
  begin
    let rec reduce(E:Ok it :Iter.T(E)  f(:E) :R) :R =
    if it.empty() then
      zero()
    else
      combine(f(it.get()) reduce(it.rest() f))
    end
      reduce
    end
```

The generator is used to generate the appropriate reduce function for a query. The parameters R, *combine*, and *zero* are chosen in dependence of the context. They are fixed for a concrete reduce function. The function *reduce* generated by the following parameterization may be used to implement a query when *list* is chosen as context:

```
let reduce = reduceGen(:E   list.zero   list.combine)
```

where *list* is an implementation of the abstract data type *List(E)*.

The next section describes how a comprehension can be implemented employing a reduce function generated in dependence of a context. The choice of appropriate parameters for the reduce generator for different contexts is discussed in section 5.2.4.

5.2.3 The Implementation of Comprehensions

In this section a context-independent implementation for comprehensions with arbitrary numbers of ranges is developed. Again as a starting point, a direct implementation of the trans-

[6] This is discussed in the next section (Compare also to Wadler's translation scheme.).

lation scheme proposed by Wadler is considered [Wadler 87].

The following query is used as example:

get *list* { *p.name* | *p* ← *persons* : *p.age* < *18* }

A possible implementation of this query according to Wadler's translation scheme is:

```
flatMap (:Person  :String
         fun(p :Person)
           if p.age < 18 then
             list.unit(p.name)
           else
             list.zero()
           end
         persons)
```

The function passed as parameter generates singleton lists from all those elements which satisfy the given filter. The function *flatMap* concatenates these generated lists. The given implementation seems to complicate things more than necessary. It would be easier to insert the elements into the result list instead of creating singleton lists and appending them. As discussed later, this solution is preferred to allow a uniform solution for queries with more than one range expression.

The reduce generator presented in the previous section is used to define a function *flatMap*. It is thereby necessary to define adequate parameters *combine* and *zero* for the generator have to be defined:

```
let zero = list.zero
let combine = list.combine
let reduce = reduceGen(zero  combine)  (* = flatMap *)
```

The implementation of a comprehension still needs an additional function *unit* to be applied to each element that takes part in the query result. In case of the list, this is a singleton operation, i.e., an operation constructing a one-element list out of an element passed as parameter:

```
let unit = list.unit
```

Employing the defined functions *zero*, *combine*, and *unit* and the generated function *reduce*, the implementation of the query has the following form:

```
reduce(persons.elements()
     fun(p :Person)
       if p.age < 18 then
         unit(p.name)
       else
         zero()
       end)
```

The resulting implementation of the comprehension is independent of the context. It may, for example, also be used to implement a comprehension in the context *sum* assuming the environment:

```
let zero() = 0
let unit(x :Int) = x
let combine(x, y :Int) = x + y
```

Adequate environments for other contexts are presented in section 5.2.4. A clear separation of the implementation of the comprehension and the context is possible. This allows a uniform realization of comprehensions in different contexts, which is a prerequisite for a solution by syntax extension:

```
< context > { < target > | < range variable > ← < range > : < filter > }
```

```
begin
  let zero = ···
  let unit = ···
  let combine = ···

  let reduce = reduceGen(zero combine)

  reduce(< range >.elements()
     fun( < range variable > :< element type of range >)
      if < filter > then
        unit( < target > )
      else
        zero()
      end)
end
```

The functions *zero*, *init*, and *combine* are defined in dependence of the context.

The element type of the range is needed for the implementation of the comprehension. It has to be specified as type of the parameter of the function passed to the function *reduce*. This type is not explicitly included in the query. This missing information leads to a problem, when the code is to be automatically generated by a syntax extension tool. There are two possible solutions: the type information can be explicitly included into the query expression by specifying a type for each range variable. This leads to extra work for the programmer and makes the syntax less compact. Therefore, it is assumed here, that the syntax extension tool supports a type inference mechanism [Damas, Milner 82], that allows to infer the missing type information.

Up to now, the discussion is restricted to the class of one-range queries. Next, an extension of the approach to queries with arbitrary numbers of ranges is examined. Again the considerations are based on the translation scheme of Wadler [Wadler 87] (see also chapter 2).

Building Block	f^i_{rest}
generator $rv_j \leftarrow r_j$	reduce(r_j.elements() **fun**(rv_j :E_j) f^{i+1}_{rest})

filter
f_j
	if f_j **then**
	f^{i+1}_{rest}
	else
	zero()
	end

target expression t	unit(t)

Figure 5.2: Case Analysis for Query Expressions

The implementation of the general case is developed by parsing the comprehension from left to right, starting with the first generator. The implementation evolves step-by-step. At each point the remaining part r of the comprehension is split into the leading part and the rest of the comprehension. The implementation of r consists of the implementation of this leading part, and the implementation f_{rest} of the rest of the comprehension.

Each generator is translated into a call of the reduce function. The function *reduce* takes two parameters: an iteration and a function to be applied to each element of the iteration. The iteration to be passed to the reduce function is the iteration over the elements of the range. The function to be passed depends on the rest of the comprehension:

fun *(r_v : E)* f^i_{rest}

where r_v is the range variable introduced by the generator and E is the type of this variable. f^i_{rest} depends on the rest of the query. Three cases are distinguishable: the considered part of the comprehension is the last one, or the following part is a further generator, or it is a filter. In case of a further generator, f^i_{rest} is a subsequent call of the function *reduce* with appropriate parameters as discussed above. If the next part is a filter f^i_{rest} is of the form:

if *f* **then** f_{rest} **else** *zero()* **end**

In the other case the target expression t has to be considered as next part for the implementation. The implementation of this part is an application of the function *unit* to the result of evaluating the target expression t. The call corresponding to the (i + 1)th generator is nested within the call of the i-th generator, or more precisely, it is passed as parameter to the call of the function for the i-th generator.

5.2.4 The Implementation of Different Contexts

Comprehensions may appear in different contexts as for example aggregates, bulks, etc. (see section 4.3). The previous section presented a comprehension independent functional implementation of comprehensions with arbitrary numbers of ranges. Together with the introduction of the reduce generator (in section 5.2.2) an approach is described to implement the context of calculating the summation over the query result and collecting the elements in a bulk type by defining three functions *zero*, *unit*, and *combine*. Employing the reduce generator, the functions *combine* and *zero* are used to generate an appropriate reduce function for the implementation of the query.

This approach may be extended to further aggregate functions and to other contexts. Table 5.3 shows the implementation of different contexts. In section 4.3.3 a context *iter* is described enabling lazy evaluation of a comprehension. In order to implement this context, adequate functions *zero* and *init* have to return an empty iterator and an iterator with a single element, respectively. They can be defined as follows:

```
let zero(E_r <:Ok)() =
  tuple
    let empty() = true
    let get() = raise error
    let rest() = raise error
  end

let unit(E_r <:Ok)(x :E_r) =
  tuple
    let empty() = false
    let get() = x
    let rest() = zero()
  end
```

The function *combine* has to append two iterators:

```
let combine (E_r <:Ok) (it1, it2 :Iter(E)) :Iter(E) =
    tuple
      let empty() = it1.empty() ∧ it2.empty()
      let get() = if it1.empty() then
                    it2.get()
                  else
                    it1.get()
                  end
      let rest() = if it1.empty() then
                     it2.rest()
                   else
                     combine(it1.rest() it2)
                   end
    end
```

The operators *zero*, *init*, and *combine* are polymorphic functions. In order to enable direct passing to the reduce generator presented in section 5.2.2, they are defined in a curried version (compare section 3.1.5).

Context	zero()	unit(x)	combine(x y)
Aggregates			
sum	zero	x	add(x y)

max	opt.nil(:E_r)	x	**if** opt.null(y) **then**
			x
			elsif opt.null(x) **then**
			orif less(opt.value(x) opt.value(y)) **then**
			y
			else
			x
			end

min	opt.nil(:E_r)	x	**if** opt.null(x) **then**
			y
			elsif opt.null(y)
			orif less(opt.value(x) opt.value(y)) **then**
			x
			else
			y
			end

count	0	1	x+y
Quantifiers			
some	false	x	x $\vee$ y
all	true	x	x $\wedge$ y
Bulks			
bag	bag.zero	bag.unit(x)	bag.combine(x y)
list	list.zero	list.unit(x)	list.combine(x y)
relation	relation.zero	relation.unit(x)	relation.combine(x y)
Single Elements			
any	opt.nil(:E_r)	opt.new(x)	**if** opt.nonNull(x) **then** x **else** y **end**

Figure 5.3: Implementation of the Contexts

According to the grammar of the query language presented in section 4.1.2 pure comprehensions are not defined as values of the language TL. It is, therefore, not necessary to represent them as TL values in the implementation. If the comprehensions are not represented by TL values, as it is the case in the solution considered so far, it is not possible to implement the combinators presented in section 4.3.3.

It is proposed in section 4.4 to represent comprehensions by iterators in order to yield a clear definition of the semantics. This approach can be realized here by utilizing the functions *zero*, *unit*, and *combine* are utilized as standard parameters for the reduce generator and the translation scheme independent from the specified context. The different contexts are then defined on the basis of this iterator. A similar approach is chosen for the imperative implementation presented in section 5.3.3. Since the implementation of the different contexts using similar in both cases, a description is omitted here. It is presented in section 5.3.4.

Representing the comprehensions by iterators has the advantage that certain contexts as, for example, *any* can be implemented more efficiently: only the part of the comprehension necessary for the query result has to be evaluated. Furthermore the combinators can be supported if the approach is chosen[7]. On the other hand, supporting comprehensions by a functional implementation of iterators requires a construction of three function closures (*empty*, *get*, and *rest*) per element of the constructed comprehension result. This leads to time and space overhead during the evaluation of queries, especially, if contexts requiring a complete evaluation of the comprehension are chosen.

To conclude the discussion of the imperative implementation, it can be stated that the operational semantics is based on a clear theoretical foundation. These foundations are smoothly extensible to support arbitrary bulk types and the implementation of mixed queries. Thus enabling a concise implementation of the kernel of the query language. Extending the implementation to other contexts than bulks leads to some problems. Some contexts as, for example, *any* are implemented inefficiently by the employed framework. Others, namely, the combinators can not be implemented at all.

Comprehensions are represented by iterators. All contexts presented in section 4.3.3 can be implemented. The necessity to generate three function closures per element of the comprehension result, however, leads to an overhead in space and time during evaluation.

In the next section an imperative implementation of the query language is developed. It overcomes most of the problems of the functional approach.

5.3 An Imperative Implementation

In the previous section a functional approach to the operational semantics of the query language is presented. Since the employed language TL support imperative as well as functional programming features, it is possible to develop an imperative approach in the same linguistic framework. To get a first intuition for a possible solution a flowchart of an algorithm implementing a one-range query in an bulk type context is considered in figure 5.4.

The algorithm consists of the following steps:

[7] A possible realization of combinators in such a setting is also discussed in section 5.3.4.

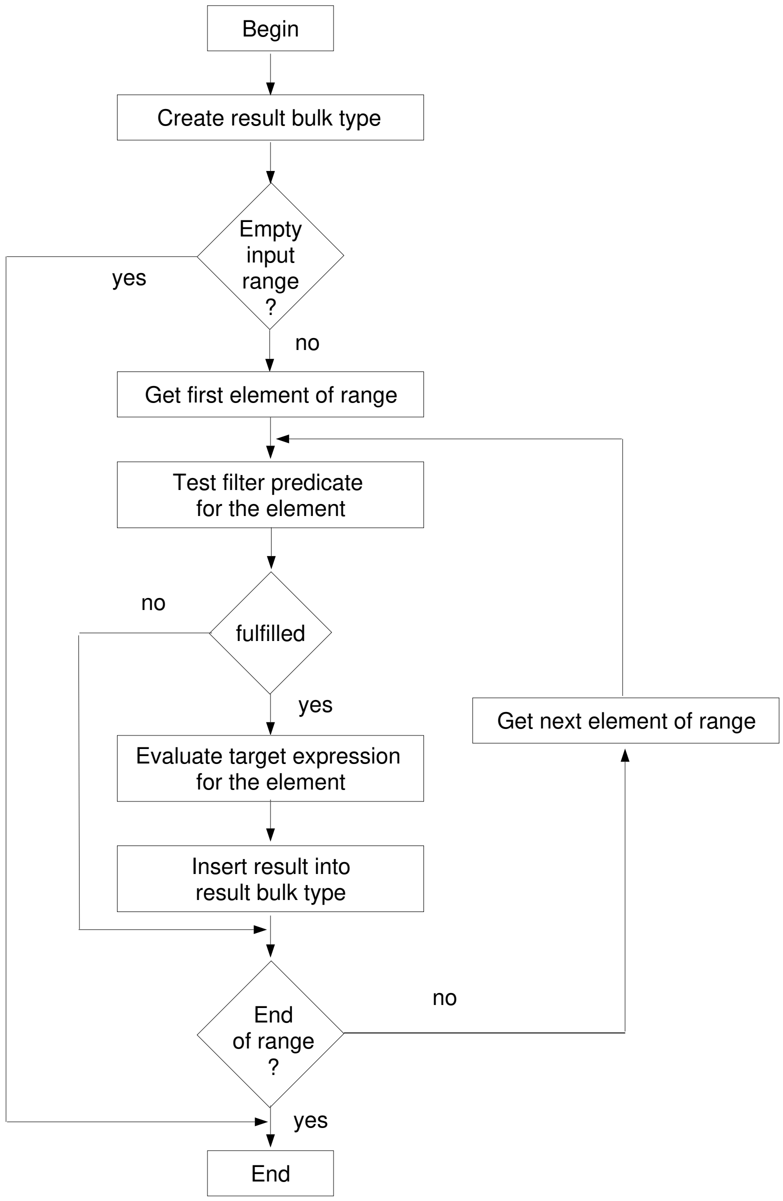

Figure 5.4: Algorithm for one-range Queries

$\boxed{\textit{create result bulk}}$	$\boxed{\textit{initialize aggregate value}}$
while *not eoi* **do**	**while** *not eoi* **do**
e = get next element of range	*e = get next element of range*
if *filter (e)* **then**	**if** *filter (e)* **then**
evaluate target expression	*evaluate target expression*
$\boxed{\textit{insert result into result bulk}}$	$\boxed{\textit{add result to the aggregate value}}$
end	**end**
end	**end**

Figure 5.5: Comparing two Pseudocodes

1. Create the result bulk.

2. Test input range for emptiness.

3. Get the first element of the range.

4. Evaluate the filter predicate for this element. If the predicate is fulfilled then

   (a) evaluate the target expression and

   (b) insert the result into the result bulk.

5. Get the next element of the range, if possible and return to step four. If no more elements are retrievable from the bulk, then stop.

The algorithm implements the comprehension in a specific context, namely, a bulk type context. The goal is to find a context-independent implementation of a given comprehension. For this reason the implementation in different contexts is compared in order to find common patterns, which are context-independent. Figure 5.5 puts the pseudocode for the implementation in the context of a bulk type in contrast to the implementation in the context of an aggregate function.

The implementation mainly consists of a *while*-loop performing the iteration over the input collection and terminating, if the end of iteration (*eoi*) is reached. Those differing parts, namely, the initialization preceding the loop and the function applied inside the filter, are shown in a framed box in the presented figure. A first version for a context-independent implementation of comprehensions is discussed in section 5.3.2. It is further developed in section 5.3.3. The realization of the different contexts is the topic of section 5.3.4.

The implementation of comprehensions assumes an environment supporting bulk types and a facility to iterate over the elements of a bulk type (as discussed in section 5.1).

Supporting libraries for bulk types and iteration over bulk types are discussed in section 5.3.1.

5.3.1 The Environment: Iteration and Bulk Types

As described in section 5.1, the environment has to include library support for bulk types and iteration abstraction. In the previous section (5.2) a functional implementation of these concepts is presented. In this section the necessary library support based on imperative programming concepts is presented.

Iteration Abstraction

In order to support queries over different bulk types in a uniform manner, a uniform format for the iteration over the ranges of queries is needed (see also section 5.2.1). Iterators are used for this purpose. In section 5.2 a functional realization of iterators is presented. For the imperative approach, iterators are implemented as objects with a mutable state. The state of the object keeps track of the progress of the iteration. The interface of the iterator objects are fixed by the following type operator:

Let *Iter(E* <:**Ok***)* <:**Ok** =
  Tuple
    eoi() :*Bool*
    get() :*E*
    next() :**Ok**
  end

An iterator over a bulk type with element type *E* is of type *Iter(E)*. The state of the iterator may be inspected by the functions *eoi* and *get* and changed by the function *next*. The function *next* updates the state of the object to point on the next element of the iteration. A call of the function *get* returns the actual element of the iteration. If the end of the iteration is reached, which can be checked using the function *eoi*, calls of the functions *get* and *next* cause an exception to be raised.

Bulk Type Support

The bulk types are also implemented as objects with a mutable state. The set of functions included in the interface depends on the specific bulk type. As mentioned in section 4.4, bulk types have to meet certain requirements if they are to be used as input for queries. In order to allow iteration over the elements of the bulk type, a function *elements* that maps the bulk type on an iterator of type *Iter* is required.

If a bulk type is to be used as context for a comprehension, i.e., if the elements of the comprehension result are collected in this kind of bulk, an *insert* operation on this bulk type is needed. The chosen implementation sets the different bulk types into a subtype relationship to a common supertype. This supertype is defined by the following type operator:

Let *QBulk(E*:**Ok***)*<:**Ok** =
  Tuple
    elements() :*Iter(E)* (* *iterator* *)

