Processing IQL Queries in the AutoMed toolkit Version 1.2

Lucas Zamboulis, Sandeep Mittal, Edgar Jasper,
Hao Fan, Alexandra Poulovassilis
School of Computer Science and Information Systems,
Birkbeck, University of London

July 29, 2008

Contents

1 Introduction

This technical report gives an outline of the IQL query language used within the AutoMed heterogeneous data integration system, and describes query processing in AutoMed. This report aims to serve as a guide to the query processing components of the AutoMed toolkit.

Report outline: Section 2 describes the aspects of IQL necessary for this report. Section 3 presents the AutoMed Query Processor and its components. Section 4 illustrates how data querying can be performed. Section 5 discusses examples developed to demonstrate and test AutoMed. Section 6 lists the changes made to AutoMed's query processor since Version 1.0, and Section 7 presents areas of future work.

2 IQL

The AutoMed Intermediate Query Language (IQL) [6] is a typed, comprehensions-based functional query language. Such languages subsume query languages such as SQL-92 and OQL in expressiveness [1]. Its purpose is to provide a common query language that queries written in various high level query languages (*e.g.* SQL, XQuery, OQL) can be translated into and out of. For the complete syntax of IQL Version 1.2 see Appendix B. For a tutorial on IQL, see [7].

2.1 Datatypes, variables & functions

IQL currently supports integer and float numbers (e.g. `5, 3.46`), strings (enclosed in single quotes, e.g. `'AutoMed'`), datetime objects (e.g. `dt '2005-05-15 23:32:45'`) and the boolean values `True` and `False`. It also supports tuples (e.g. {`5, 'AutoMed', dt '2005-05-15 23:32:45'`}), and enumerated lists (e.g. `[5, 5, 6, 7, 7, 7]`), bags (e.g. `B[5, 5, 6, 7, 7, 7]`) and sets (e.g. `S[5, 6, 7]`) of homogeneous values (i.e. values of the same type). In principle, the tuple, list, bag and set value constructors can be arbitrarily nested. However, to reduce parser complexity, a limit to the levels of nesting has been set, which is currently 40.

Variables are represented by identifiers starting with a lowercase character. Anonymous functions may be defined using *lambda abstractions*. For example, the following function, when applied to a triple of values, adds the three components the triple:

$$lambda\ \ \{x, y, z\}\ \ (x\ +\ (y\ + z))$$

$\{x, y, z\}$ is a *pattern* (i.e. an expression comprising values and tuple constructors only) and is the formal parameter of the anonymous function, while $(x + (y + z))$ is the *body* of the lambda abstraction[1]. Thus, for example, the expression $(lambda\ \ \{x, y, z\}\ \ (x\ +\ (y\ + z)))\ \ \{5, 6, 7\}$ evaluates to 18.

IQL has built-in support for the common boolean, arithmetic, relational and collection operators. Appendix A lists the current set of built-in functions, which is easily extensible (see Section 3.8). The binary IQL built-in functions are supported in both infix and prefix form, with the latter being enclosed in

[1] Note that the IQL parser expects the body of a lambda abstraction to be enclosed within parentheses. See [5] for more information on lambda abstractions and functional languages.

brackets e.g. the list append operator is `++` when used infix and `(++)` when used prefix. In prefix form, such operators may be applied only to a partial complement of their arguments (i.e. to 0 or 1 argument only) e.g. `(+) 5` is a function that adds 5 to its argument, so `((+) 5) 6` returns 11 (and is equivalent to `(+) 5 6`).

2.2 Higher-level constructs

IQL also supports *let expressions* and *list, bag* and *set comprehensions*. These do not add additional expressiveness to the language but are 'syntactic sugar', allowing queries that are easy to write and read; they also facilitate the translation between IQL and various high-level query languages. Furthermore, several optimisation techniques can be applied to comprehensions, as discussed in Section 3.6.

let expressions assign an expression to a variable and this variable can then be used within another expression. In particular, in *let* v *equal* e_1 *in* e_2, expression e_1 is assigned to variable v, which appears within expression e_2.

List comprehensions are of the form $[h|q_1; \ldots q_n]$, where h is an expression termed the *head* and $q_1, \ldots, q_n$ are *qualifiers*, with $n \geq 0$. A qualifier may be either a *filter* or a *generator*. Generators are of the form $p \leftarrow e$ and iterate a *pattern* p over a list-valued expression e. Filters are boolean-valued expressions that act as filters on the variable instantiations generated by the generators of the comprehension.

The following is an example of a list comprehension:

$$[\{x, y\}|x \leftarrow [1, 2, 3]; y \leftarrow [\text{'}a\text{'}, \text{'}b\text{'}]; x > 1]$$

This undertakes a Cartesian product of the two lists followed by a selection; the result is

$$[\{2,\text{'}a\text{'}\}, \{2,\text{'}b\text{'}\}, \{3,\text{'}a\text{'}\}, \{3,\text{'}b\text{'}\}]$$

Set comprehensions have similar syntax, except starting with the symbol $S[$ and expecting their generators to iterate over set-valued expressions. Similarly, bag comprehensions start with the symbol $B[$ and expect bag-valued expressions in their generators.

List comprehensions may be translated into simple function applications and lambda abstractions as follows:

$$\begin{array}{ll} [e|p \leftarrow s; Q] & \Longrightarrow flatmap\ \ (lambda\ \ p\ \ [e|Q])\ \ s \\ [e|e'; Q] & \Longrightarrow if\ \ e'\ \ [e|Q]\ \ [\,] \\ [e|] & \Longrightarrow [e] \end{array}$$

Here, the if function takes three arguments and returns its second argument if its first argument is true, its third argument if its first argument is false, and `Null` if its first argument is `Null`. The $flatmap$ function, operating on lists, is defined as follows:

$$\begin{array}{lcl} flatmap\ \ f\ \ [\,] & = & [\,] \\ flatmap\ \ f\ \ (Cons\ \ x\ \ xs) & = & (f\ \ x)\ \ +\!+\ \ (flatmap\ \ f\ \ xs) \end{array}$$

where [ ] is the empty list and $Cons\ x\ xs$ represents a list of which x is the head and xs the rest of the list.

We note that $flatmap$ is an overloaded function that operates also on sets and bags — see below. The ++ operator is similarly overloaded: it appends two lists together, and undertakes set union and bag union on sets and bags, respectively.

Set comprehensions may be translated into simple function applications and lambda abstractions as follows:

$$\begin{array}{ll} S[e|p \leftarrow s; Q] & \Longrightarrow flatmap\ \ (lambda\ \ p\ \ S[e|Q])\ \ s \\ S[e|e'; Q] & \Longrightarrow if\ \ e'\ \ S[e|Q]\ \ SNil \\ S[e|] & \Longrightarrow (SCons\ \ e\ \ SNil) \end{array}$$

Here, $SNil$ represents the empty set and $SCons\ x\ xs$ represents a set of which x is an arbitrary member and xs is the rest of the set. The $flatmap$ function, operating on sets, is defined as follows:

$$\begin{array}{lcl} flatmap\ \ f\ \ SNil & = & SNil \\ flatmap\ \ f\ \ (SCons\ \ x\ \ xs) & = & (f\ \ x)\ \ ++\ \ (flatmap\ \ f\ \ xs) \end{array}$$

Bag comprehensions may be translated into simple function applications and lambda abstractions as follows:

$$\begin{array}{ll} B[e|p \leftarrow s; Q] & \Longrightarrow flatmap\ (lambda\ p\ B[e|Q])\ s \\ B[e|e'; Q] & \Longrightarrow if\ \ e'\ \ B[e|Q]\ \ BNil \\ B[e|] & \Longrightarrow (BCons\ \ e\ \ BNil) \end{array}$$

Here, $BNil$ represents the empty bag and $BCons\ x\ xs$ represents a bag of which x is an arbitrary member and xs is the rest of the bag. The $flatmap$ function, operating on bags, is defined as follows:

$$\begin{array}{lcl} flatmap\ \ f\ \ BNil & = & BNil \\ flatmap\ \ f\ \ (BCons\ \ x\ \ xs) & = & (f\ \ x)\ \ ++\ \ (flatmap\ \ f\ \ xs) \end{array}$$

IQL supports *variable unification* within comprehensions, *e.g.* the following query evaluates to [{3,5}]:

$$[\{a, c\}|\{a, b\} \leftarrow [\{1, 2\}, \{3, 4\}];\ \{b, c\} \leftarrow [\{4, 5\}, \{6, 7\}]]$$