```
    insert(e :E) :Ok
  end
```

Different kinds of bulk types may be defined as subtypes of *QBulk*. This is shown for lists in the following example:

```
Let QList(E <:Ok) <:QBulk(E) =
  Tuple
    Repeat QBulk(E)
      ···  (* further bulk type operations *)
  end
```

The type of a list with element type E is a subtype of *QBulk(E)*. For every bulk type a function *new* has to be implemented which allows to create new collections of this kind. The function *new* has to be polymorphic providing means to create collections with an arbitrary element type. The desired element type is passed as parameter to the function. For the type *QList* the signature of the function *new* looks as follows:

```
new(E:Ok) :QList(E)
```

As discussed in section 5.2.1, additional parameters are needed for the creation of sets and relations. When a set is created, an equality function for the duplicate elimination has to be specified. The definition of a relation with unique keys requires a function *key* to extract a key from an element, and a function *keyEqual* to test whether two keys are equal. The functions *key* and *keyEqual* are used to ensure the uniqueness of keys inserting new elements and to support a function *lookup* that allows the access of an element of the relation given its key. The type of relations can be described by the following type operator:

```
Let Relation(E, K <:Ok) <:QBulk(E) =
  Tuple
    Repeat QBulk(E)
      delete(k :K) :Ok
      lookup(k :K) :E
      member(e :E) :Bool
          ...
      end
```

Note that the type operator describing the type *Relation* does not only depend on the element type of the relation, but also of the type K of the keys. Beside the function *lookup* the interface contains a function *delete*, deleting an element specified by its key and a function *member* performing a membership test based on key equality.

5.3.2 A Context-Independent Implementation

Queries consist of a context and a comprehension. In the introduction of this section an attempt is made to split the implementation of a query into a context-independent part, representing the operational semantics of the comprehension, and an implementation of the context. Implementing different contexts is the topic of the section 5.3.4, whereas it is the aim of this section to explore a context-independent implementation of comprehensions. In order to achieve this goal, those parts of a comprehension which are common in different contexts have to be recognized.

In the introduction of this chapter an implementation of a comprehension in two representative contexts is given. The implementations in the introduction are given in pseudocode. An implementation in TL, in the contexts *list* as an example for collecting the result elements in a bulk type and *count* as an example for the application of an aggregate function to the comprehension result, may look as follows:

```
get list                                  get count
  { p | p ← persons : p.age < 18 }          { p | p ← persons : p.age < 18 }

 ⇒                                         ⇒

begin                                     begin
 let result = list.new()                   let var count = 0
 let currentIter = persons.elements()      let currentIter = persons.elements()
 while not(currentIter.eoi()) do           while not(currentIter.eoi()) do
  let p = currentIter.get()                 let p = currentIter.get()
  if p.age < 18 then                        if p.age < 18 then
   result.insert(p)                          count := count + 1
  end                                       end
  currentIter.next()                        currentIter.next()
 end (* while *)                           end (* while *)
 result                                    count
end                                       end
```

The left query returns the list of all persons having an age less than eighteen years, whereas the right query computes the number of these persons. The implementation uses the environment introduced in section 5.3.1. Both implementations start with the initialization of variables, namely, of the variables *result* and *count*. It is assumed that the type of the list *persons* is a subtype of *QBulk(Person)*. This allows the application of the function *elements* to it yielding an iteration over its elements.

The *while*-loop iterates over the elements of the range *persons*. Each run through the body of the loop redefines the range variable *p* to be the actual element of the iteration. This binding establishes the environment necessary for the evaluation of the filter and the target expression that may refer to the identifier *p*.

If the actual element passes the filter, the target expression is evaluated. In case of the *list* context, the result of this evaluation is inserted into the resulting list, whereas in the case of the *count* context the variable *count* is incremented.

At the end of each pass through the body of the loop the state of the iteration is updated to point to the next element. The loop is terminated if the end of the iteration is reached. As described in section 5.3.1 this may be tested by a call of the function *currentIter.eoi*. The expected query result is the list containing the result elements in the *list* context and the number of these elements in the *count* context, i.e., the values of the variables *result* and *count* after execution of the *while*-loop, respectively. Since a sequence of expressions evaluates to the value of the last expression in TL, the last expression of the implementation denotes the result of the query. The variables *result* and *count*, therefore, have to be placed at the end of the implementation.

The context-dependent parts of the implementation are the initialization of the variables, the call of the functions *insert* and + inside of the filter, and the last expression of the implementation. To separate the context-independent parts of the implementation, the concrete function inside the filter is replaced by a function *action* that may be defined for every context in an adequate way. The definition of the function *action* and the initialization of the variables may be included in a context-dependent prologue preceding the implementation of the comprehension. The query result is denoted by a context-dependent epilogue. So the implementation of the comprehension is wrapped by a prologue and an epilogue. Now, a context-independent formulation of the implementation of a comprehension is possible. This is shown for the general case of a comprehension containing one range:

$$< context > \quad \{ \ < target > \ | < range\ variable > \ \leftarrow \ < range > \ : \ < filter > \quad \}$$

The implementation of this context-independent query looks as follows:

```
begin

  (* prologue ... *)

  let currentIter = < range > .elements()
  while not(currentIter.eoi()) do
    let < range variable > = currentIter.get()
    if < filter > then
      action( < target > )
    end
    currentIter.next()
  end (* while *)

  (* epilogue ... *)