To achieve this, the input query is scanned and any variable encountered for the first time is left as is, whereas the rest are renamed using the naming scheme $\$vu\_n\_i\_j$. The leading $ character means this is a system variable, n is the original name of the variable, i is an integer identifying the comprehension in which n occurs, and j is an integer counting the number of times n has appeared within the comprehension identified by i (i is useful for nested comprehensions). Any necessary equality qualifiers between these system variables are then inserted at the end of the comprehension's qualifiers. The above query is therefore translated into the following query:

$$[\{a, c\}|\{a, b\} \leftarrow [\{1, 2\}, \{3, 4\}]; \{\$vu\_b\_1\_1, c\} \leftarrow [\{4, 5\}, \{6, 7\}]; (=)\ b\ \$vu\_b\_1\_1)]$$

2.3 Typing

In common with many functional languages, IQL supports ML-style parametric polymorphism of types and functions, as well as overloading of function names. This is the same type system as the PFL functional database language [8], excluding at present subtyping and inheritance, although these aspects could readily be added to IQL if required in the future. However, the current AutoMed Query Processor does not include a type checker (it is up to users to ensure the type-correctness of their IQL queries!) although such a component could readily be added, leveraging essentially the PFL type checker described in [8].

The syntax of IQL types is as follows, where σ ranges over type expressions, α over type variables and κ^n over n-ary type constructors (such as the list, set, bag and product type constructors — see below):

$$\sigma ::= \alpha \mid \sigma_1 \rightarrow \sigma_2 \mid \kappa^n\ \sigma_1\ \ldots\ \sigma_n$$

Types of the form $\sigma_1 \rightarrow \sigma_2$ are *function types*; the operator $\rightarrow$ is right-associative, so that $\sigma_1 \rightarrow \sigma_2 \rightarrow \sigma_3$ and $\sigma_1 \rightarrow (\sigma_2 \rightarrow \sigma_3)$ are synonymous.

Types of the form κ^0 are *atomic types* (so the set of 0-ary type constructors and the set of atomic types are identical). IQL's atomic types are $Integer$, $Float$, $String$, $Boolean$ and $DateTime$.

Types of the form $\kappa^n\ \sigma_1\ \ldots\ \sigma_n$, where $n > 0$, are *structured types*. IQL's type constructors for structured types are as follows:

$$\begin{array}{lcl}
List & : & Type \rightarrow Type \\
Set & : & Type \rightarrow Type \\
Bag & : & Type \rightarrow Type \\
Product1 & : & Type \rightarrow Type \\
Product2 & : & Type \rightarrow Type \rightarrow Type \\
Product3 & : & Type \rightarrow Type \rightarrow Type \rightarrow Type \\
\ldots &&
\end{array}$$

These declarations can be read as stating that a type constructor takes one or more types as arguments and constructs a new type.

Both structured and atomic types are populated by *value constructors* which return values of that type when applied to appropriate arguments. The 0-ary value constructors populating the $Integer$, $Float$, $String$, $Boolean$ and $DateTime$ types are respectively the integer, float, string, boolean and date-time values. Also available are polymorphic constructors for empty lists, sets and bags, for lists/sets/bags consisting of a head and tail, and for tuples:

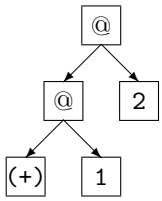
$$\begin{array}{lcl}
Nil & :: & List\ a \\
Cons & :: & a \rightarrow (List\ a) \rightarrow (List\ a) \\
SNil & :: & Set\ a \\
SCons & :: & a \rightarrow (Set\ a) \rightarrow (Set\ a) \\
BNil & :: & Bag\ a \\
BCons & :: & a \rightarrow (Bag\ a) \rightarrow (Bag\ a) \\
Tuple1 & :: & a \rightarrow (Product1\ a) \\
Tuple2 & :: & a \rightarrow b \rightarrow (Product2\ a\ b) \\
Tuple3 & :: & a \rightarrow b \rightarrow c \rightarrow (Product3\ a\ b\ c) \\
\ldots &&
\end{array}$$

In the above declarations, a, b, and c are type variables that can be instantiated by any type. Generally, in IQL identifiers starting with a lower-case letter are variables while identifiers starting with an upper-case letter are constructors.

Some syntactic sugar is supported for structured values in that, for any expressions $e_1, \ldots, e_n$, an n-tuple $\{e_1, \ldots, e_n\}$ is synonymous with $Tuple_n\ e_1\ \ldots e_n$, an enumerated list $[e_1, \ldots, e_n]$ is synonymous with $Cons\ e_1\ (\ldots\ Cons\ (e_n\ [])\ldots)$, and $[]$ is synonymous with Nil. Likewise for enumerated bags and sets.

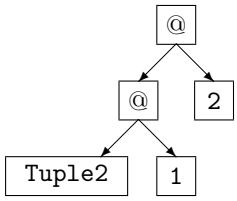
2.4 Abstract representation

IQL queries are represented internally as a full binary *abstract syntax tree*. All non-leaf cells are either apply cells (@) or lambda cells (λ). An apply cell represents the left child being applied to the right child. For example, the following tree represents the expression `1 + 2`:

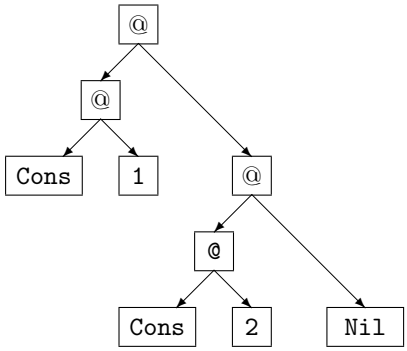


Note that all functions are represented internally in prefix form, and can be partially applied to less than a full complement of their arguments. So the result of any apply cell can be considered to be the result of its left child applied to its right child.

Leaf cells may be constants, variables, constructors or function names. For example, the following represents the pair `{1,2}`:

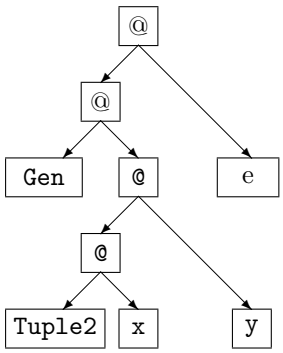


and the following the list `[1,2]`:

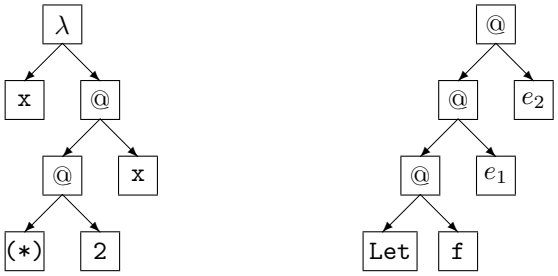


The same principle applies for the representation of sets and bags. The only difference is that for sets, the constructors `SCons` and `SNil` are used, instead of `Cons` and `Nil`, and for bags the constructors `BCons` and `BNil`.

The following figure illustrates the abstract syntax tree for the generator $\{\mathtt{x},\mathtt{y}\} \leftarrow \mathtt{e}$ of some comprehension:



The following figures illustrate the abstract syntax trees for the lambda abstraction *lambda* x $(2 * x)$ and for the let expression *let* f *equal* e_1 *in* e_2, respectively:



The above abstract representation for IQL queries is implemented as the `ASG` class — *abstract syntax graph*[2]. The purpose of this class is to hide the implementation details and provide useful methods to work with. It is intended that `ASG` objects be the standard way to store IQL queries and that the various elements of the AutoMed query processing software should act upon `ASG` representations of queries. The following code snippet illustrates the creation of an `ASG` object:

```
String query = "<<person,pname>>";
ASG q = new ASG(query);
```

Note the use of an AutoMed scheme in the above query, delimited by double chevrons. Evaluating any scheme in IQL results in a collection of values of the appropriate type. For example, if `person` is a relational table with a single key attribute `pid` of IQL type `Integer` and a non-key attribute `pname` of IQL type

[2]Note that in general we are dealing with an abstract syntax graph rather than an abstract syntax tree. Although parsing an IQL query results in a tree, it is possible that, when the tree is being evaluated, some substructures become shared. Rather than duplicate these substructures to retain a tree structure, the original abstract syntax tree may become a directed acyclic graph.