end
```

The implementation performs different computations in different contexts. This principle is called *uninstantiated comprehension-thread* in the remainder of the text. The prologue determines the environment for the interpretation of the loop, especially it binds the identifier *action* to a concrete function. This equals to a determined control flow that may be instantiated with different actions *action* dependent on the context.

Until now, only one-range queries are considered. In case of a one-range query, the elements of the range are scanned sequentially. The iteration order for n-range queries is described in section 4.4: All elements of the n-th range are considered for the first element of the (n-1)th range before the second element of the (n-1)th range is considered. This rule propagates further for the preceding ranges.

Building Block	f^i_{rest}

generator
$rv_j \leftarrow r_j$
	let currentIter = r_j.elements()
	while not(currentIter.eoi()) **do**
	  let rv_j = currentIter.get()
	  f^{i+1}_{rest}
	  currentIter.next()
	end

filter f_j	**if** f_j **then** f^{i+1}_{rest} **end**
target expression t	action(t)

Figure 5.6: Case Analysis for Query Expressions

The implementation of a n-range query has to contain an additional loop for each range to iterate over its elements. In order to realize the specified iteration order, the loop for the second range has to be nested inside the loop for the first range and so on. The target expression is contained in the innermost loop. Note that the evaluation of filters, ranges, and target expressions can not only access the actual element of the direct enclosing iteration, but of all iterations embracing the evaluated expression. As an example, the evaluation of the filter in the following query is considered:

get *list* { *c* | *p* ← *persons, c* ← *p.children : p.age − c.age > 50* }

The query computes the list of all children that are more than 50 years younger than their father. The filter predicate involves elements of the second range as well as elements of the first range. The task of implementing the general case mainly consists of nesting loops of iterations and computing the elements of the result in the innermost loop.

A systematic way to evolve the implementation for the general case of a n-range query is to consider the different building blocks of a comprehension and their role in the implementation. It is assumed that the comprehension is parsed from left to right, starting with the first generator (compare to section 5.2.3). At each dividing alternative of the syntax, the remaining part of the comprehension is split into the following part and the rest of the comprehension. Table 5.6 shows the implementation of the remainder of the comprehension in the different cases. In each case f_{rest} represents the implementation of the rest of the query. The target expression is considered as last part for the definition of the operational semantics.

5.3.3 Comprehension Threads (Binding Sequences)

The implementation proposed in the previous section has the advantage of being compact and simple. Unfortunately, it is not adequate for yielding an iterator as comprehension result as proposed in section 4.4. An iterator performs a lazy evaluation: the comprehension is evaluated element-by-element on request. The *while*-loops in the implementation presented in section 5.3.2 perform a complete evaluation of the comprehension, which can be regarded as an instant materialization of the comprehension result in contrast to the intended lazy evaluation. The evaluation of the comprehension may be preempted at any point employing exceptions, but it is not possible to continue the evaluation later on request. All bindings representing the actual state of the iterations over all involved ranges are lost when leaving the nested *while*-loops.

One reason for the compactness of the considered solution is that the nested *while*-loops rebuild the scope of the range variables in a rather direct and natural way (compare also section 4.2). The solution presented in this section treats the aspects of bindings and scope of the range variables more explicitly. The range variables are, therefore, realized by updatable variables.

In a comprehension with n ranges the Cartesian product CP of the n ranges ($r_1, \cdots, r_n$) is considered. Some of the ranges are restricted by filters f_j ($1 \leq$ j $\leq$ p). Each element of CP is a sequence of n bindings

$$rv_1 = \ \ldots \qquad rv_2 = \ \ldots \qquad \cdots \quad rv_n = \ \ldots$$

where rv_i is the range variable of the range r_i ($1 \leq$ i $\leq$ n). Such a sequence of bindings forms the environment in which the target expression is evaluated. A range r_i and a filter f_j following the (i-1)-th generator are evaluated in an environment, where the range variables rv_1, ..., rv_{i-1} are bound. Each filter prohibits some of the binding sequences of CP. The target expression is only evaluated for bindings fulfilling all filters. The elements in CP are traversed in the order described on page 93: first, the range variables are all bound to the first possible values in the corresponding ranges. Then, the range variable rv_n is subsequently bound to the values of the range r_n. If the end of r_n is reached rv_{n-1} has to be actualized to the next possible value in the range r_{n-1}. The set of possible values of r_{n-1} may be restricted by a filter.

According to the building blocks, namely, filter, generator, and target expression, a comprehension may be split into k parts, each consisting of a representative of one of the building blocks. The target expression forms $part_0$. It is the main idea of the implementation to define a sequence of functions $nextBnds_l$, one for each part$_l$ ($l = 1, \cdots, k$) contained in a comprehension[8]. A function $nextBnds_l$ creates the next possible sequence of bindings in the above sense. In the implementation of $part_l$ the function $nextBnds_{l-1}$ is used to establish the next environment (sequence of bindings) for the evaluation of $part_l$. Function $nextBnds_l$ is defined in the implementation of $part_l$ for use in $part_{l+1}$. The definition of $nextBnds_l$ uses the definition of $nextBnds_{l-1}$.

Additionally, the implementation attempts to produce the first possible binding sequence B_{first} of all range variables $rv_1 \cdots rv_n$. This may only be achieved if the comprehension result is not empty. The bindings are established step by step in the implementation of the parts of the comprehensions. To allow a closer view, three cases for part$_l$ of the comprehension are examined:

[8] The target expression forms the initial part of the comprehension. Its implementation does not fit into this abstract description, since it does not define a function $nextBnds_l$.

1. **$\mathbf{part}_l$ is the i-th generator of the comprehension $\mathbf{rv}_i \leftarrow \mathbf{r}_i$:** As mentioned above, the implementation of the generator consists of two steps: the further development of the first possible binding B_{first} and the definition of a function $nextBnds_l$.

   (a) **$\mathbf{part}_l$ is the first generator (l=1, i=1):**

   The first generator plays a special role since it has no predecessors. If the end of the range r_1 is reached there are no more new bindings to be established. The evaluation of the whole comprehension is completed. If r_1 is not empty, the range variable rv_1 is initialized with the first element of r_1. The function $nextBnds_1$ actualizes the variable rv_1 to the next element of r_1.

   (b) **$\mathbf{part}_j$ is a further generator (i > 1):**

   In the case of the i-th generator, the first step consists of establishing a first binding for range variable rv_i. For this reason, r_i is evaluated in the environment produced by the previous parts. If r_i is empty, the function $nextBnds_{j-1}$ is called to establish the next possible binding. This call changes at least the binding of range variable rv_{i-1} and may also propagate to the bindings of former range variables, if the end of r_{i-1} is reached. Afterwards r_i has to be re-evaluated in this environment, since it may depend on one or more of the range variables $rv_1, \ldots, rv_{i-1}$. If the first non-empty range r_i is found the range variable rv_i is bound to its first element, thus extending the sequence of bindings B_{first} by a new binding. The function $nextBnds_j$ is defined as follows: It attempts to extract the next element from range r_i. If the end of range r_i is reached the next possible binding for $rv_1, \ldots, rv_{i-1}$ is established by a call of function $nextBnds_{j-1}$. Again in this case r_i has to be re-evaluated.

2. **$\mathbf{part}_l$ is the filter f_j following the i-th generator:**

   Since filters do not introduce new range variables the implementation may only actualize existing bindings to develop B_{first}. If the bindings defined by the previous parts do not fulfil filter f_j, the next possible binding is established by a call of the function $nextBnds_{l-1}$. This has to be repeated until a binding is found that fulfils filter f_j. This is the new version of B_{first}.

   The function $nextBnds_l$ is defined in a similar manner. It establishes new bindings by repeated calls of the function $nextBnds_{l-1}$ until a binding is achieved that fulfils filter f_j.

3. **$\mathbf{part}_l$ is the target expression (l=0):**

   The target expression may refer to all range variables and so it has to be evaluated in the environment where all range variables are already bound. Therefore the implementation of the target expression forms the last part of the implementation of the complete comprehension. The implementation of the whole comprehension forms a sequence of expressions. The sequence evaluates to the value yielded by evaluating the last expression in the sequence which is the last expression of the implementation of the target expression. Since the comprehension result is intended to be an iterator, this last expression has to be an iterator. The functions *eoi*, *get*, and *next* froming the interface of an iterator have to be defined:

- *eoi* has to return the Boolean value *true* if the end of the comprehension result is reached; this is the case if no more new bindings can be established;
- *get* has to return the actual element of the comprehension result; therefore it evaluates the target expression in the environment formed by the actual binding of the variables rv_1, ..., rv_n and returns the result;
- *next* has to establish the next bindings for the range variables; this is just what the last function in the defined sequence of functions $nextBnds_k$ does. It, therefore, may be used to define the function *next*.

An adequate combination of repeated calls of these three functions allows the implementation of all different contexts presented in section 4.3. This is described in the next section.

The presented ideas can be directly used to develop a concrete implementation for comprehensions. The implementation of the comprehension

get *list* { *p.name* | *p* ←*persons* :*p.age* < *18* }

looks as follows:

```
try                                      (* first generator *)
  let firstIter = persons.elements()
  if firstIter.eoi() then
    raise eocException
  end
  let var p = firstIter.get()
  let nextBnds() =                       (* definition of nextBnds1 *)
  begin
    firstIter.next()
    if firstIter.eoi() then
      raise eocException
    end
    p := firstIter.get()
  end

  while not(p.age < 18) do               (* filter *)
    nextBnds()
  end
  let nextBnds() =                       (* definition of nextBnds2 *)
  begin
    nextBnds()
    while not(p.age < 18) do
      nextBnds()
    end
  end

  let var eoi = false                    (* target expression *)
  tuple
```

```
    let eoi() = eoi
    let get() = p.name
    let next() = try
                   nextBnds()
                 when eoiException then
                   eoi := true
                 end (* try *)
   end (* tuple *)

  when eocException then                    (* empty comprehension result *)
    tuple
      let eoi() = true
      let get() = raise emptyException
      let next() = raise emptyException
    end
  end (* try *)
```

The implementation evaluates to a value of type *Iter(:String)*.

For a systematic development of an implementation for the general case, the comprehension is parsed from left to right starting with the first generator (compare to section 5.2.3 and 5.3.2). At each point the remaining part of the comprehension is split into the next building block and the rest of the comprehension. f_{rest} denotes the implementation of the rest of the comprehension. The first generator $rv_1 \rightarrow r_1$ is implemented by the following code:

```
try
  let firstIter = r1.elements()
  if firstIter.eoi() then
    raise eocException
  end
  let var rv1 = firstIter.get()
  let nextBnds1() =
  begin
    firstIter.next()
    if firstIter.eoi() then
      raise eocException
    end
    rv1 := firstIter.get()
  end
  frest
when eocException then
  tuple
    let eoi() = true
    let get() = raise emptyException
    let next() = raise emptyException
  end
end
```

firstIter is the iterator over the first range r_1. An exception *eocException* is raised, if the end of this iteration is reached. The last part of the implementation (**when** *eocException* $\cdots$) handles the case of an empty comprehension result (*eocException* is raised): an empty iterator is generated.

The implementation of further generators $rv_i \leftarrow r_i$ (i>1) is rather similar:

```
let var currentIter = r_i.elements()
while currentIter.eoi() do
  nextBnds()
  currentIter := r_i.elements()
end
let var rv_i = currentIter.get()
let nextBnds_l() =
begin
  currentIter.next()
  while currentIter.eoi() do
    nextBnds_{l-1}()
    currentIter := r_i.elements()
  end
  rv_i := currentIter.get()
end
f_rest
```

In order to restrict the number of different identifiers[9], the same variable name (*currentIter*) is used for the iterator of the second and all further ranges. This is also true for the function name *nextBnds*. It is rebound in every part of the comprehension implementation[10]. Note that binding the same identifier to a new value by the *let*-construct produces a new variable.

The implementation for a filter f_j looks as follows:

```
while not(f_j) do
  nextBnds_{l-1}()
end
let nextBnds_l() =
begin
  nextBnds_{l-1}()
  while not(f_j) do
    nextBnds_{l-1}()
  end
end
f_rest
```

[9]This is of interest for the automatic generation of the code by a syntax extension tool (see section 5.4).

[10]The indices $l-1$ and l are only inserted as reference to the above description of the solution. They are not part of the implementation.

It mainly consists of establishing new bindings by repeated calls of *nextBnds* until a binding which fulfils the filter is found.

Finally, the implementation for a target expression t is considered. Since it forms the last part of the implementation, it does not contain an implementation f_{rest} of the rest of the comprehension:

```
let var eoc = false
tuple
  let eoi() = eoc
  let get() =
    if not(eoc) then t else raise eocException end
let next() =
  if not(eoc) then
    try
      nextBnds_k()
    when eocException then
      eoc := true
    end (* try *)
  else
    raise eocException
  end
end (* tuple *)
```

The variable *eoc* is set to the Boolean value *true* if the end of the comprehension result is reached (*eocException* is raised). The function *get* and *next* are defined as described above.

5.3.4 An Extensible Library of Contexts

A query consists of a context and a comprehension. A context-independent implementation for comprehensions is developed in the previous sections. This section is devoted to the implementation of the contexts. In section 5.3.2, three components are identified to be part of the implementation of the contexts:

C_{init}: the initialization of variables used for the computation of the query result;

Action: the definition of a function *action* that is applied to the elements of the comprehension result;

C_{return}: the specification of the query result as last expression of the implementation of the entire query.

The implementation of a comprehension is formed by a sequence of expressions that evaluates to an iterator over the elements of the comprehension result. Therefore three functions *eoi*, *get* and *next* are defined (iterator interface), that can be used by the contexts. Calls of the functions *get* and *next* have to be combined in different ways for different contexts. This leads to a further part that has to be contained in the implementation of the context.

C_{wrap}: the combination of calls of the functions *get* and *next* in an adequate way for the specific context.

In section 4.3.2 two classes of contexts are distinguished: result-oriented contexts introduced by the keyword **get** and side-effect oriented contexts introduced by the keyword **do**. The considerations in this introduction and the first part of this section are restricted to the result-oriented contexts. The implementation of side-effect oriented contexts is discussed in the last part of this section.

Contexts as Objects

The contexts may be implemented as objects with a fixed interface. The interface common to all context objects may be defined by a type operator of the following form:

```
Let Context(E_r, R <:Ok) <:Ok
  Tuple
    wrap(:Iter.T(E_r)):R
  end
```

A context for a comprehension with element type E_r and result type R of the query is of type *Context*(E_r *R*).

The interface of each context object consists of a single function *wrap*. This function forms the implementations of the parts C_{wrap} and C_{return}. The initialization of the variables (C_{init}) and the definition of the function *action* are part of the definition of the context object.

The contexts for the presented query language are supported by a library named *contexts*. For each query a new context object with fresh local variables has to be defined. The context library, therefore, supports functions that generate context objects. The concrete contexts supported by the library will be discussed in the next part of this section.

Given the implementation of a comprehension with element type E_r, an object $context_i$ of type $Context(E_r R)$ may be utilized to embed the comprehension into the context implemented by $context_i$:

```
begin
  let compResult (* : Iter(E_r) *) =
      < implementation of the comprehension >
    context_i.wrap(compResult)
end
```

The result of the evaluation of the implementation of the comprehension is a value of type *Iter*(E_r). It is bound to an identifier *compResult*. The function *wrap* of the context is used to combine the functions *compResult.get* and *compResult.next* defined by the comprehension implementation in a way adequate for the chosen context. It returns the result of the query. Different ways of combination are discussed in the next part of this section.

Context	C_{init}	Action(t)	C_{wrap}	C_{return}
		Aggregates		
sum	**let var** sum = zero	sum := add(sum t)	loopAll	sum
max	**let var** max = opt.nil(:E_r)	**if** opt.null(max) **orif** less(opt.value(max) t) **then** max := opt.new(t) **end**	loopAll	opt.value (max)
min	**let var** min = opt.nil(:E_r)	**if** opt.null(min) **orif** less(t opt.value(min)) **then** min := opt.new(t) **end**	loopAll	opt.value (min)
count	**let var** count = 0	count := count + 1	loopAll	count
		Quantifiers		
some	**let var** result = false	**if** t **then** result := true **end**	**while** not(result) $\wedge$ not(eoi()) **do** action(get()) next() **end**	result
all	**let var** result = true	**if** not(t) **then** result := false **end**	**while** result $\wedge$ not(eoi()) **do** action(get()) next() **end**	result
		Bulks		
bag	**let** result = bag.new(:E_r)	result.insert(t)	loopAll	result
list	**let** result = list.new(:E_r)	result.insert(t)	loopAll	result
set	**let** result = set.new(equal)	result.insert(t)	loopAll	result

relation	**let** result =
	relation.new
	(key
	keyEqual
	equal)
		if result.member(t) **then**
		let e = result.lookup(key(t))
		if not(equal(e t)) **then**
		raise error
		end
		else
		result.insert(t)
		end
			loopAll	result

		Single Elements		
any	**let var** e = opt.nil(:E_r)	e = opt.new(t)	action(get())	opt.value(e)
the only	**let var** e = opt.nil(:E_r)	e := opt.new(t)	action(get()) **try** next() **raise** error **else ok end**	opt.value(e)

Figure 5.7: Implementation of the Contexts

The Implementation of Different Contexts

The set of contexts presented in section 4.3 is used as a source of examples for the discussion of implementation aspects. Before the implementation of the different contexts is described in detail, different ways of combining the functions *next* and *get* are examined. The classification of the result-oriented contexts according to their evaluation presented in section 4.3.2 is a starting point for this discussion.

First the contexts causing a complete evaluation of the comprehension are considered. Examples for this group of contexts are the computation of aggregating functions and the collection of the comprehension's elements into bulk types. In this case C_{wrap} consists of a loop construct:

while *not(eoi())* **do**
action(get())
next()
end

The loop body is an application of the function *action*, to a call of the function *get* followed by a call of the function *next*. The loop is executed until the end of the iteration over the comprehension result is reached (*eoi()= true*). Since this way of combining the functions is used by many contexts, it is defined as a function *loopAll* of type **Fun**$(E_r$ <:**Ok** *action*$(:E_r)$:**Ok** *comp :Iter*$(:E_r))$:**Ok**. This function is useful for the definition of new contexts.

The second group of contexts are those causing partial evaluation of the comprehension. For this group no uniform treatment for the combination of the functions *get* and *next* exists. In the context *any*, for example, it is sufficient to call *get* to yield the first element of the comprehension result. The function *next* is not needed at all.

In other cases the functions *get* and *next* have to be called in alternation until a certain condition is reached. As an example the context *some* is considered, where the condition is the retrieval of an element satisfying the Boolean condition formulated in the target expression of the comprehension. C_{wrap} can be defined as follows:

while *not(result)* $\wedge$ *not(eoi())* **do**
action(get())
next()
end

with *result* initialized to *false* and *action* defined as[11]:

let *action(t :Bool)* =
if *t* **then** *(* t represents the predicate *)*
result := true
end

The *while*-loop checks if the condition is fulfilled and if the end of comprehension result is reached. The third group of contexts enables lazy evaluation of the comprehension result. The contexts *iter* and *range* belong to this group.

[11]Note that *action* is applied to the result of the evaluation of the target expression (see section 5.3.2).

The implementation of the context *iter* os trivial, since it has to return an iterator over the elements of the comprehension result, and such an iterator *compResult* is already generated by the implementation of the comprehension (see section 5.3.3). The context, therefore, just passes the iterator implemented by the comprehension through.

The context *range* is dedicated as context for comprehensions used as ranges of other comprehensions. According to the implementation of the kernel of the query language presented in section 5.3.3, a range r has to enable the application of a function *p.elements()*. For this reason the iterator *compResult* describing the comprehension result has to be enclosed in a function closure. For the context *range* C_{return} can look as follows:

> **tuple**
>   **let** *elements()* = *compResult*
> **end**

C_{init}, *action*, and C_{wrap} are not needed for the contexts *iter* and *range*.

Table 5.7 shows how the contexts presented in section 4.3.3 may be implemented. The definitions for C_{init}, *action*, C_{wrap}, C_{return} for the different contexts are listed in the table. In section 4.3.3 a special class of contexts, the combinators, is identified. The implementations of these contexts are not included in the table. It is discussed separately in the next part of this section.

The Implementation of Combinators

The combinators are contexts that can be combined with other contexts (compare section 4.3.3). They form mappings from iterators to iterators. Since comprehension results are iterators, combinators can be applied to comprehensions as all other contexts and additionally further contexts can be applied to comprehensions already embedded in combinators.

Combinators have to compute an iterator from an existing iterator. For this reason *eoi*, *get*, and *next* have to be defined from existing functions *eoi*, *get*, and *next*.

As described in section 4.3.3 different kinds of combinators can be distinguished according to the degree of materialization of the comprehension result that is necessary. For contexts as *sortBy* a complete materialization of the comprehension result is necessary. The complete comprehension result has to be inspected and sorted before the first element of a sorted iterator can be determined. In order to enable efficient sorting, all elements of the comprehension result have to be inserted into a structure that supports sorting efficiently, e.g., into a search-tree. Then an iterator for this structure can be defined that iterates over the elements in the intended order. The implementation of this combinator is very similar to the implementation of the bulk type contexts and therefore not considered further.

For other combinators it is sufficient to keep track of the elements that have already been visited during the iteration. An example for this kind of combinators is *uniqueOn* that is used for duplicate elimination. To implement this combinator the elements that have already been traversed are inserted into a structure *temp* that enables efficient membership test. This leads to the following definition of the function $next_r$ of the resulting iterator:

> **let** $next_r()$ =

```
begin
  next()
  while temp.member(get()) and not(eoi()) do
    next()
  end
  temp.insert(get())
end
```

The implementation uses the functions *next*, *get*, and *eoi* defined by the implementation of the comprehension. The functions *get* and *eoi* can be directly adopted for the according functions of the resulting iterator.

For a third kind of combinators, exemplified by the combinators *fromTo* and *flatten*, no materialization of the comprehension result is necessary. The context *fromTo* has two parameters *lowerLimit* and *upperLimit*. For the implementation of this combinator the first *lowerLimit-1* elements of the comprehension result are skipped by *lowerLimit-1* calls of the function *next*. The iteration over the elements in position *lowerLimit* to *upperLimit* forms the resulting iterator. A counter can be employed to keep track of the position.

The combinator flatten can only be applied to comprehension results whose elements are bulk values. The resulting iterator has to specify an iteration over all elements of all bulk types contained as elements in the comprehension result. For this reason the function *elements* is subsequently applied to each element of the comprehension and the yielded iterators are used to visit the elements of the bulk types resembling the elements of the comprehension result. If the comprehension result is empty, the combinator returns an empty iterator. In the non-empty case the resulting iterator may be implemented as follows:

```
let var eoiFlag = false
let var temp = get().elements()
while temp.eoi() ∧ not(eoi()) do
  next()
  temp := get().elements()
end
if eoi() then eoiFlag := true end
tuple
  let eoi_r() = eoiFlag
  let get_r() = if not(eoi_r) then
                  temp.get()
                else
                  raise eoiException
                end
  let next_r() = if not(eoi_r) then
                   if not(temp.eoi()) then
                     temp.next()
                   else
                     while temp.eoi() ∧ not(eoi()) do
                       next()
                       temp := get().elements()
```

```
                end
                if eoi() then
                  eoiFlag := true
                end
              end
            else
              raise eoiException
            end
  end
```

Side-effect Oriented Contexts

The specification of a side-effecting context consists of a keyword *do* followed by a side-effect-oriented function *doIt*. This function is to be applied to each element of the comprehension result. If the function *doIt* is specified as parameter *action*, this kind of context can be implemented by an application of the function *loopAll* (see page 103). *loopAll* applies the function *doIt* to all elements of the iteration specified by the comprehension result.

5.4 Realization by Syntax Extension

In chapter 3 a syntax extension tool and its extension language is introduced. To specify the extensions to a host language one has to give a set of transformation rules describing the extensions and alternations to the aimed host language's grammar and the intended semantics.

The syntax extension rules have to be based on a LL(1) grammar. A LL(1) grammar for the developed query langauge is described in the first part of this section. This grammar constitutes the basis for the rules to the syntax extension tool, which are presented in the second part of this section. The rules are based on the imperative implementation of the query language described in section 5.3.

5.4.1 A LL(1) Grammar

In section 4.1.2 a grammar for the kernel of the proposed query language is presented. This grammar is taken as starting point for the grammar used formulating the syntax extension of the host language. The grammar for the syntax extension tool has to fulfil the LL(1)-property [Aho, Ullman 73]. Some changes to the grammar of section 4.1.2 are therefore necessary. This leads to the following grammar:

Query ::=
  get *Value* "{" *Comprehension* "}" |
  do *Value* "{" *Comprehension* "}"

Comprehension ::=
  Value "|" *Ide* "<-" *Value Rest*

Rest ::=
    "," *Ide* "<-" *Value Rest* |
    ":" *Value Rest* |
    ε

The second production describes the construction of the body of the comprehension. Each comprehension consists of a target expression and a generator followed by the rest of the comprehension. The rest of the comprehesion can be a further generator or a filter. These are the two non-empty alternatives of the production *Rest*[12]. The third alternative of the rule describes the situation that the end of the comprehension body is reached. This is denoted by an empty production (epsilon production).

The grammar has to be embedded into the grammar of the host language. This is done by defining *Query* as an further alternative of the rules for *Values* of the grammar of the language TL (see appendix B).

This grammar forms the basis for the rules driving the syntax extension tool. The rules are presented in the next section.

5.4.2 Rules for the Syntax Extension Tool

The LL(1) grammar described in the previous section determines the syntax of the query language. In order to implement the query language, a set of rules have to be specified. Each rule consists of a pattern describing the new syntax and a semantic interpretation of the introduced syntax. The pattern may contain placeholders that can be used in the semantic interpretation. The semantic interpretation is defined in the language TL enriched with the placeholders. To facilitate distinction from other identifiers the placeholders are preceded by a underscore in the presented rules.

The two different classes of contexts introduced by the keywords **do** and **get** are alternatives in the following rule:

```
queryG ::=
  "get" _context = valG "{" _comprehension = comprehensionG "}"
⇒
  let compResult = _comprehension
  _context.wrap(compResult)

|  "do"  _doIt = valG "{" _comprehension = comprehensionG "}"
⇒
  let compResult = _comprehension
  while not (compResult.eoi()) do
    _doIt(compResult.get())
    compResult.next()
  end
```

[12] Note that both alternatives start with a different symbol.

The first alternative consists of the keyword **get** followed by a concrete context and a comprehension. The construct returned by the rule *comprehensionG* represents an expression that evaluates to an iterator representing the comprehension result. This comprehension result is bound to the identifier *compResult* in the body of the rule. The use of the context function *wrap* follows the implementation presented in section 5.3.4.

In the second alternative the introducing keyword **do** is followed by a side-effect function *doIt*. Again the value returned by *comprehensionG* is bound to an identifier (*compResult*). A *while*-loop is specified to iterate over the comprehension result and to apply the function *doIt* to each element of the iteration.

The second rule is concerned with the semantic interpretation of the comprehension body.

```
comprehensionG ::=
  _nextBnds = NEW IDENTIFIER
   _target = valG "|" _rv = ideG "<-" _range = valG _rest = restG(_nextBnds)
⇒
   try
     let firstIter = _range.elements()
     if firstIter.eoi() then
       raise eocException
     end
     let var _rv = firstIter.get()
     let _nextBnds() =
       begin
         firstIter.next()
         if firstIter.eoi() then
           raise eocException
         end
         _rv := firstIter.get()
       end
       _rest
     let var eoc = false
     tuple
       let eoi() = eoc
       let get() = if not(eoc) then
                     _target
                   else
                     raise eocException
                   end
       let next() = if not(eoc) then
                      try
                        _nextBnds()
                      when eocException then
                        eoc := true
                      end
                    else
                      raise eocException
```

```
                end
        end (* tuple *)

      when eocException then
        tuple
          let eoi() = true
          let get() = raise eocException
          let next() = raise eocException
        end
      end (* try *)
```

The first building block is the target expression followed by the first generator and the rest of the comprehension body. The body of the rule is mainly a direct adoption of the implementation for the first generator and for the target expression presented in section 5.3.3.

If a non-global identifier, i.e., an identifier that is defined inside of one rule is to be used by another rule it has to be passed explicitly as derived attribute between the rules. This is the case for the variable *_nextBnds*[13]. The comprehension does not contain an according identifier for *_nextBnds*. It is, therefore, necessary to generate a new identifier for this purpose and a placeholder for this identifier in order to allow its use as derived attribute. Unfortunately the employed syntax extension tool does not support a facility for the generation of identifiers. The expression *NEW IDENTIFIER* is used to simulate this facility.

The implementation of the parts of the comprehension following the first generator are covered by the rule *restG*. The construct *_rest* produced by this rule is comparable to f_{rest} described in section 5.3.3. It is therefore inserted in the same place of the implementation of the generator.

As described above there exist three different cases for the rest of the query, resulting in a rule with three alternatives for *restG*:

```
 restG(_nextBnds) ::=
    _generator = generatorG(_nextBnds)
⇒       _generator
|   _filter = filterG(_nextBnds)
⇒       _filter
|   (* empty *)
⇒     (* *)
```

The alternatives for the generator and filter expression are described by separate rules:

```
 generatorG(_nextBnds) ::=
    "," _rv = ideG "<-" _range = valG  _rest = restG(_nextBnds)
⇒
     let var currentIter = _range.elements()
     while currentIter.eoi() do
```

[13] *_nextBnds* is used as identifier for the function that computes the next possible bindings of the range variables in the comprehensions (compare to section 5.3.3).