`String`, then the above query would return a list of pairs of the form {`i,n`} where `i` is an integer and `n` is a string[3].

A scheme `s` is expanded internally into the representation `:P:S:M:C:s:T`, where:

- `S` is the name of the schema `s` belongs to,
- `M` is the name of the modelling language in which `S` is defined, e.g. `sql`, `xmldss`.
- `C` states what kind of modelling construct `s` is, e.g. `table`, `column` (from the `sql` modelling language), `element`, `attribute` (from the `xmldss` modelling language).
- `T` is the type of the scheme `s`, and
- `P` identifies the AutoMed repository where `S` and `s` are stored (that is useful in a peer-to-peer setting, for example, where there may be multiple interoperating AutoMed installations).

For example, for the above scheme `<<person,pname>>`, its fully expanded representation could be

```
:Peer5:s3:sql:column:<<person,pname>>:'List (Product2 Integer String)'
```

where `s3` is the name of the schema within the `Peer5` repository that this scheme belongs to.

Note that it is possible for some or all of this additional information not to be available to AutoMed in different contexts of usage. Thus, any or all of P, S, M, C or T may be missing, though the colons remain in the internal representation e.g. when first parsed within a query, the scheme `<<person,pname>>` is initially represented by `:::::<<person,pname>>:`.

3 Query Processing in AutoMed

This section discusses query processing in AutoMed. Section 3.1 discusses how the AutoMed Query Processor (AQP) handles input queries that are expressed in a high-level query language, rather than IQL. Section 3.2 gives a brief introduction to the AQP. Section 3.3 describes how users can define their own data source integration semantics. Section 3.4 describes the configuration of the AQP and of its components. Sections 3.5–3.8 describe each of the AQP components. Section 3.9 discusses extensibility of the AQP components. Section 3.10 discusses logging. Section 3.11 discusses testing of the AutoMed code. Section 3.12 discusses memory and performance issues raised during the development of the AQP.

[3]Generally, the extent of a scheme may be a list, bag or set. Currently however all the AutoMed wrappers return list-valued extents. This may change in the near future.

3.1 High-Level Query Translation

Query processing in AutoMed is currently tightly coupled with the IQL query language. To enable the use of high-level query languages such as SQL and XQuery with AutoMed, the AutoMed `Translator` component can be used to translate a query expressed in a high-level query language into the equivalent IQL query, which can then be submitted to the AQP.

The `Translator` interface specifies two methods, `getIQL()`, which translates the input high-level query into an IQL query, and `translateResult(ASG)`, which takes an IQL result and translates it back to the high-level language.

Currently, the AQP supports a subset of SQL (composed from possibly nested Select-Project-Join-Union queries, aggregation functions, and GROUP BY) and a subset of XQuery (FLWR queries without nesting) for global schemas whose high-level data model is the relational data model or XML Schema/XMLDSS [9], respectively.

3.2 AQP

Figure 1 illustrates the AQP components, namely the `QueryReformulator`, the `QueryOptimiser`, the `QueryAnnotator` and the `QueryEvaluator`.

The AQP is used to evaluate queries submitted to a virtual schema against a set of data sources. It does this by coordinating the above components but is agnostic of the internals of these components. Each of these components may have more than one implementation. The implementation to be used for each component is specified using the AQP's `QueryProcessorConfiguration` component. This configures the AQP as well as the components it contains and is discussed in more detail in Section 3.4. `QueryProcessorConfiguration` provides a default implementation for each of the components used by the AQP.

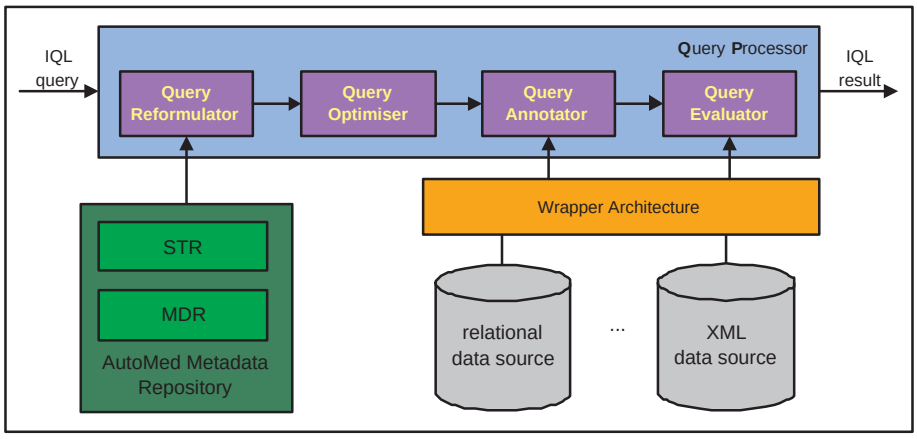


Figure 1: The AutoMed Query Processor Architecture.

The `QueryProcessor` class implements the AutoMed Query Processor. An instance of this class can be created by calling the default constructor. The AQP does not require any initialisation and any number of AQP instances can be created, for example so as to parallelise the processing of multiple queries.

When a query needs to be evaluated, the `process` method of the `Query Processor` class is called with the following parameters:

- the query, in the form of an ASG
- the target schema i.e. the virtual schema against which the query is to be evaluated
- the array of data source schemas with respect to which the query will be reformulated prior to evaluation
- the required data integration semantics: can be one of "choose", "(++)", "intersect", "union", an expression, or null — in which case the default is "(++)"; see Section 3.3 for more details
- the AQP Configuration — see Section 3.4 for more details.

The input query is first reformulated using the `QueryReformulator` component to an equivalent query that only contains data source constructs. The `QueryOptimiser` component then optimises the reformulated query and the `QueryAnnotator` component inserts AutoMed Wrapper objects within the optimised query. The `QueryEvaluator` is then used to evaluate the annotated query. The result of each of these stages of query processing, along with more fine-grained debugging information, is logged.

Note that it is possible to specify the source and target schema parameters as `null`, in case a user wants to use the `QueryProcessor` stand-alone and not within a data integration setting. If so, the query reformulation and query annotation steps are not performed and the query is evaluated without any reference to any data sources.

3.3 Integration Semantics

Integration semantics define the way data from different data sources are combined to form the extent of global schema constructs. Currently, only user-defined integration semantics are supported and, if not provided, the default "append" semantics are used. It is envisaged, however, that in the future the default behaviour will be to inspect the semantics carried by the `id` transformations themselves if no user-defined integration semantics are provided.

The user has the following ways of specifying the integration semantics to an instance of AQP:

(i) Shorthand: The following four integration operators can be used as a shorthand notation. The AQP combines results from all the data sources using the same operator, in the order the data sources are specified in the input data source schemas array:

Choose: values coming from just one data source are returned; which data source is chosen depends on the `ChooseOptimiser` (Section 3.6) or, if this optimiser is not invoked, on a non-deterministic choice by the *choose* IQL function (see Appendix A).

Append: all values returned by the data sources are returned, combined using the ++ operator.

Intersect: the values returned from the data sources are combined using the *intersect* operator (see Appendix A).

Union: the values returned from the data sources are combined using the duplicate-eliminating *union* operator (see Appendix A).

(ii) Verbose: The user can specify an expression containing the names of the schemas of the data sources, composed using any of the operators listed above or indeed any of the other operators on collections, such as $--$ (see Appendix A). The schema name of each data source must be prefixed by a $ symbol in such an expression e.g. `($S1 intersect $S2) ++ ($S1 -- $S3)`

3.4 Configuration

The `QueryProcessorConfiguration` component allows the user to set a number of parameters that configure the setup of the AQP and of its components. These parameters either specify the implementation of each component of the AQP (each component in Figure 1 may have more than one implementations) or specify runtime options such as whether the AQP should perform optimisation or not and the reformulation approach it should employ.

The `QueryProcessorConfiguration` component currently supports the following options:

Component implementations: each AQP component may have more than one implementation; the default ones used are the `QueryReformulator`, the `StandardOptimisationProvider`, the `StandardQueryAnnotationProvider`, and the `Evaluator`.

Reformulation approach: the default value is BAV reformulation; the user may modify this option to specify the use of GAV reformulation or LAV (for more on supported reformulation approaches see Section 3.5).