```
    _nextBnds()
    currentIter := _range.elements()
  end
  let var _rv = currentIter.get()
  let _nextBnds() =
    begin
      currentIter.next()
      while currentIter.eoi() do
        _nextBnds()
        currentIter := _range.elements()
      end
      _rv := currentIter.get()
    end
    _rest
```

```
filterG(_nextBnds) ::=
  ":" _predicate = valG  _rest = restG(_nextBnds)
⇒
  while not(_predicate) do
    _nextBnds()
  end
  let _nextBnds() =
    begin
      _nextBnds()
      while not(_predicate) do
        _nextBnds()
      end
    end
    _rest
```

The bodies of the rules are mainly direct adaptions of the implementation for further generators and filters, respectively. As in the case of the first generator the implementation of the rest of the expression is generated by the rule *restG* in *generatorG* and *filterG*.

The rules can be used to generate a new frontend that accepts queries containing the introduced comprehension-based syntax as well as expressions of the host language TL. The expressions containing the newly introduced syntax are transformed to abstract syntax trees of the language TL according to the rules and may then be further processed as TL-code.

Chapter 6

Conclusions and Further Research

In the previous section an extensible framework for a query language in a typed environment is developed. A functional and an imperative implementation for the query language are presented defining the operational semantics. The first part of this chapter gives a summary of this work.

Since the realization of a query language employing syntax extension technology is a novel approach, the main interest of the work is on the feasibility of the approach. Efficiency and optimization aspects are not discussed in detail. The second part of this chapter describes several starting points for optimizations of the query language.

The third part of the chapter summarizes the experiences made with the employed technology. The language TL used as host language for the query language and as target language for the syntax extension as well as the employed syntax extension tool are considered.

The last part of the section discusses topics for the further research related to the presented work.

6.1 Summary

The starting point of this work has been the goal to combine the user-friendly syntax and the declarativity of a built-in query language with the flexibility and extensibility of a query language realized by an add-on approach. The design goals extensibility, uniformity, flexibility, and optimizability have been formulated for the query language.

The developed query language is based on the comprehension notation. Comprehension are a concise notation with a well-defined semantics known from list manipulations in functional programming. The comprehensions form the kernel of the query language. In order to increase the flexibility and extensibility, the comprehensions are augmented by a novel concept, the contexts. The contexts fix the last processing step for the sequence of elements determined by the enclosed comprehension.

The query language is realized in a typed environment. By allowing arbitrary user-defined bulk types as input of a query and even different bulk types as ranges of a single query (mixed queries) the query language provides a high flexibility. Avoiding the restriction to special bulk types or special element types keeps the query language independent of a specific data model.

Instead of building in a fixed set of contexts into the query language, the query language just provides a framework for the definition of contexts. Concrete contexts are provided by libraries. Since neither the contexts nor the queried bulk types are built-in, one can speak of an extensible framework for a query language. The framework together with the extensible libraries supporting the bulk types and the contexts form a complete query language environment.

The syntax of the query language is described by a grammar. Typing aspects and the expressive power of the query language are also discussed. The semantics of the query language and especially of the mixed queries is based on the semantics of comprehensions described in literature and on bulk-morphisms. A representative set of contexts is described in order to illustrate the wide variety of possible contexts. Furthermore, enhanced features for the query language are discussed. It is shown that naming, parameterization, and recursion can be smoothly integrated into the presented framework for a query language.

A functional and an imperative implementation for a query language are presented. Both implementations use iterators as intermediate representation. The functional approach is based on a ringad implementation of bulk types. It is an extension of translation schemes for comprehensions discussed in literature. The representation of comprehension results by functional iterators allows the implementation of all contexts of the chosen representative sets but leads to efficiency problems for large comprehension results.

Iterators with states and an explicit treatment of the binding sequences for the range variables of the comprehensions form the basic concepts for the imperative implementation of the query language. In addition to the implementation of the query language kernel formed by the comprehensions, possible implementations for the representative set of contexts are considered.

The presented query language is realized employing syntax extension technology. On the basis of the imperative implementation of the query language rules for the used syntax extension technology have been developed.

6.2 Efficiency and Optimization

The chosen approach offers several starting points for optimizations. This is illustrated by figure 6.1:

1. The kernel of the presented query language is based on comprehensions. In [Trinder, Wadler 89] a set of optimization rules for comprehensions is described. These rules can be used to develop optimization rules for the presented language. Such rules have to be applied at a point that still allows an identification of the building blocks of the comprehensions. A good candidate for this purpose is the abstraction phase in which the parse trees are transformed into abstract syntax trees. In this phase the syntactical analysis is completed and the building blocks of the comprehensions can still be identified. The rules would have to be applied to intermediate representations of the comprehensions. Methods known from program transformation could be used to transform these representations according to the rules [Cordy et al. 91]. For many of these transformations it is necessary that the applied methods are capable to recognize

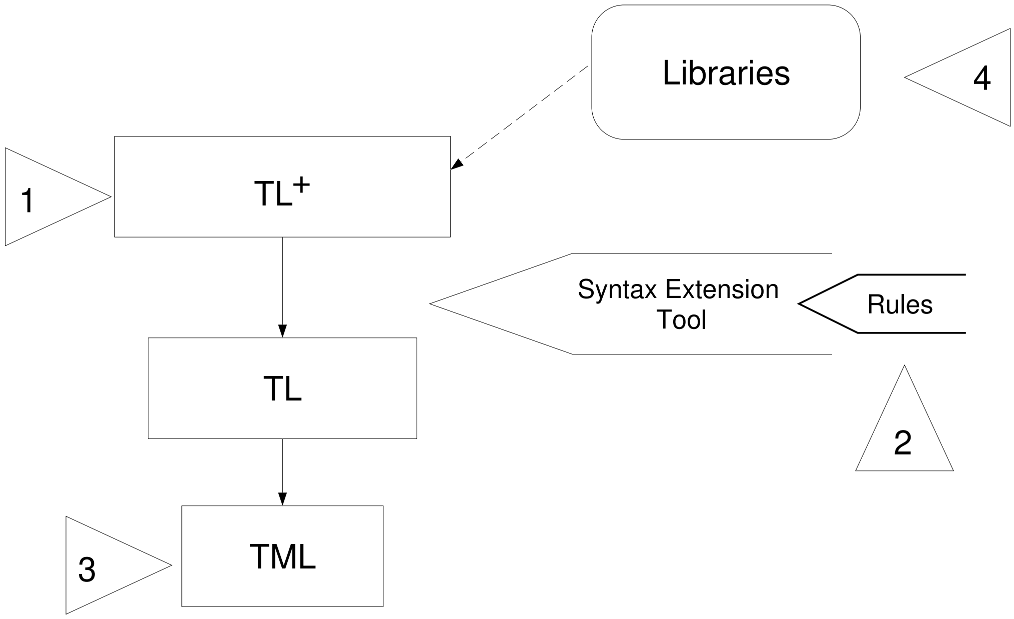

Figure 6.1: Different Starting Points for Optimization

the dependencies of the building blocks of the comprehension from the introduced range variables, since these dependencies prohibit certain transformations.

2. A further starting point for the improvement of the efficiency is the implementation of the query language. In chapter 5 different approaches for this implementation are introduced, namely, a functional and an imperative approach. Further implementations can be developed. The implementation of the query language is not built-in. Modification of the implementation is possible by changing the rules driving the syntax extension tool. Different prototypical realizations of the query language could be developed employing different implementations. The different versions could be compared in order to find the most efficient solution.

3. The queries are transformed to abstract syntax trees representing TL code by the syntax extension tool. After this transformation they are subject to the TL code optimization. The code representing the queries is generated automatically, driven by rules. This leads to repeating patterns in the code. The code optimizer could be tuned towards an effective optimization of these patterns. On the other hand the choice of the most efficient implementation of the query language is also influenced by the employed optimization strategies. Different implementations are more or less amendable towards an existing optimizer.

   Currently an optimizer supporting dynamic optimization of TML[1] code is developed at the University of Hamburg. This new development promises to support further optimization of control and data flows occurring during the evaluation of queries.

[1] TML is the Tycoon Machine Language an intermediate program representation based on Continuation Passing Style [Appel] emphasizing program compilation, transformation, and analysis.

4. Iterators play a central role in the implementation of the query language (compare section 4.4). Their optimization, therefore, is of special interest. Methods for iterator optimization proposed in literature [Liskov et al. 77] could be applied.

   The efficiency of the query language depends on the efficiency of the bulk type implementations. So it can be argued that these implementations are also a starting point for optimizations. The bulk types are not built-in; they are supported by libraries. If the implementation of a bulk type proves to be inadequate or not efficient enough for an application, it can be replaced by a more efficient implementation.

   It is of special interest for the efficiency of the query language how efficient the iteration over the elements of a bulk type is implemented, i.e., the function *elements*. In section 4.4, it is described that for certain bulk types different iteration orders over their elements are possible. In dependence of further information, e.g. access paths, the most efficient iteration order could be chosen by the implementation of the query language. This can be regarded as a further method towards optimization.

Each query has to be transformed employing the syntax extension tool before it is compiled (embedded queries) or interpreted (ad hoc queries). The performance of the syntax extension tool, therefore, is of great influence for the efficiency of the evaluation of ad-hoc queries and it increases the processing time of embedded queries.

6.3 Requirements towards the Technology

The operational semantics of the proposed query language is defined by an implementation in the Tycoon language (TL). A syntax extension tool with its associated extension language TLExt has been utilized for the embedding of the designed query language into the host language TL. This section describes the experiences made with the employed technology.

6.3.1 Language Technology

This section summarizes the experiences made during the implementation of the query language using the language TL. To summarize the experience with the Tycoon language and system, the statement can be made that the system is very well suited for the design and realization of an extensible framework for add-on query languages.

TL supports functional as well as imperative programming features. This allows an imperative and a functional implementation of the query language in the same linguistic environment. This facilitates the comparison of the different implementations of the query language. TL can be compiled as well as interpreted. This is also true for the extended language. For this reason it is possible to formulate ad-hoc queries that are evaluated by the interpreter. Furthermore queries can be embedded in programs performing further processing of the query results. These programs can be compiled and later executed. Libraries for frequently used queries could be developed.

Parametric polymorphism is a prerequisite for the support of user-defined bulk types with arbitrary element types. Parametric types are used to define type operators for new bulk types. These operators can be parameterized by an arbitrary type E yielding the type of a collection with element type E. Type operators are also used to define a uniform format for the iteration over bulk types (iterators). Polymorphic functions (parametric polymorphism on the value level) are needed for the definition of operations on the bulk types (e.g., for the operations *new* to create bulk types, see sections 5.2.1 and 5.3.1).

Functions are first-class values in TL. They, therefore, may be included as components into tuples. This feature is used for the definition of interfaces for state based bulk types and iterators in the imperative approach. Tuples with function components describe these interfaces (see section 5.3.1). Included concepts from imperative programming as exceptions and mutable values have supported the implementation of a state based iterator and of state based bulk types.

The facility to include functions and types as components into a tuple enables the definition of abstract data types in TL. The ringads forming the basis of the functional definition of the query language are realized by abstract data types (see section 5.2.1).

Higher-order functions have proven to be very useful for the definition of bulk types and for the implementation of the query language. Equality functions necessary for the definition of sets and relations are passed as function parameters (see sections 4.3.3). Another example of a higher-order function is the reduce generator presented in section 5.2.2. It is parameterized with the different functions in dependence of the chosen context and returns a reduce function for this context.

Bulk types have to fulfil certain restrictions if they are to be used as ranges of the query language. In the functional as well as in the imperative approach these restrictions are expressible by a subtype relationship to a common supertype. The subtype relationship between tuple types in TL is used to express these restrictions for the interfaces of the bulk types (see section 5.2.1 and 5.3.1).

The modularization capabilities of TL are used for the definition of the bulk type library (see section 5.2.1 and 5.3.1) and the context library (see section 5.3.4). The additional structuring facility offered by the possibility to group modules and interfaces into libraries is very helpful in this setting in order to achieve a well structured environment for the query language (see section 5.1).

6.3.2 The Syntax Extension Tool

The syntax of the designed query language is mapped by syntax extension to the host language TL. A syntax extension tool developed at the University of Hamburg [Schröder 93] is employed for this purpose. The implementation of the query language employing syntax extension technology has a twofold purpose. On the one hand the feasibility of such an approach is examined by developing a prototypic implementation. On the other hand, the requirements towards the employed tools are investigated. This section summarizes the experiences made with the used syntax extension tool.