Range semantics: the default for this option is `Range` semantics; the user may modify this option to specify the use of the lower- or upper-bound query only (for more on `Range` semantics see Section 3.5).

Query language: the default query language for the AQP is the IQL query language, but a user may modify this option to enable the use of SQL or XQuery as the input query language.

Function table: the function table contains the IQL functions available to the `Evaluator` component. The default configuration of the AQP uses the `StandardFunctionTable`, which contains the built-in IQL functions described in Appendix A. An AutoMed developer can create his/her own IQL functions, define a new function table that references these functions, and assign this new function table to the AQP, which will add the referenced functions to the list of functions available to the `Evaluator` through the `StandardFunctionTable`.

Optimisation: boolean option that specifies whether the AQP performs optimisation using the `QueryOptimiser` component; default value is true.

Disjoint IDs: setting this boolean option to true specifies that key attribute values arising in the virtual integrated schema from different data sources are non-overlapping and therefore allows the `EmptyJoinOptimiser` component (see below) to remove any such equijoins across data sources.

Type checking: boolean option that specifies whether the AQP performs type checking; currently there is no implementation for the type checking component, and so the default value is false.

3.5 Query Reformulation

The `QueryReformulator` component is able to reformulate queries submitted to a virtual schema against a set of extensional data sources using GAV, LAV or BAV reformulation. This section first presents the generic design and implementation of `QueryReformulator` component, then discusses the implementation of each of the reformulation techniques. For more details on GAV, LAV and BAV reformulation in AutoMed, see [4].

The AutoMed query processing components were implemented having modularity and extensibility in mind. As a result, it is possible for a user to change at runtime the settings and configuration of the AQP. Furthermore, extending the AQP and IQL with new reformulation techniques, IQL functions and query processing components is straightforward.

3.5.1 Design and generic implementation

Similarly to the AQP, it is possible to have more than one `QueryReformulator` instances. However, as discussed below, the data structures used by the `QueryReformulator` maintain caches of their instances, so it is possible for multiple `QueryReformulator` instances to access the same set of view definitions. An instance of the `QueryReformulator` can be obtained using the `QueryReformulationFactory`, which initialises a `QueryReformulator` instance using the default constructor, *i.e.* no initialisation is required while constructing an instance. When a query on a target schema needs to be reformulated, the `reformulate` method is called with the names of the target and data source schemas, the integration semantics and a `QueryProcessorConfiguration` instance.

The `QueryReformulator` component is split into three layers in order to be able to provide all three types of reformulation and at the same time be ready for future extensions, *e.g.* adding other types of LAV reformulation algorithms. All three currently implemented reformulation techniques produce a view map that contains view definitions of the target schema constructs in terms of the data source schema constructs. This map can then be used to unfold any virtual schema constructs occurring in a user query.

Layer 1 - `AtomicViewMap`

The first layer abstracts and stores in memory the metadata contained in the transformation pathways between the target schema and the data source schemas so as to reduce costly interactions with the AutoMed repository. For each pathway, six data structures are created, each storing a different type of transformation for a single pathway — *e.g.* twelve such data structures will be created for given a target schema and two data source schemas.

Each such data structure is essentially a map, where the key for each row is the `SchemeInfo` object corresponding to a certain schema object of the target schema. The `SchemeInfo` object in this case also contains the AutoMed repository object identifier (OID) for that construct, which does not change when a transformation is defined from one schema to another, unless it is an `ident` transformation. The value corresponding to a key is the query supplied with the transformation (for `add`/`delete`/`extend`/ `contract` transformations), the renamed construct (for `rename` transformations) or another `SchemeInfo` object (for `ident` transformations).

This functionality is provided by a single class, `AtomicViewMap`, meaning that each of these data structures is an instance of this class. An instance of this class for a given target and data source schema will be likely be used multiple times during the lifetime of an instance of the AQP, so the `AtomicViewMap` class maintains a cache of its own instances.

Layer 2 - `CompositeViewMap`

The second layer uses the data structures of the first layer to produce the view definition of each target schema construct in terms of the schema constructs of a single data source[4]. This level distinguishes between the three currently available reformulation techniques by providing three components, one for performing GAV reformulation, one for performing LAV reformulation using the inverse rules technique, and one for performing BAV reformulation (note again that these components operate on a target schema and a *single* data source). At this point, the reason for the first layer to store the transformations of a single pathway in six different data structures becomes clear: GAV reformulation does not consider `add` and `extend` transformations, LAV reformulation does not consider `delete` and `contract` transformations and BAV reformulation considers all six types of transformations. Storing different types of transformations in different data structures allows the second layer to select and consider only the types of transformations that are of relevance to the reformulation technique at hand.

The three aforementioned components of the second layer are the `GAVViewMap`, the `LAVInvRulesViewMap` and the `BAVViewMap`; these components are subclasses of the `CompositeViewMap` component, which abstracts the common functionality of these components. Note also that, just like the `AtomicViewMap` component, the `CompositeViewMap` component also maintains a cache of the instances of its three subclasses.

Layer 3 - `QueryReformulator`

The third layer consists of a single component, the `QueryReformulator`. It employs as many instances of one of the three components of the second layer as there are data source schemas, and produces a map containing the view definition of each target schema construct in terms of the schema constructs of the data sources. This layer does not maintain a cache of its instances for two reasons. First, due to caching at the two lower levels, producing its own set

[4]Note that this applies not only for GAV reformulation, but also for LAV and BAV reformulation — see Section 3.5.3 and [4] for details.

of view definitions is cheap, and, second, this simplifies the process of changing the user-defined integration semantics at runtime.

3.5.2 GAV Query Reformulation

When using GAV as the reformulation technique, the AQP uses only those portions of BAV pathways that define target schema constructs in terms of data source constructs, namely `delete`, `contract`, `rename` and `ident` transformations (assuming a target to data source traversal of the schema transformation pathways). The `add` and `extend` transformations are not considered, as these cannot contribute to the view definitions of the virtual schema constructs in terms of data source constructs (they are `delete` and `contract` transformations respectively if traversed in the direction from a data source schema to the target schema).

The view definition for each construct of a virtual schema is derivable from the BAV pathways using the GAV view generation algorithm described in [4]. Each `GAVViewMap` (a specialisation of `CompositeViewMap` discussed above) provides definitions of target schema constructs as views over the constructs of a single data source schema. The GAV reformulation algorithm uses these view definitions to unfold the query over a target schema in terms of constructs of the data source schemas. Section 3.5.5 provides more details on how each type of primitive transformation is handled by `CompositeViewMap`.

3.5.3 LAV Query Reformulation

When using LAV as the reformulation technique, the AQP uses only those portions of BAV pathways that define data source constructs in terms of target schema constructs, namely, `add`, `extend`, `rename` and `ident` transformations (assuming a target to data source traversal of the pathways).

`LAVInvRulesViewMap` (a specialisation of class `CompositeViewMap` discussed above) uses the *Inverse Rules* technique [2] to derive view definitions of the target schema constructs in terms of the data source schema constructs by inverting the LAV view definitions of the pathway. The LAV reformulation algorithm uses these derived view definitions to unfold the query over the target schema in terms of data source schema constructs. If more than one view definition is available for a particular target schema construct, these view definitions are merged according to the semantics of the `merge` operator defined in [4]. Section 3.5.5 provides more details on how each type of primitive transformation is handled by `CompositeViewMap`.

3.5.4 BAV Query Reformulation

When using BAV as the reformulation technique, the AQP uses all information contained in BAV pathways. `BAVViewMap` (a specialisation of `CompositeViewMap` discussed above) combines the GAV and LAV techniques to derive view definitions of the target schema constructs in terms of the data source schema constructs. If more than one view definition is available for a particular target schema construct then, the view definitions are merged according to the semantics defined in [4]. Section 3.5.5 provides more details on how each type of primitive transformation is handled by `CompositeViewMap`.

3.5.5 Handling Primitive Transformations

`AtomicViewMap` stores details of primitive transformations exactly as these are stored in the AutoMed repository. Each `add/delete` transformation is supplied with a query q, describing the extent of the added/deleted construct using the rest of the schema constructs. Each `extend/contract` transformation is supplied with a query of the form $Range\ q_l\ q_u$, specifying a lower and an upper bound on the extent of the construct. The lower bound may be `Void` and the upper bound may be `Any`, which respectively indicate no known information about the lower or upper bound of the extent of the new construct. `CompositeViewMap` handles each `AtomicViewMap` based on the transformation type as described below.

- `delete(c,`q`)`: Any occurrence of c within the view definitions is replaced by q, which describes how to reconstruct the extent of construct c.
- `contract(c,`$Range\ q_l\ q_u$`)`: Any occurrence of c within the view definitions is replaced either by the lower-bound query q_l, the upper-bound query q_u, or the full query $Range\ q_l\ q_u$, depending on the `Range` semantics configuration parameter of the AQP.
- `add(c,`q`)`: The rule is inverted using the Inverse Rules algorithm so as to get definitions of one or more target schema constructs defined in terms of c. Any occurrences of these target schema constructs within the view definitions are replaced by the body of inverted rule.
- `extend(c,`$Range\ q_l\ q_u$`)`: q_l and q_u are inverted individually in the same way the query supplied with an `add` transformation, to obtain a set of rules with bodies of the form $Range\ Void\ q$ or $Range\ q\ Any$ (see [4] for details).

  Any occurrences of target schema constructs within the view definitions are then replaced by the lower-bound, upper-bound or full *Range* queries in the inverted rule bodies, depending on the `Range semantics` configuration parameter of the AQP.
- `rename(`c,c'`)`: All references to c in the view definitions are replaced by references to c'.
- `ident`: Since the key to the hashmap contained in a `CompositeViewMap` is the AutoMed repository identifier, and since an `ident` transformation states that its schema arguments are equivalent, processing of such a transformation consists of updating the keys in the hashmap of the `CompositeViewMap` object to contain new `SchemeInfo` objects that contain a new repository identifier.

If more than one view definitions are encountered for a particular construct while constructing the `CompositeViewMap`, the definitions are merged using the `merge` operator, as defined in [4].

3.5.6 Handling Dual Model Data Sources

In general, AutoMed allows for a data source to be represented by two data models, one termed datasource-oriented and one termed AutoMed-oriented. Some

data models may be defined using a single data model (e.g. `xmldss`), whereas others may use the dual model approach (e.g. `sql`).

Throughout this section, the term 'data source schema' referred in fact to the AutoMed-oriented schemas. If a data source follows the dual-model approach, an extra reformulation step is needed between the datasource-oriented and the AutoMed-oriented schemas. This is performed in the `CompositeViewMap` component and always uses GAV reformulation to reformulate AutoMed-oriented constructs to datasource-oriented constructs, regardless of the reformulation technique specified in the instance of `QueryProcessorConfiguration` supplied to the AQP.

3.6 Logical optimisation

After the initial query on the virtual schema has been reformulated into a query containing data source schema constructs, the logical optimiser component performs various logical optimisations on the query. The goal of this component is twofold: first, to simplify the query by performing algebraic optimisations, and, second, to build the largest possible subqueries that can be pushed down to the local data sources for evaluation. The `QueryOptimisationProvider` component serves as a 'policy' class, coordinating the individual optimisers. The default configuration for AQP uses the `StandardOptimisationProvider`, but an AutoMed developer can create their own optimisation provider and use it with AQP to suit the needs of their specific integration setting. The rest of this section describes the functionality of each optimiser as well as of the `StandardOptimisationProvider`.

ChooseOptimiser. The `choose` IQL function returns either of its arguments non-deterministically. It is possible, however, to determine which one of the two operands of this IQL operator is cheaper to evaluate using some heuristic. In particular, the `ChooseOptimiser` favours the operand that invokes the lowest number of IQL built-in functions — note that a comprehension is also considered a function in this context.

CollectionSimplification. This optimiser eliminates empty collection arguments within the reformulated query and simplifies the reformulated query based on the semantics of `Null`, `Void` and `Any` with respect to the collection-valued IQL operators (see Appendix A.4 and A.5).

ComprehensionDistributor. Within a comprehension, generators may iterate over expressions of the form e_1 ++ ... ++ e_m where the e_i may refer to different data sources, thereby preventing the query processor from sending the whole comprehension to a single wrapper. Application of this optimiser splits the original comprehension into a number of simpler comprehensions, some of which may now refer to a single data source, and therefore can be sent to a single wrapper. The number of comprehensions created by this optimiser to replace the original comprehension is $\prod_{i=1}^{n} m_i$, where m_i is the number of expressions involved in the i^{th} generator and n is the number of generators. For example, the following query:

$$[x|\{x,y\} \leftarrow (::S_1:::\langle\langle \mathsf{student}, \mathsf{name}\rangle\rangle: \text{++} ::S_2:\langle\langle \mathsf{student}, \mathsf{name}\rangle\rangle:);\\ \{x,z\} \leftarrow (::S_1:::\langle\langle \mathsf{staff}, \mathsf{name}\rangle\rangle: \text{++} ::S_2:::\langle\langle \mathsf{staff}, \mathsf{name}\rangle\rangle:); z =' Fred']$$

would be rewritten as follows:

$[x|\{x,y\} \leftarrow ::S_1::: \langle\langle \mathsf{student}, \mathsf{name} \rangle\rangle :; \{x,z\} \leftarrow ::S_1::: \langle\langle \mathsf{staff}, \mathsf{name} \rangle\rangle :; z =' Fred']++$
$[x|\{x,y\} \leftarrow ::S_1::: \langle\langle \mathsf{student}, \mathsf{name} \rangle\rangle :; \{x,z\} \leftarrow ::S_2::: \langle\langle \mathsf{staff}, \mathsf{name} \rangle\rangle :; z =' Fred']++$
$[x|\{x,y\} \leftarrow ::S_2::: \langle\langle \mathsf{student}, \mathsf{name} \rangle\rangle :; \{x,z\} \leftarrow ::S_1::: \langle\langle \mathsf{staff}, \mathsf{name} \rangle\rangle :; z =' Fred']++$
$[x|\{x,y\} \leftarrow ::S_2::: \langle\langle \mathsf{student}, \mathsf{name} \rangle\rangle :; \{x,z\} \leftarrow ::S_2::: \langle\langle \mathsf{staff}, \mathsf{name} \rangle\rangle :; z =' Fred']$

and therefore the first and fourth comprehensions could be sent down to the data sources of S_1 and S_2 respectively. Note that this optimisation is the equivalent of distributing selections and projections over the union operation in the relational algebra.

EmptyJoinOptimiser. This optimiser eliminates comprehensions for which it can infer that they will return empty results because they are undertaking a join over non-overlapping attributes. In particular, this optimisation can applied over attributes that are known to have globally unique values over the data sources being integrated:

$$[h|e; p_1 \leftarrow e_1; e'; p_2 \leftarrow e_2; e''] \Rightarrow []$$

Here, the patterns p_1 and p_2 need to have one or more variables in common, and the values within e_1 and e_2 corresponding to these variables need to be known to be non-overlapping.

Note that the use of the `ComprehensionDistributor` optimiser can lead to a significant number of comprehensions and use of the `EmptyJoinOptimiser` may be able to eliminate some of these. For example, using the output of the example above, the output of `EmptyJoinOptimiser` optimiser would be as follows, provided that the `EmptyJoinOptimiser` had knowledge (e.g. from the Disjoint IDs configuration parameter) that the key values within the virtual integrated schema arising from S_1 and S_2 are disjoint:

$[x|\{x,y\} \leftarrow ::S_1::: \langle\langle \mathsf{student}, \mathsf{name} \rangle\rangle :; \{x,z\} \leftarrow ::S_1::: \langle\langle \mathsf{staff}, \mathsf{name} \rangle\rangle :; z =' Fred']++$
$[x|\{x,y\} \leftarrow ::S_2::: \langle\langle \mathsf{student}, \mathsf{name} \rangle\rangle :; \{x,z\} \leftarrow ::S_2::: \langle\langle \mathsf{staff}, \mathsf{name} \rangle\rangle :; z =' Fred']$