The syntax extension tool proved to be very user-friendly. The semantic interpretation of the introduced syntax extensions can be formulated in the employed host language TL. This facilitated the development of the rules for the syntax extension tool. The implementation of the query language developed in section 5.3 was adapted to be used for the formulation of the rules. A further improvement of the user-friendliness is the treatment of bindings supported by the tool. Normally, in the expansion phase of the syntax extension variable captures can occur, but are resolved in the used tool. Furthermore, the tool guarantees the termination of expansions. So, the user is not concerned with the problem of termination when designing the rules for the extension. In addition, the rules are often less complex since no condition assuring termination have to be included. The conformity checks for the sorts have proven to be a further useful support for the development process of the rules. They allowed to detect faults in an early phase of the implementation.

Concerning the power of the tool most of the provided features proved to be necessary for the implementation of the query language. Since the grammar of the query language contains mutual recursive rules, it is crucial for the approach that the tool allows the definition of these rules for the syntax extension. Inherited attributes are needed for the propagation of the identifier that is used in several rules.

For the implementation of the query language an identifier for a function is needed, that is used by several rules, for redefinition of this function. For this reason this identifier has to be passed between the rules as inherited/derived attribute. Since the identifier is no part of the query it is not possible to define a placeholder for it and therefore it can not be passed as attribute between the rules. This restriction could be overcome by providing a mechanism that allows the creation of an identifier and the binding of a placeholder to it.

Further, for the syntax extension it is not possible to infer types that are necessary for the expansion. This problem raised when the functional approach of the query language was considered: the element type of the ranges of the comprehension is necessary for function signatures in the expansion process. This element type is inferable from the existing type information.

6.4 Further Research

The framework for a query language presented in this work represents a prototypical implementation. For this reason there are many interesting starting points for further research related to this approach. Furthermore, several extensions of the approach are possible.

In section 6.2 several starting points for optimization of the developed query language are described. The impact of the different approaches on the efficiency of the query language demands for further investigation. Currently an optimizer supporting dynamic optimization is developed at the University of Hamburg. It is of special interest to examine the benefit of this optimizer for the query language.

The proposed query language is not restricted to a particular data model. Instead it is designed to support different data models, for example, by allowing nested structures as well as flat structures. It is open to new data models, e.g., to object oriented models with object generating semantics. The impact of such new models and queries specific to such new models are of interest. They could be captured by extending the set of contexts in the environment of the query language. The necessary contexts and their realization is subject to further research.

The developed framework for a query language is based on definitions of bulk types given in literature. These are mainly homogenous bulk types: all elements of an instance of a bulk type have a common type. Although, restricted forms of inhomogenity are also possible: The element type can be specified by a variant type allowing the choice between a fixed number of different types for the elements. A further possibility is the use of subtype polymorphism enabling the management of elements of arbitrary subtypes of a common supertype in a single collection. This approach is restricted because of the resulting loss of type information. Employing dynamic types for the elements enables the definition of unrestricted inhomogenous bulk types. Each element carries its type information that can be inspected when the element is retrieved from the bulk type. Related topics for further research are the computational complexity of a query language in the presence of inhomogenous bulk types, the class of queries evolving from such a setting, and their realization in the proposed framework.

A further application of the developed query language is querying the meta data, i.e., the database schema and related information. This is of special interest for object oriented systems where the underlying model carries inherently much more information than relational database schemata [Kifer et al. 92]. Querying meta data is only possible if the meta data are given in an adequat format, i.e., they have to be described in the host language of the query language. It could be examined which classes of queries are important in such a scenario. Futhermore, it could be investigated if information retrievable by such queries could support optimization tasks.

An approach for the integration of recursive queries into the presented framework is discussed in section 4.5.3. The proposed approach requires further investigation, in especially, with respect to the query evaluation strategies and the structures needed to support the evaluation. Of special interest is the space complexity of such structures. Different evaluation strategies for recursive queries could be implemented.

Appendix A

Contexts in other Query Languages

A.1 DBPL

There are three kinds of expressions for the formulation of queries in DBPL, namely, *selective access expressions*, *constructive access expressions*, and *quantified expressions*. Selective and constructive access expressions are first class values if the language. They can be named and parameterized by relation expressions, as well as by other values. DBPL supports a uniform naming and parameterization construct for them, introduced by the keywords SELECTOR and CONSTRUCTOR, respectively. In the following a list of further features of DBPL, relevant to the discussion, is given:

1. **Bulk Constructors of Bulk Type 'Relation':** The context for constructing a copy of a relation bulk type is specified by a syntactic construct adapted from set theory. An example is:

   PersonRel{ < *access expression* > }

   with $PersonRel$ the type of the newly defined relation.

2. **Iteration:** The iteration over the elements of an access expression is defined with the *FOR* − *DO* and *WHILE* − *DO* constructs.

3. **Assigment Operations:** A relation variable rel of type $RType$ can be updated by a relation expression rex using one of the relation update operators ':+', ':-', or ':&' for insertion, deletion, and update, respectively.

4. **Standard Procedures:**

   A set of standard procedures related to relational expressions are defined in DBPL. The operations take a relation expression and a variable of an adequate type as parameter. Some of them rely on an order of the elements. This order is determined by the key of the relation if existent:

   Lowest   *selects the first element of a relation*

Highest	*selects the last element of a relation*
This	*selects the element in the relation equivalent to a given ordinal*
Card	*returns the number of relation elements in a given relation*
Next	*selects the next element of a relation*
Prior	*selects the prior element of a relation*

The access expressions can be pllaced in different contexts. These contexts are listed in the following table together with the corresponding contexts of the query language presented in this work.

Context in T_{L}^+	Language Construct	Description of the Language Construct
get relation	RType { ... }	Relation Expression denotes sets of relation elements
do ⋯	FOR ⋯ DO	Iteration as one-element-at-a-time processing possibly applying functions to each single element.
get some	SOME	Existential quantification
get all	ALL	Universal quantification
do insert	:+	Relation insert
do delete	:–	Relation delete
do update	:&	Relation replacement
get count	CARD	Standard procedure returning the number of elements in a relation.

Note that only the first two contexts in the table can be used as direct contexts of access expressions. In DBPL quantified boolean expressions, assignment equations, and the presented standard procedures can not be directly applied as contexts for selective and constructive access expressions. They can only be applied to access expressions enclosed in a relation constructor since they are defined on relation expressions.

A.2 COOL

The COOL query language (COOL-QL) is an extension of the (nested) relational algebra. COOL-QL is an object-oriented query language where the inputs and the outputs of the query operations are sets of objects at a time. The query operators have object-preserving semantics, such that the results of queries are some of the the existing objects from the data base. The kernel of the query language is formed by operators *select* and *project* to express element selection and element projection, respectively. Furthermore, it contins the set operations *union*, *intersect*, and *difference*. In COOL-QL object preserving operators and value generating operators are distinguished. COOL-QL allows query expressions to operate on subtype hierarchies of the data model (*guard* operator) and operations allowing explicit object evolution (*create, gain, lose, and delete*). The following table lists possible contexts for queries (set-expressions):

Context in TL$^+$	Language Construct	Description of the Language Construct
get set	extract	Generates a set of tuples from a set of objects specified by a set expression.
get the	pick	Destructs a singleton set into its member value.
do ...	apply_to_all	Takes a sequence of update operations and executes it for each element specified by a given set expression.

A.3 Fibonacci

Fibonacci is an object-oriented language incorporating subtyping. Objects are modelled as entities with an immutable identity and a mutable, encapsulated state. The supported bulk types: sequences, class, and association types are non-standard bulk types.

Sequences are ordered collections of homogenous values with duplicates. Classes are very similar to sequences, but in contrast to them they allow updates and the formulation of constraints. Associations are sequences of tuples. They can be updated and it is possible to define constraints on them that connect association fields with classes.

The query language kernel is formed by an operator *where* to express filter predicates, an operator *in* to introduce range variables, and operators *times* and *join* to express carthesian products and natural joins. Furthermore it supports two operators *project* and *out* for the formulation of projections. Queries (sequences) can be placed in the following contexts:

Context in TL$^+$	Language Construct	Description of the Language Construct
expressible in the query language	for .. concat ..	Iterates over a sequence applying an sequence-returning operator to each element; the results are concatenated
get the	the	Returns the element of the singleton sequence
get any	pick	Returns one element of a sequence non-deterministically
expressible in the query language	.. group by ..	Returns a partition of the elements of a sequence into subsequences
get uniqueOn	setof ..	Eliminates duplicates from a sequence

A.4 O_2SQL

O_2SQL is the query language of the object oriented database system O_2. It is a functional query language in the sense that a query is a function whose arguments may be other queries or functions. The data model distinguishes objects and tuple values. It supports three kind of bulk types: lists with positions, sets with duplicates (i.e., bags) and unique sets (no duplicates). Queries are interpreted in an interactive mode and it is possible to pass queries from programs to the query interpreter. The query language kernel of O_2SQL is mainly SQL. Operations on base types and tuples and the construction of tuples, lists, and sets are also viewed as queries in the language. The following operators of the language can be interpreted as contexts:

Context in TL$^+$	Language Construct	Description of the Language Construct
get all	forall x in y : p(x)	Polymorphic universal quantifier for lists and sets
get some	exists x in y : p(x)	Polymorphic existential quantifier for lists and sets
get list fromTo (i i)	L[i]	Returns the i-th element of a given list
get list fromTo (i j)	L[i:j]	Returns for a given list all elements of the list starting with position i and ending with position j
get list fromTo (1 1)	first(x)	Returns the first element of any list x
—	last (y)	Returns the last element of any list x
get min	min	returns for a list of numerical values the smallest element and for strings the first string in ASCII alphabetical order
get max	max	Returns for nay list x of numerical value its largest element and for strings the last in ASCII alphabetical order
get count	count(x)	Returns the length/cardinality of a list/set
get sum	sum	Returns for any list/set of real of integer values the sum of its elements.
—	avg(x)	Returns for any list/set of real or integer values the average of its elements
get set	listtoset(x)	Returns for a list the corresponding set with duplicates removed
get list flatten **get** set flatten	flatten(x)	Flattens lists of lists and lists of sets to a flat list; flattens sets of lists and sets of sets to a flat set
get the	element	Returns for any singleton set its members value
get uniqueOn	unique, distinct	Eliminates duplicates in the resulting set/list
get list sortBy	sort ... in ... by $f_1, \ldots, f_n$	Returns a list containing all elements of list/set sorted by the keys given by the f_i's. Complex object values whose keys are identical are solved internally. The order relations are system defined.

Appendix B

Grammar of the Language TL$^+$

B.1 Productions

B.1.1 Compilation Units

The grammar of TL$^+$ is described by the following productions that define a non-ambiguous LL(1) grammar. The grammar of TL$^+$ extends that of TL [Matthes 92a] by the rules presented in section 5.4.1. *Unit* is the root production for the language. The production of $Value_3$ is extended by a branch for *Query*.