CollectionOperatorReorganiser. A reformulated query may contain a number of comprehensions referring to different data sources. These comprehensions may be scattered throughout the query, and grouping them together may enable the query processor to send larger subqueries to the wrappers. This optimiser therefore reorders comprehensions which are arguments of the `++` and `--` built-in functions. As a simple example, the following query:

$$[\{x,y\}|\{x,y\} \leftarrow ::S_1::: \langle\langle \mathsf{staff}, \mathsf{name} \rangle\rangle :] ++ [\{x,y\}|\{x,y\} \leftarrow ::S_2::: \langle\langle \mathsf{person}, \mathsf{name} \rangle\rangle :] ++ [\{x,y\}|\{x,y\} \leftarrow ::S_1::: \langle\langle \mathsf{student}, \mathsf{name} \rangle\rangle :]$$

would be rewritten to:

$$[\{x,y\}|\{x,y\} \leftarrow ::S_1::: \langle\langle \mathsf{staff}, \mathsf{name} \rangle\rangle :] ++ [\{x,y\}|\{x,y\} \leftarrow ::S_1::: \langle\langle \mathsf{student}, \mathsf{name} \rangle\rangle :] ++ [\{x,y\}|\{x,y\} \leftarrow ::S_2::: \langle\langle \mathsf{person}, \mathsf{name} \rangle\rangle :]$$

NestingOptimiser. Consider the following query over schemas S_1 and S_2:

$$[f(p_1,p_2,p_3)|p_1 \leftarrow e^1_{S_1}; p_2 \leftarrow e^1_{S_2}; p_3 \leftarrow e^2_{p_1}; q_{p_1}; q_{p_2}; q_{p_{1,2}}]$$

Here, $e^i_{S_j}$ is the expression of the i^{th} generator, referring to schema S_j, q_{p_k} is a filter expression referring to variables from pattern p_k, $q_{p_{k,l}}$ is a filter referring to variables from patterns p_k and p_l, and $f(p_1, p_2, p_3)$ is an arbitrary expression over the variables of the patterns p_i.

This optimiser rewrites the above query by grouping where possible qualifiers with generators, and nesting them into a new comprehension that can be sent as one subquery to a data source wrapper (this is the analogue of pushing selections through joins in relational languages):

$$[f(p_1,p_2,p_3)|\{p_1,p_3\} \leftarrow [\{p_1,p_3\}|p_1 \leftarrow e^1_{S_1}; p_3 \leftarrow e^2_{S_1}; q_{p_1}];\\ p_2 \leftarrow [p_2|p_2 \leftarrow e^1_{S_2}; q_{p_2}]; q_{p_{1,2}}]$$

UnnestOptimiser. This optimiser unnests nested comprehensions. In general, a comprehension of the form:

$$[h_1|e_1; p_1 \leftarrow [p_2|Q_1; \ldots; Q_n]; e_2]$$

can be unnested to:

$$[h_1|e_1; Q'_1; \ldots; Q'_n; e_2]$$

provided the patterns p_1 and p_2 match i.e. p_1 can be obtained from p_2 by variable renaming. Each Q'_i is obtained from Q_i by applying the same renaming.

SQLWrapperSKJOptimiser. As discussed in Section 3.5.6, in AutoMed relational data sources have two layers of schemas, the datasource-oriented schema and the AutoMed-oriented schema. In this dual relational model, a column scheme $\langle\langle \mathsf{R}, \mathsf{a} \rangle\rangle$ representing the i^{th} column of an n-ary table R is reformulated into a selection-projection comprehension $[\{k, a_i\}|\{k, a_1, ..., a_n\} \leftarrow \langle\langle \mathsf{R}, \mathsf{n} \rangle\rangle]$, where $\langle\langle \mathsf{R}, \mathsf{n} \rangle\rangle$ is the datasource-oriented scheme representing R. This causes a global query such as:

$$[\{x,y\}|\{k,x\} \leftarrow \langle\langle \mathsf{R}, \mathsf{a1} \rangle\rangle;\\ \{k,y\} \leftarrow \langle\langle \mathsf{R}, \mathsf{a2} \rangle\rangle].$$

to be reformulated into a join of nested comprehension:

$$[\{x,y\}|\{k,x\} \leftarrow [\{k,a_1\}|\{k,a_1,...,a_n\} \leftarrow \langle\langle \mathsf{R}, \mathsf{n} \rangle\rangle];\\ \{k,y\} \leftarrow [\{k,a_2\}|\{k,a_1,...,a_n\} \leftarrow \langle\langle \mathsf{R}, \mathsf{n} \rangle\rangle]].$$

This self-joining over nested comprehensions iterating over the same table is undesirable as it imposes an unnecessary cost in the evaluation of the query. The nested comprehensions are first removed using the `UnnestOptimiser` described earlier, and so the above query is rewritten as follows:

$$[\{x,y\}|\{k,x,y,...,a_n\} \leftarrow \langle\langle \mathsf{R}, \mathsf{n} \rangle\rangle;\\ \{k,x,y,...,a_n\} \leftarrow \langle\langle \mathsf{R}, \mathsf{n} \rangle\rangle]$$

However, this comprehension still contains a self-join of the table R over its key attribute(s) (k). The `SQLWrapperSKJOptimiser` eliminates such self-joins, and the result for the above query is:

$$[\{x,y\}|\{k,x,y,...,a_n\} \leftarrow \langle\langle \mathsf{R}, \mathsf{n} \rangle\rangle]$$

As another example, consider the following query on relations R and S:

$$[\{x,y,a,b,c\}|\{x,a\} \leftarrow [\{k,a_1\}|\{k,a_1,a_2,a_3\} \leftarrow \langle\langle \mathsf{R},3\rangle\rangle];\\ \{x,b\} \leftarrow [\{k,a_2\}|\{k,a_1,a_2,a_3\} \leftarrow \langle\langle \mathsf{R},3\rangle\rangle];\\ \{y,b\} \leftarrow [\{k,a_1\}|\{k,a_1,a_2\} \leftarrow \langle\langle \mathsf{S},2\rangle\rangle];\\ \{y,c\} \leftarrow [\{k,a_2\}|\{k,a_1,a_2\} \leftarrow \langle\langle \mathsf{S},2\rangle\rangle]; a > c]$$

This query is rewritten by the same two optimisers into:

$$[\{x,y,a,b,c\}|\{x,a,b,a_3\} \leftarrow \langle\langle \mathsf{R},3\rangle\rangle;\\ \{y,b,c\} \leftarrow \langle\langle \mathsf{S},2\rangle\rangle; a > c]$$

ComprehensionHeadSimplifier. The head of a comprehension may contain an arbitrary IQL expression in general. This may have a number of undesirable effects, such some of the optimisers not being applicable, or wrappers not being able to accept a comprehension for translation. The `ComprehensionHeadSimplifier` rewrites comprehension of the form $[h|Q]$ to queries of the form *map* (*lambda* p h) $[p|Q]$, where p is a tuple of variables comprising all the free variables appearing in the expression h.

ConstantInPatternOptimiser. Within comprehensions, equality filters between a variable v and a constant c can be removed, and v is replaced by c in the rest of the of the comprehension. For example, the following query:

$$[\{x_1,y\}|\{x_1,y\} \leftarrow \langle\langle \mathsf{student},\mathsf{name}\rangle\rangle; x_1 = 5; \{x_2,z\} \leftarrow \langle\langle \mathsf{staff},\mathsf{name}\rangle\rangle; x_1 = x_2]$$

is rewritten as:

$$[\{5,y\}|\{5,y\} \leftarrow \langle\langle \mathsf{student},\mathsf{name}\rangle\rangle; \{5,z\} \leftarrow \langle\langle \mathsf{staff},\mathsf{name}\rangle\rangle]$$

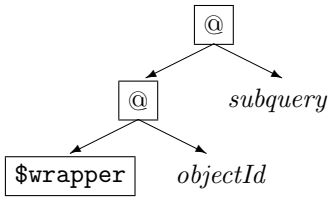
Note that filter $x_1 = x_2$ was removed since it was made redundant. Also note that, if after the optimisation, the head of a *set* comprehension no longer contains variables, the body is removed.

StandardOptimisationProvider. This coordinates the application of the above optimisers to the reformulated query. It first applies the `ChooseOptimiser`, followed by `CollectionSimplification`. Several of the other optimisers are then applied repeatedly, until no optimiser further modifies the query; this is because an optimisation performed by one optimiser may enable further optimisations by the rest of the optimisers, but which were not possible before. The order in which the optimisers are applied within this loop is as follows.

First, the `EmptyJoinOptimiser` is applied, followed by the `ComprehensionDistributor`, transforming complex comprehensions involving multiple data sources into simpler ones, some of which may refer to a single datasource. The `CollectionOperatorReorganiser` next reorganises the query, bringing together comprehensions referring to the same datasource. Then, the `NestingOptimiser` is applied, nesting the resulting comprehension in a way that enables larger subqueries to be sent down to the wrappers. Finally, the `UnnestOptimiser` and the `SQLWrapperSKJOptimiser` are applied.