Unit::=
    (*Library* | *Interface* | *Module* | *Import* | *Bindings*) ";"*;*
Library::=
    library *identifier Import* **with** { *ComponentSignatures* }
    [**hide** { *identifier* }] **end***;*
ComponentSignatures::=
    library { *identifier* } |
    interface { *identifier* } |
    module { *identifier* ":" *identifier* }*;*
Interface::=
    interface *identifier Import* **export** *Signatures* **end***;*
Module::=
    module *identifier Import* **export** *Bindings* **end***;*
Import::=
    [**import** { [":"] *identifier* }]*;*

B.1.2 Bindings

Bindings::=
    { *TypeBindings* | *ValueBindings* | **open** *ValueIde* [":" *Type*]
      ":" [**Dyn**] *Type* | [**var**] *Value* }*;*
TypeBindings::=

{ **Let** [**Rec**] *TypeBinding* { **and** *TypeBinding* } };
TypeBinding::=
[**Dyn**] *TypeIde Parameters* ["<:" *Type*] "=" *Type;*
ValueBindings::=
{ **let** [**rec**] *ValueBinding* { **and** *ValueBinding* } };
ValueBinding::=
[**var**] *ValueIde Parameters* [":" *Type*] "=" *Value;*

B.1.3 Values

Value::=
Value$_1$ { (**orif** | **andif** | *colonInfix*) *Value*$_1$ };
Value$_1$*::=*
Value$_2$ { *infix* *Value*$_2$ };
Value$_2$*::=*
Value$_3$ { "(" *Bindings* ")" | "?" *CaseIde* | "!" *CaseIde* |
"." *FieldIde* | "[" *Value* "]" |
of *Bindings Location* **end** };
Value$_3$*::=*
"{" *Value* "}" |
ValueIde |
ok |
int | *char* | *string* | *real* | *longreal* |
fun "(" *Signatures* ")" [":" *Type*] *Location Value* |
tuple *Location* [**case** *CaseIde* **of** *Type* [**with**]] *Bindings* **end** |
record *Location Bindings* **end** |
extend *Value* **with** *Bindings* **end** |
array *Location Bindings* **end** |
exception *Value* [**with** *Signatures* **end**] |
begin *Location Bindings* **end** |
if *Value* **then** *Bindings* { **elsif** *Vakue* **then** *Bindings* }
[**else** *Bindings*] **end** |
case [**of**] *Value* { **when** *CaseIdeList* [**with** *ValueIde*] **then** *Bindings* }
[**else** *Bindings*] **end** |
typecase { *ValueIde* "." } *TypeIde* { **when** *Type* **then** *Bindings* }
[**else** *Bindings*] **end** |
loop *Bindings* **end** |
exit |
while *Value* **do** *Bindings* **end** |
for *ValueIde* "=" *Value* (**upto** | **downto**) *Value* **do** *Bindings* **end** |
try *Bindings* { **when** *Value* [**with** *ValueIde*] **then** *Bindings* }
[**else** *Bindings*] **end** |
raise *Value* [**with** *Bindings* **end**] |
reraise |
assert *Value* |
Query;

Location::=
  [**in** *Value*];
Queries::=
  get *Value* "{" *Comprehension* "}" |
  do *Value* "{" *Comprehension* "}";
Comprehesions::=
  Value "|" *ValueIde* "<-" *Value* [*Rest*];
Rest::=
  "," *ValueIde* "<-" *Value* [*Rest*] |
  ":" *Value* [*Rest*];

B.1.4 Signatures

Signatures::=
  { *TypeSignatures* | *ValueSignatures* | *TypeBindings* | **Repeat** *Type* };
TypeSignatures::=
  [**Dyn**] [*TypeIdeList Parameters*] "<:" *Type;*
ValueSignatures::=
  [**var**] [*ValueIdeList Parameters*] ":" *Type;*
Parameters::=
  { "(" *Signatures* ")" };

B.1.5 Types

Type::=
  $Type_1$ { *colonInfix* $Type_1$ };
$Type_1$::=
  $Type_2$ { *infix* $Type_2$ };
$Type_2$::=
  $Type_3$ { "(" { *Type* } ")" };
$Type_3$::=
  "{" *Type* "}" |
  { *ValueIde* "." } *TypeIde* |
  Ok | **Nok** |
  Fun "(" *Signatures* ")" ":" *Type* |
  Tuple *Signatures* { **case** *CaseIdeList* [**with** *Signatures*] } **end** |
  Record *Signatures* **end** |
  Exception [**with** *Signatures* **end**] |
  Oper "(" *Signatures* ")" ["<:" *Type*] *Type;*

B.1.6 Identifier

ValueIdeList, TypeIdeList, CaseIdeList::=
  Ide { "," *Ide* };
Ide, ValueIde, TypeIde, FieldIde, CaseIde::=

identifier | *infix* | *colonInfix* | `"{"` *Ide* `"}"`;

Bibliography

Abiteboul, Beeri 88: S. Abiteboul and C. Beeri. "On the Power of Languages for the Manipulation of Complex Objects". Technical Report 846, INRIA, France, 1988. (Manuscript of the revised version from October 1992).

Adrion, Branstad 81: W.R. Adrion and M.A. Branstad. "The Functional Data Model and the Data Language DAPLEX". *ACM Transactions on Database Systems*, 6(1):140–173, 1981.

Aho, Ullman 73: A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling*, Volume I. Prentice Hall, 1973.

Albano et al. 85: A. Albano, L. Cardelli, and R. Orsini. "Galileo: A Strongly Typed, Interactive Conceptual Language". *ACM Transactions on Database Systems*, 10(2):230–260, 1985. (also in *Readings in Object-Oriented Database Systems*, S.B. Zdonik and D. Maier (eds.), Morgan Kauffmann, San Mateo, California, pages 147–161, 1990).

Albano et al. 91: A. Albano, G. Ghelli, and R. Orsini. "A Relationship mechanism for strongly typed object-oriented database programming languages". In: *Proceedings of the Seventeeth International Conference on Very Large Databases, Barcelona (Catalonia, Spain), September 3–6, 1991*, pages 565–576. Morgan Kaufmann Publishers, 1991. (also as Fide Technical Report 91/17, Department of Computing Science, University of Glasgow).

Albano et al. 93: A. Albano, R. Bergamini, Ghelli G., and R. Orsini. "An Introduction to the Database Programming Language Fibonacci". Fide$_2$ Technical Report 93/64, Department of Computing Science, University of Glasgow, 1993.

Appel : A. Appel. *Compiling with Continuation.*

Astrahan et al. 76: M.M. Astrahan, M.W. Blasgen, D.D. Chamberlin, K.P. Eswaran, J.N. Gray, P.P. Griffiths, W.F. King, R.A. Lorie, P.R. McJones, J.W. Mehl, G.R. Putzolu, I.L. Traiger, B.W. Wade, and V. Watson. "System R: Relational Approach to Database Management". *ACM Transactions on Database Systems*, 1(2):97–137, 1976.

Atkinson et al. 90: M.P. Atkinson, P. Richard, and P.W. Trinder. "Bulk Types for Large Scale Programming". 1990. (also as Rapport Technique Altair 60-90 Nov. 1990, GIP Altair, France).

Atkinson et al. 93: M.P. Atkinson, P.W. Trinder, and D.A. Watt. "Bulk Type Constructors". Fide Technical Report FIDE/93/61, Department of Computing Science, University of Glasgow, 1993.

Atkinson, Buneman 87: M.P. Atkinson and O.P. Buneman. "Types and Persistence in Database Programming Languages". *ACM Computing Surveys*, 19(2):105–190, 1987.

Bancilhon et al. 87: F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. "FAD, a Powerful and Simple Database Language". In: P.M. Stocker, W. Kent, and P. Hammersley, (eds.), *Proceedings of the Thirteenth International Conference on Very Large Databases, Brighton, England, September 1–4, 1987*, pages 97–105. Morgan Kaufmann Publishers, 1987.

Bancilhon et al. 92: F. Bancilhon, S. Cluet, and C. Delobel. "A Query Language for O_2". In: F. Bancilhon, C Delobel, and P. Kanellakis, (eds.), *Building an Object-Oriented Database System – The Story of O_2*, chapter 11. Morgan Kaufmann Publishers, 1992.

Bancilhon, Ramakrishnan 86: F. Bancilhon and R. Ramakrishnan. "An Amateur's Introduction to Recursive Query Processing Strategies". In: C. Zaniolo, (ed.), *Proceedings of the International Conference on Management of Data, Washington, D.C., May 28–30, 1986*, pages 16–52. Association for Computing Machinery, 1986. (SIGMOD RECORD, Volume 15, Number 2, June 1986).

Beeri, Milo 92: C. Beeri and T. Milo. "Functional and Predicative Programming in OODB's". In: *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1992*. Association for Computing Machinery, 1992.

Beeri, Ta-Shma 94: C. Beeri and P. Ta-Shma. "Bulk Data Types - A Theoretical Approach". In: C. Beeri, A. Ohori, and Shasha D.E, (eds.), *Proceedings of the Fourth International Workshop on Database Programming Languages – Object Models and Languages, New York City, USA 30 August – 1 September, 1993*, pages 80–96. Springer-Verlag, 1994.

Beeri 92a: C. Beeri. "Object-Oriented Databases – Models and Query Languages". In: A. Pirotte, C. Delobel, and G. Gottlob, (eds.), *Proceedings of the 3rd International Conference on Extending Database Technology, Vienna, Austria, March 23–27, 1992*, Volume 580, *Lecture Notes in Computer Science*. Springer-Verlag, 1992. Tutorial held at the conference.

Beeri 92b: C. Beeri. "On Monads and Comprehensions". (Lecture given for the Database and Information Science Group at the University of Hamburg, on September 17th, 1992), 1992.

Bird, Wadler 88: R.S. Bird and P. Wadler. *Introduction to functional programming.* Prentice Hall International Series in Computer Science. Prentice Hall, 1988.

Breazu-Tannen, Subrahmanyam 91: V. Breazu-Tannen and R. Subrahmanyam. "Logical and Computational Aspects of Programming with Sets/Bags/Lists". In: J.L. Albert, B. Monien, and M.R. Artalejo, (eds.), *Proceedings of the 18th International Colloquium on Automata, Languages and Programming Madrid, Spain, July 8–12, 1991*, Volume 510, *Lecture Notes in Computer Science*, pages 60–75. Springer-Verlag, 1991.

Bültzingsloe wen 87: G. von Bültzingsloe wen. "Translating and Optimising SQL Queries Having Aggregates". In: *Proceedings of the Thirteenth International Conference on Very Large Databases, Brighton, England, September 1–4, 1987*, pages 235–245. Morgan Kaufmann Publishers, 1987.

Burstall 69: R.M. Burstall. "Proving properties of programs by structural induction". *Computer Journal*, 12:41–48, 1969.

Cardelli et al. 91: L. Cardelli, S. Martini, J.C. Mitchell, and A. Scedrov. "An extension of system F with subtyping". In: T. Ito and A.R. Meyer, (eds.), *Proceedings of the International Conference on Theoretical Aspects of Computer Software, Sendai, Japan, September 24–27, 1991*, Volume 526, *Lecture Notes in Computer Science*, pages 750–770. Springer-Verlag, 1991.

Cardelli et al. 94: L. Cardelli, F. Matthes, and M. Abadi. "Extensible Grammars for Language Specialization". In: C. Beeri, A. Ohori, and Shasha D.E, (eds.), *Proceedings of the Fourth International Workshop on Database Programming Languages – Object Models and Languages, New York City, USA 30 August – 1 September, 1993*, Workshops in Computing, pages 11–31. Springer-Verlag, 1994.

Cardelli, Wegner 85: L. Cardelli and P. Wegner. "On understanding types, data abstraction, and polymorphism". *ACM Computing Surveys*, 17(4):471–522, 1985.

Cardelli 89: L. Cardelli. "Typeful Programming". Digital Systems Research Center Report No. 45, Digital Systems Research Center, Palo Alto, California, 1989.

Cardelli 90: L. Cardelli. "The Quest Language and System (Tracking Draft)". (shipped as part of the Quest V.12 system distribution), 1990.

Cardelli 93: L. Cardelli. "An implementation of $F_{<:}$". Digital Systems Research Center Report No. 97, Digital Systems Research Center, Palo Alto, California, 1993.

Carey et al. 88: M.J. Carey, D.J. DeWitt, and S.L. Vandenberg. "A data model and query language for EXODUS". In: *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Chicago, Illinois*, pages 413–423. acm, 1988. SIGMOD RECORD Volume 17, Number 3, September.

Ceri et al. 90: S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer-Verlag, 1990.

Chamberlin, et al. 76: D.D Chamberlin and et al. "SEQUEL 2: A uniform mechanism to data definition, manipulation, and control". *IBM System Journal*, 20(6):560–575, 1976.

Cluet, Moerkotte 94: S. Cluet and G. Moerkotte. "Nesting Queries in Object Bases". In: C. Beeri, A. Ohori, and Shasha D.E, (eds.), *Proceedings of the Fourth International Workshop on Database Programming Languages – Object Models and Languages, New York City, USA 30 August – 1 September, 1993*, Workshops in Computing, pages 226–242. Springer-Verlag, 1994. (also appeared as Fide$_2$ Technical Report FIDE/93/69).

Codd 72: E.F. Codd. *Relational Completeness of Database Sublanguages*, Volume 6, *Database Systems: Courant Computer Science Series.* Prentice Hall, 1972.

Cordy et al. 91: J.R. Cordy, C.D. Halpern-Hamu, and E. Promislow. "TXL: A Rapid Prototyping System for Programming Language Dialects". *Computer Languages*, 16(1):97–107, 1991.

Damas, Milner 82: L. Damas and R. Milner. "Principal type-schemes for functional languages". In: *Conference Record of the Ninth Annual ACM Syposium on Principles of Programming Languages, Albuquerque, New Mexico, January 25–27, 1982*, pages 207–212. Association for Computing Machinery, 1982.

Date 89: C.J. Date. *A Guide to the SQL Standard.* Addison-Wesley, 2nd edition, 1989.

Dayal 87: U. Dayal. "Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers". In: P.M. Stocker, W. Kent, and P. Hammersley, (eds.), *Proceedings of the Thirteenth International Conference on Very Large Databases, Brighton, England, September 1–4, 1987*, pages 197–208. Morgan Kaufmann Publishers, 1987.

Feather 87: M.S. Feather. "A survey and classification of some program transformation approaches and techniques". In: L.G.L.T. Meertens, (ed.), *Program Specification and Transformation*, pages 165–196. North-Holland Publishing Company, 1987.

Fegaras 94: L. Fegaras. "Efficient Optimization of Iterative Queries". In: C. Beeri, A. Ohori, and Shasha D.E, (eds.), *Proceedings of the Fourth International Workshop on Database Programming Languages – Object Models and Languages, New York City, USA 30 August – 1 September, 1993*, pages 200–225. Springer-Verlag, 1994.

Field, Harrison 88: A.J. Field and P.G. Harrison. *Functional Programming.* Addison-Wesley, 1988.

Ghelli et al. 92: G. Ghelli, R. Orsini, A.P. Paz, and P. Trinder. "Design of an Integrated Query and Manipulation Notation for Database Languages". In: *Proceedings of the 25th Hawaii International Conference on System Sciences*, pages 777–786. The Institute of Electrical and Electronics Engineers, Inc., 1992.

Goldberg, Robson 83: A. Goldberg and D. Robson. *Smalltalk-80: The language and its implementation.* Addison-Wesley, 1983.

Gougen et al. 76: J.A. Gougen, J.W. Thatcher, and E.G. Wagner. "An initial algebra approach to the specification, correctness and implementation of abstract data types.". Technical Report IBM Research Report RC 6487, IBM Almaden Research Center, San Jose, California, 1976.

Heijenoord 67: J. van Heijenoord, (ed.). *From Frege to Gödel. A source book in mathematical logic.* Harvard University Press, 1967.

Heytens, Nikhil 91: M.L. Heytens and R.S. Nikhil. "List Comprehensions in Agna, A Parallel Persistent Object Store". In: J. Hughes, (ed.), *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 26–30, 1991*, Volume 523, *Lecture Notes in Computer Science*, pages 569–591. Springer-Verlag, 1991.

Hudak 89: P. Hudak. "Conception, Evolution, and Application of Functional Programming Languages". *ACM Computing Surveys*, 21(3):359–411, 1989.

Hull, Su 90: R. Hull and J. Su. "On bulk data type constructors and manipulation primitives: A framework for analysing expressive power and complexity". In: *Proceedings of the Second International Workshop on Database Programming Languages, Salishan, Oregon, June 4–8, 1989*, pages 396–410. Morgan Kaufmann Publishers, 1990.