3.7 Query annotation

After the input query has been reformulated and processed by the logical optimiser, wrapper functions, which are responsible for evaluating IQL queries on the data sources, are inserted within the query. Wrappers are implemented in IQL as a two-argument built-in function; a wrapper subtree is therefore represented as follows:



where *objectId* is the unique id of the Java Wrapper object, assigned at runtime, and *subquery* is the subquery sent to that datasource wrapper.

In AQP v1, this task was performed by the `FragmentProcessor`, which created a wrapper subtree for every scheme within the query. In the current version, this task is performed by the `StandardQueryAnnotationProvider`, which detects the largest possible subqueries that datasource wrappers can handle, and creates the appropriate wrapper subtrees. Section 3.7.1 gives a detailed description of how this is accomplished, while Section 3.7.2 provides details of how a wrapper can be upgraded to handle larger IQL queries.

3.7.1 The QueryAnnotator component

Each AutoMed wrapper type is capable of translating a subset of the IQL language. For example, the `SQLWrapper` can translate simple comprehensions[5] with simple filters[6] and the `++` function; the `SAXWrapper` can translate only single schemes; and the `BBKSQLWrapper` can translate arbitrarily nested comprehensions, aggregation functions and the `++` and `distinct` functions.

We have defined a number of grammars which correspond to subsets of the IQL language and have generated a parser corresponding to each grammar; so if a subquery adheres to a certain grammar, then the corresponding parser will accept it, otherwise it will raise a syntax error.

Each type of wrapper specifies the IQL grammar it can translate (otherwise, the default 'schemes-only' grammar is assumed); the `StandardQueryAnnotationProvider` invokes the parser associated with each datasource wrapper to determine whether the wrapper can translate a given subquery. Note that if a wrapper's translation capabilities do not correspond to an existing grammar, a new grammar must be defined and its corresponding parser must be generated.

The `StandardQueryAnnotationProvider` is initialised by creating a map containing a wrapper w_t corresponding to each data source schema S_t. It then traverses the query DAG, and for each subquery q it encounters replaces q with a wrapper subtree containing q as its second argument if the wrapper w_t can translate the query q.

3.7.2 Upgrading the translation capabilities of AutoMed wrappers

To upgrade an AutoMed wrapper to be able to translate more complex IQL queries than single schemes, the wrapper developer needs to perform the following tasks:

[5]By 'simple comprehensions' we mean non-nested (flat) comprehensions whose head is 'simple', i.e. is not nested and does not contain any functions.

[6]By 'simple filters' we mean filters comprising a comparison between a variable and a constant or between two constants.

1. Modify the wrapper code to accept and be able to evaluate not only single scheme Cells, but larger ASGs.

2. *(Optional)* Define and compile a JavaCC grammar defining the subset of IQL queries that the wrapper can handle.[7]

3. Override the method `public QueryParser getQueryParser(ASG query)` in the wrapper class, so that it returns an instance of the parser created in the previous step.

Step 2 is optional, in that it is not necessary to define a new grammar if the IQL subset the wrapper can handle is already defined in an existing JavaCC grammar. In this case step 2 can be omitted and the parser corresponding to that grammar can be used when overriding the method specified in step 3. Appendix C contains the JavaCC source files for three parsers currently contained in the AutoMed distribution: one that allows only simple schemes; one that allows schemes, simple comprehensions, and the `++` operator; and one that allows the subset of IQL translatable by the `BBKSQLWrapper` discussed earlier.

The following code snippet shows how method `getQueryParser(ASG query)` can be overridden to indicate that the wrapper can handle queries defined by the SimpleCompAppend.jj JavaCC grammar of Appendix C.2.

```
public QueryParser getQueryParser(ASG query) {
 Reader r = new BufferedReader(new StringReader(query.toString()));
 return new SimpleCompAppend(r);
}
```

3.8 Query Evaluation

Evaluation of an annotated query is performed by an instance of the `QueryEvaluationProvider`, which can be obtained through the `QueryEvaluationFactory`. Currently, there is only one implementation, the `Evaluator`.

When an `Evaluator` instance is created, its constructor is passed the same `QueryProcessorConfiguration` instance supplied to the AQP. This contains a `FunctionTable` instance, which contains the IQL built-in functions that are to be used with this instance of AQP and its components. In particular, this object contains mappings between the names of the IQL functions and the Java classes that actually implement them. Each such class implements the `BuiltInFunction` interface, which contains two methods, `getArity()` returning the arity of the function[8], and `perform(Cell[] args, Evaluator e)` implementing the function itself.

The default configuration of the `QueryProcessor` component (see Section 4.3) uses only the `StandardFunctionTable`, which contains the built-in IQL functions described in Appendix A. An AutoMed developer can create his/her own

[7] JavaCC (Java Compiler Compiler) is a parser generator for the Java programming language. JavaCC is similar to Yacc in that it generates a parser for a grammar provided in BNF-like notation, except the output is Java source code.

[8] The `Evaluator` needs to know the arity of a built-in function — how many arguments it takes. Although built-in functions may be partially applied within expressions, in order to be reduced they require a full complement of their arguments.

IQL functions and define a new function table, referencing these functions. The new function table can then be added to the function table used in a certain `QueryProcessorConfiguration` instance, as illustrated in Section 4.3.

3.8.1 The `Evaluator` component

Evaluating a query expressed in a functional language consists of performing reductions on reducible expressions until no more reductions can be performed. It is then said to be in *normal form*. There may be more than one reducible expressions within the query at any one time and thus a choice of reductions, but the order in which these reductions are performed makes no difference and the result is the same, provided the evaluation terminates.

The `Evaluator` component always reduces the leftmost outermost reducible expression — this is known as normal-order reduction and has the best possible termination behaviour. It also has the effect that, function arguments are not evaluated unless they are actually needed for the evaluation of the function. Also, since an IQL query can be internally represented as an acyclic graph, several identical sub-expressions of it may in fact be evaluated at the same time. These two properties provide for "lazy evaluation" in IQL.

The `Evaluator` contains methods for full evaluation of an expression i.e. reduction to normal form, and for reduction to the so-called *weak head normal form* (information on weak head normal form can be found in [5]).

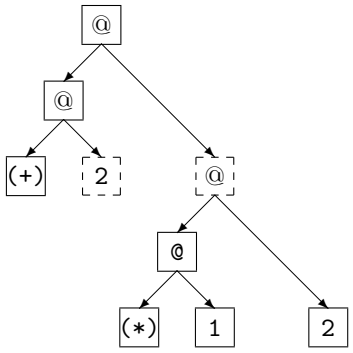
The reductions performed by the `Evaluator` component are of two types — reduction of lambda expressions and reduction of built-in functions. Higher level constructs, such as let expressions and comprehensions, are evaluated by first being translated into their equivalent expressions comprising lambda abstractions and built-in functions (as discussed in Section 2.2 above).

The built-in functions of IQL fall into two categories. There are ordinary built-in functions, such as `sort`, `(+)` and the like, which are part of the IQL language aimed at the user. There are also special built-in functions, such as `Comp`, `Let` and `Gen`, which are not intended for direct use but exist to represent internally constructs of IQL such as comprehensions and let expressions. These special functions provide a view of these constructs that turns them into simple function applications like the rest of IQL.

3.8.2 Implementation of built-in functions

A number of standard functions have been implemented. The functions of Appendix A are included in the `StandardFunctionTable` class, which extends `FunctionTable` with a number of standard IQL functions.

To illustrate the evaluation of IQL functions, consider the query $2 + (1 * 2)$. The following figure shows the tree representation of this query that the `Evaluator` will simplify:



In evaluating this query the `Evaluator` will obtain a `BuiltInFunction` for the (+) function from its `FunctionTable`. It will then call the `perform(Cell args[], Evaluator e)` method of this `BuiltInFunction` passing it references to the two dashed cells as the `args[]` argument. However, the method cannot work out the result because one of the arguments is an expression in need of evaluation. It therefore uses the reference to the `Evaluator` object that it has been passed to evaluate this argument first.

As illustrated with the above example, the `perform` method will frequently need to evaluate fully or bring to weak head normal form some of the arguments it is passed before it can return a result. It must therefore be provided with a reference to the `Evaluator` object it should use to accomplish this task. The decision to evaluate these arguments, bring them to weak head normal form, or leave them as-is is left to the `BuiltInFunction`, because the `Evaluator` doesn't know whether it will be necessary to evaluate them or not. For example, consider a query of the form ($False\ and\ e$); in this case, the IQL function *and* does not need to evaluate e at all.