Ichbiah 83: J. Ichbiah. "The Programming Language Ada: Reference Manual". Technical Report ANSI/MIL-STD-1815A-1983, Joint Program Office, Department of Defense, Washington, D.C., 1983.

Ingres 89: Relational Technology, Inc., Alameda, California. *EQUEL User's Guide*, UNIX release 6.2 edition, 1989.

ISO9075 92: "Informations technology - Database Language SQL2, Document ISO/IEC-9075-1992". International Standards Organization, 1992.

Kato et al. 90: K. Kato, T. Masuda, and Y. Kiyoki. "A Comprehension-Based Database Language and its Distributed Execution". In: *Proceedings of the 10th International Conference on Distributed Computing Systems, Paris, France, May 28 – June 1, 1990*, pages 442–449. The Institute of Electrical and Electronics Engineers, Inc., 1990.

Kifer et al. 92: M. Kifer, W. Kim, and Y. Sagiv. "Querying Object-Oriented Databases". In: *Proccedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, June 2–15, 1992*, pages 393–402. Association for Computing Machinery, 1992. SIGMOD Record, Volume 21, Issue 2, June, 1992.

Kirch, Müßig 92: F. Kirch and S. Müßig. "Entwicklung eines generischen Datenbankbrowsers in einer polymorphen Programmiersprache". Studienarbeit, Fachbereich Informatik, Universität Hamburg, Germany, 1992. (in German).

Koch et al. 83: J. Koch, M. Mall, P. Putfarken, M. Reimer, J.W. Schmidt, and C.A. Zehnder. "Modula/R Report, Lilith Version". Lidas memo, Department Informatik, ETH Zürich, Switzerland, 1983.

Kohlbecker 86: E. E. Kohlbecker. *Syntactic Extensions in the Programming Language Lisp.* Technical report, University of Indiana, 1986.

Lambek, Scott 86: J. Lambek and P.J. Scott. *Introduction to higher order categorial logic*, Volume 7, *Cambridge studies in advanced mathematics.* Cambridge University Press, 1986.

Landin 64: P.J. Landin. "The Mechanical Evaluation of Expressions". *Computer Journal*, 6(4):308–320, 1964.

Lécluse, Richard 89: C. Lécluse and P. Richard. "The O_2 Database Programming Language". In: P.M.G. Apers and G. Widerhold, (eds.), *Proceedings of the Fifteenth Internaltional Conference on Very Large Databases, Amsterdam, The Netherlands, August 22–25, 1989*, pages 411–422. Morgan Kaufmann Publishers, 1989. (also as Rapport Technique 26-89, GIP Altair, France).

Liskov et al. 77: B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. "Abstraction mechanisms in CLU". *Communications of the ACM*, 20(8):564–576, 1977.

Lockemann, Schmidt 87: P. Lockemann and J.W. Schmidt, (eds.). *Datenbank-Handbuch.* Springer-Verlag, 1987. (in German).

Lorie, Wade 79: R.A. Lorie and B.W. Wade. "The compilation of a high level data language". Research Report RR RJ 2589, IBM Almaden Research Center, San Jose, California, 1979.

MacLane 71: S. MacLane. *Categories for the working mathematician*, Volume 5, *Graduate Texts in Mathematics.* Springer-Verlag, 1971.

Matthes, Schmidt 92: F. Matthes and J.W. Schmidt. "Bulk Types: Built-In or Add-On?". In: *Proceedings of the Third International Workshop on Database Programming Languages, Nafplion, Greece, August 27–30, 1991*, pages 33–54. Morgan Kaufmann Publishers, 1992. (also appeared as Fide Technical Report FIDE/91/20, Department of Computing Science, University of Glasgow).

Matthes 91: F. Matthes. "P-Quest: Installation and User Manual". Universität Hamburg, Fachbereich Informatik, Internal document, DBIS Tycoon Report 102-92, 1991.

Matthes 92a: F. Matthes. *Generische Datenbankprogrammierung: Sprachliche und architektonische Grundlangen.* Dissertation zur Erlangung des Doktorgrades der Naturwissenschaften, Fachbereich Informatik, Universität Hamburg, Germany, 1992. (in German).

Matthes 92b: F. Matthes. "Preliminary Definition of the Tycoon Language TL". Universität Hamburg, Fachbereich Informatik, Internal document, DBIS Tycoon Report 062-92, 1992.

Matthes 93: F. Matthes. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung.* Springer-Verlag, 1993. (in German).

Mauny 91: M. Mauny. "Functional Programming using CAML". Technical Report, INRIA, France, 1991.

McCarthey et al. 65: J. McCarthey, P.W. Abrahams, and et.al. Edwards, D.J. *LISP 1.5 Programmer's Manual.* Massachusetts Institute of Technology, 2nd edition, 1965.

Mehlhorn, Näher 92: K. Mehlhorn and S. Näher. "LEDA – A Library of Efficient Data Types and Algorithms". Technical Report, Max-Planck-Institut für Informatik, Saarbrücken, 1992.

Melton 93: "(ISO/ANSI Working Draft) Database Language SQL3; X3H2-93-359R/MUN-003". 1993.

Meyer 90: B. Meyer. *Introduction to the Theory of Programming Languages.* Prentice Hall International Series in Computer Science. Prentice Hall, 1990.

Meyer 92: B. Meyer. *Eiffel the Language.* Prentice Hall Object-Oriented Series. Prentice Hall, 1992.

ModISO 91: IOS/IEC JTC1/SC22/WG13. *Interim Version of the 4th Draft Modula-2 Standard*, 1991.

Moggi 89: E. Moggi. "Computational lambda-calculus and monads". In: *Proceedings of the Fourth Symposium on Logic in Computer Science, Pacific Groove, California, June 5–8, 1989*, pages 14–23. IEEE Computer Society Press, 1989.

Morrison et al. 89: R. Morrison, A. Brown, R.C.H. Connor, and A. Dearle. "The Napier88 Reference Manual". Technical Report PPRR-77-89, Department of Computing Science, University of Glasgow, 1989.

Müller 91: R. Müller. "Sprachprozessoren und Objektspeicher: Schnittstellenentwurf und -implementierung". Diplomarbeit, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt/Main, Germany, 1991. (in German).

Nelson 91: G. Nelson, (ed.). *Systems programming with Modula-3.* Prentice Hall series in innovative technology. Prentice Hall, 1991.

Niederée et al. 92: C. Niederée, S. Müßig, and F. Matthes. "P-Quest User Manual". Universität Hamburg, Fachbereich Informatik, Internal document, DBIS Tycoon Report 102-92, 1992.

Niederée 92: C. Niederée. "Generische Dienste für datenintensive Anwendungen: Iterationsabstraktion, Integritätsüberwachung, Fehlererholung". Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, 1992. (in German).

Ohori et al. 89: A. Ohori, P. Buneman, and V. Brezeau-Tannen. "Database Programming in Machiavelli - a Polymorphic Language with Static Type Inference". In: J. Clifford, B. Lindsay, and D. Maier, (eds.), *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Portland, Oregon, May 31 – June 2, 1989*, pages 46–57. Association for Computing Machinery, 1989. SIGMOD RECORD Volume 18, Number 2, June.

Oracle 91: Oracle Coorporation. *PL/SQL User's Guide and Reference, Version 1.0*, 1991.

Paulson 91: L.C. Paulson. *ML for the working programmer.* Cambridge University Press, 1991.

Peyton-Jones 87: S.L. Peyton-Jones. *The Implementation of Funtional Programming Languages.* Prentice Hall, 1987.

Reimer, Diener 83: M. Reimer and A. Diener. "The Modula/R Compiler for Lilith". LIDAS Memo 051-83, Department Informatik, ETH Zürich, Switzerland, 1983.

Reynolds 83: J.C. Reynolds. "Types, abstractions, and parametric polymorphism". In: R.E.A. Mason, (ed.), *Information Processing*, pages 513–523. North-Holland Publishing Company, 1983.

Ross 92: P. Ross. *Bulk Data Types: A Theoretical Approach.* PhD thesis, Department of Computer Science, The Hebrew University of Jerusalem, Israel, 1992.

Rovner et al. 85: P. Rovner, R. Levin, and J. Wick. "On Extending Modula-2 for Building Large, Integrated Systems". Digital Systems Research Center Report No. 3, Digital Systems Research Center, Palo Alto, California, 1985.

Schmidt, Matthes 92: J.W. Schmidt and F. Matthes. "The Database Programming Language DBPL – Rational and Report". Technical Report FBI-HH-B-158/92, Fachbereich Informatik, Universität Hamburg, Germany, 1992.

Schmidt 77: J. Schmidt. "Some High Level Language Constructs for Data of Type Relation". *ACM Transactions on Database Systems*, 2(3):247–261, 1977.

Schröder 93: G. Schröder. "Syntaktische Erweiterbarkeit von Programmiersprachen unter Benennungs-, Bindungs- und Typisierungsinvarianzen". Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, 1993. (in German).

Smedt et al. 93: P. De Smedt, S. Ceri, M.-A. Neimat, M.-C. Shan, and R. Ahmed. "Recursive Functions in Iris". In: F. Golshani, (ed.), *Proceedings of the Ninth International Conference on Data Engineering, April 19–23, 1993, Vienna, Austria*, pages 145–154. The Institute of Electrical and Electronics Engineers, Inc., 1993.

Smith et al. 81: J.M Smith, S. Fox, and T. Landers. "Reference Manual for ADAPLEX". Technical Report CCA-81-02, Computer Corporation of America, Cambridge, MA, 1981.

Smith et al. 83: J.M Smith, S. Fox, and T. Landers. "ADAPLEX: Rationale and Reference Manual". Technical Report CCA-83-08, Computer Corporation of America, Cambridge, MA, 1983.

Stemple, Sheard 91: D. Stemple and T. Sheard. "A Recursive Base for Database Programming Primitives". In: *Proceedings of the First International East/West Database Workshop on Next Generation Information System Technology, Kiev, USSR, October 9 – 12, 1990*, Volume 504, *Lecture Notes in Computer Science*, 1991.

Stonebraker et al. 76: M. Stonebraker, E. Wong, P. Kreps, and G.D. Held. "The Design and implementation of INGRES". *ACM Transactions on Database Systems*, 1(3):189–222, 1976.

Stonebraker, Rowe 86: M. Stonebraker and L.A. Rowe. "The Design of POSTGRES". In: C. Zaniolo, (ed.), *Proceedings of the International Conference on Management of Data, Washington, D.C., May 28–30, 1986*, pages 340–355. Association for Computing Machinery, 1986.

Trinder, Wadler 89: P.W. Trinder and P. Wadler. "Improving List Comprehension Database Queries". In: *Proceedings of the Fourth IEEE Region 10 International Conference - Information Technologies for the 90's E^2C^2; Energy, Electronics, Computers, Communications, Bombay, India, November 22 – 24, 1989*, pages 186–192. The Institute of Electrical and Electronics Engineers, Inc., 1989.

Trinder 89: P. Trinder. *A Functional Database.* PhD thesis, Oxford University, 1989.

Trinder 92: P. Trinder. "Comprehensions, a Query Notation for DBPLs". In: P. Kanellakis and J.W. Schmidt, (eds.), *Proceedings of the Third International Workshop on Database Programming Languages, Nafplion, Greece, August 27–30, 1991*, pages 55–68. Morgan Kaufmann Publishers, 1992.

Turner 82: D.A. Turner. "Recursion Equations as a programming language". In: J. Darlington, P. Henderson, and D. Turner, (eds.), *Functional Programming and its Applications*, pages 1–28. Cambridge University Press, 1982.

Turner 85: D.A. Turner. "Miranda: A non-strict functional language with polymorphic types". In: *Proceedings of the 1985 IFIP Conference on Functional Programming Languages* *and Computer Architecture*, Lecture Notes in Computer Science, pages 1–16. Springer-Verlag, 1985.

Turner 87: D.A. Turner. *Miranda System Manual*, 1987.

Ullman 88a: J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, Volume I. Computer Science Press, 1988.

Ullman 88b: J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, Volume II. Computer Science Press, 1988.

Vuillemin 74: J. Vuillemin. "The Mechanical Evaluation of Expressions". *Journal of Computer and System Science*, 9(3):332–354, 1974.

Wadler 87: P. Wadler. "List Comprehensions". In: Simon L. Peyton Jones, (ed.), *The Implementation of Functional Programming Languages (chapter 7)*, pages 127–138. Prentice Hall, 1987.

Wadler 89: P. Wadler. "Theorems for free!". In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture, Imperial College, London, England, September 11-13, 1989*, pages 347–359. Association for Computing Machinery, 1989.

Wadler 90: P. Wadler. "Deforestation: Transforming Programs to eleminate trees". *Theoretical Computer Science*, 73:231–248, 1990.

Wadler 92: P. Wadler. "The essence of functional programming". In: *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, January 19–22, 1992*, pages 1–13. Association for Computing Machinery, 1992.

Wadsworth 71: C. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus.* PhD thesis, Oxford University, 1971.

Wasserman et al. 81: A.L. Wasserman, D.D. Sheretz, and M.L. Kerstin. "Revised Report on the Programming Language PLAIN". *ACM SIGPLAN Notices*, 16(5):59–80, 1981.

Watt, Trinder 91: D.A. Watt and P. Trinder. "Towards a Theory of Bulk Types". Fide Technical Report FIDE/91/26, Department of Computing Science, University of Glasgow, 1991.

Wirth 71: N. Wirth. "The programming language PASCAL". *Acta Informatica*, (1):35–63, 1971.

Wirth 87: N. Wirth. "The Programming Language Oberon". Technical report, Department Informatik, ETH Zürich, Switzerland, 1987.

Wong 92: L. Wong. "A Conservative Property of a Nested Relational Query Language". Technical Report MS-CIS-92-59, University of Pennsylvania, Computer and Information Science Department, Pennsylvania, 1992.

Zermelo 08: E. Zermelo. "Untersuchungen über die Grundlagen der Mengenlehre". *Mathematische Annalen*, 65:261–281, 1908. (article in German; a translation may be found in [Heijenoord 67]).

Zloof 77: M.M. Zloof. "Query-By-Example: A database language". *IBM System Journal*, 16(4):324–343, 1977.

Zook et al. 77: W. Zook, K. Youssefi, N. Whyte, P. Rubenstein, P. Kreps, G. Held, J. Ford, R. Berman, and E. Allman. *INGRES Reference Manual*, 1977.

Danksagung

Mein besonderer Dank gilt Herrn Gerald Schröder für die intensive, stete Betreuung, die zahlreichen inhaltlichen Diskussionen und vielen Anregungen, die zur Entstehung dieser Arbeit beigetragen haben. Herrn Dr. Florian Matthes möchte ich für die inhaltlich sehr wertvollen Diskussionen danken. Herrn Prof. Dr. Joachim W. Schmidt danke ich für die Ratschläge zu Verbesserung der Arbeit sowie für die exzellente Arbeitsumgebung und Herrn Prof. Dr. Heinz Züllinghoven für die kritische Durchsicht. Mein Dank gilt auch Herrn Prof. Dr. Catriel Beeri, der während seines Aufenthaltes in Hamburg entscheidende Impulse für die theoretischen Betrachtungen in dieser Arbeit gab. Frau Helen Brodie möchte ich für die tatkräftige Unterstützung im administrativem Bereich danken.

Weiterhin möchte ich Claudia für die Unterstützung während der Diplomarbeit, Rosi und Herrn Böttcher für die vielen Kommata und Korrekturen, Herrn Sven Müßig für die Durchsicht des Literaturverzeichnisses, Herrn Andreas Rudloff für die Hilfe bei vielen technischen Fragen und Diskussionen über rekursive Anfragen, Herrn Andreas Gawecki für interessante Diskussionen, Herrn Thomas Sidow für die Instandhaltung des Druck- und Rechenbetriebes danken.

Eine Reihe von interessanten Impulsen für meine Arbeit stammen auch aus der intensiven Zusammenarbeit des Arbeitsbereichs DBIS mit anderen Projektpartnern und seinen zahlreichen Besuchern.

Erklärung

Ich erkläre hiermit, die vorliegende Arbeit selbständig durchgeführt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Hamburg, den 10. März 1994

Dominic M. Juhász