3.8.3 Incremental evaluation

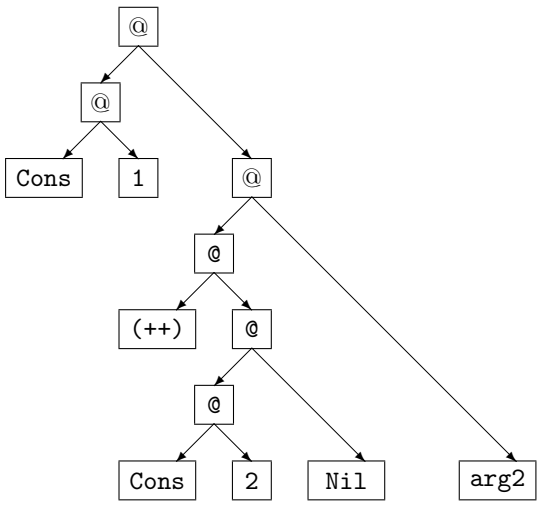
The `Evaluator` supports *streaming* for those operators with at least one collection argument. This means that, when evaluating a collection-valued expression, and if the user wishes so (see Section 4), the `Evaluator` can evaluate the expression *incrementally*, i.e. a single call to the `Evaluator` produces only n results of the final collection, where n is user-defined. This feature is particularly useful for large or indeed infinite collections.

To enable incremental processing of functions with collection arguments, the `Evaluator.evaluate(Cell)` method has been modified so as to stop processing whenever n results of the final collection have been produced. To achieve this, the `Evaluator.weakHeadNF(Cell)` method has been modified to count the number of times that any function that supports streaming is invoked. The next n results are then produced at the user's request, with a new call to `Evaluator.evaluate(Cell)` (see Section 4).

The functions that currently support incremental query processing are (++), $flatmap$ and $\$wrapper$.

We now describe the streaming version of function (++). Given collection-valued arguments arg_1 and arg_2, instead of processing arg_1 and appending the unprocessed arg_2 at the end of arg_1 (as is the case with instant evaluation), the function now processes n items from the collection $arg1$ and appends the

expression (arg'_1 ++ arg_2) to the end of these n processed items, where arg'_1 is the collection remaining after these n items have been removed from arg. For example, for $arg_1 = [1, 2]$ and with $n = 1$, the first call to (arg_1 ++ arg_2) would yield:



Stream processing for function $\$wrapper$ is implemented at the wrapper level, rather than at the IQL function itself, due to the differences of the implementation details for each wrapper that can support streaming. The `SQLWrapper`, currently the only wrapper that supports incremental processing of queries, implements streaming as follows. An IQL query q, with unique identifier id, received by the `SQLWrapper` is first sent for evaluation to the data source and a streaming `ResultSet` JDBC object is created. This is stored in a cache in the wrapper, to which the key is id. A single execution call at any wrapper is similar to that of function (++), i.e. the same intermediate structure is created. The only difference is that the `SQLWrapper` needs to store a pointer to the current row in the query result, in order to be able to continue evaluation from the right index in subsequent calls.

3.9 Extensibility

The AQP components have been designed with extensibility and adaptability in mind. This means that low-level components have been designed to be as generic as possible and higher-level components to be as adaptable as possible.

Function tables ensure that each developer can easily create his/her own IQL functions, create a new function table and add it to the `QueryProcessor` at hand using class `QueryProcessorConfiguration` (see Section 4).

Query reformulation has been split into three layers, making it easier to provide BAV, GAV and LAV reformulation by reducing as much as possible the reformulation-specific code. As a result, adding alternative reformulation components is straightforward.

The component-based approach followed for the development of the AQP allows for the development of different implementations for each query processing component. This is especially useful for the optimisation process, where each different integration setting may require a different optimisation strategy.

Finally, the `QueryProcessorConfiguration` component provides a single point of configuration information for all other query processing components.

3.10 Logging

Logging within the AQP is supported using the Java logging API (see `http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/overview.html`). The location of the log file is `/.automed/logs/AutoMedQProcPackage.log` within the AutoMed user's home directory.

3.11 Test Units

A number of unit and system tests have been developed to help maintain and debug AutoMed code. Unit tests validate the functionality of individual classes, whereas system tests validate the functionality of larger components of the AutoMed architecture.

JUnit[9] 3.8.1 has been used as the unit testing framework in this release, but it is envisaged that JUnit 4.0 will be used when AutoMed switches to Java 5.

3.12 Memory & Performance Issues

This section discusses memory and performance issues raised during the implementation of the AQP. Section 3.12.1 discusses a caching mechanism for the AQP, Section 3.12.2 discusses issues concerning garbage collection, and, finally, Section 3.12.3 discusses the importance of carefully choosing the Java List implementation to use when developing components for the AQP.

3.12.1 Cache

The `CallToWrapper` IQL function (not listed in Appendix A as it is only used internally) implements a basic caching system. Whenever a wrapper call is made, the cache stores the query sent to the wrapper together with the result of the wrapper call in a Java `HashMap` object. If the result of an intermediate query is larger than a certain threshold, then the result is stored in a file on disk; otherwise the result is stored in memory. To avoid returning stale data to the user, the `Evaluator` component clears the cache after each query is evaluated and so caching only occurs within a single query, not across queries. This fact renders the cache discussed here an internal `QueryEvaluator` mechanism, rather than a query processing cache.

The usage of a threshold to store a result in memory or on disk implies that the caching mechanism calculates the size of a result. However, calculating the exact size would be costly and could degrade performance significantly. The cache therefore *estimates* the size of the result, by counting the number of items in the resulting collection. This is an estimation, because *e.g.* an item in a collection can have an arbitrary size or may be another collection.

[9] See http://www.junit.org

3.12.2 Garbage collection

Initial tests on the AQP revealed that, even though garbage collection is performed normally by the Java Virtual Machine, explicit calls must be made across the AQP, to release in-memory ASG objects no longer needed. For example, after a query is optimised by the `QueryOptimiser` component, explicit calls make all optimiser objects null to make them available for garbage collection; this is necessary, as they may contain data structures such as hashmaps containing large ASG objects. Other explicit calls include clearing the `CallToWrapper` cache and re-initialising all IQL functions.

This initialisation of AQP components occurs once when first creating an AQP instance and then *after* a component is used. This happens so that ASG/Cell objects no longer necessary for the query processing cycle are available for garbage-collection immediately. The only ASG/Cell objects present in memory after a query processing cycle are those within the `QueryReformulator` component and those used to represent optimisation rules within optimisers.

3.12.3 Java collection types

Within the AQP, which is fully implemented in Java, there are two types of List implementations widely used: ArrayList and LinkedList. LinkedList objects are doubly linked lists, whereas ArrayList objects are more similar to arrays, but can still be augmented. It is therefore crucial for the overall system performance to carefully select which List implementation is used, as each one has advantages in different situations.

The ArrayList offers constant time positional access, meaning that it is faster to retrieve an item. On the other hand, the LinkedList is faster when one adds elements to the beginning of the List or iterates over the List to delete elements from its interior, as it requires constant time whereas in ArrayList requires linear time. However, positional access requires linear time in a LinkedList and constant time in an ArrayList. Furthermore, the constant factor for LinkedList is much worse.

4 Using AQP - Examples

4.1 Creating IQL queries

There are two ways in which `ASG` structures can be created. The first is to create the textual representation, then parse it. For example, the code needed to create the expression `1+2` is `ASG g = new ASG("1+2");`.

The second method is to create each `Cell` of the `ASG` individually. We demonstrate this method by listing the code necessary to create different kinds of structures.

Constants, constructors, etc. The following illustrate the creation of `Cell` objects representing, respectively, a String value, an Integer value, a datetime value, and a Wrapper function:

```
Cell c = new Cell("'Computer Science'");
Cell c = new Cell("12345");
Cell c = new Cell("dt '2005-02-24 23:15:48'");
```

```
Cell c = new Cell("$wrapper");
```

The `Cell` constructor determines the type of `Cell` to create using method `ReservedTokens.returnTag(String)`.

Functions. The following illustrates the creation of the expression `1+2`:

```
Cell arg1 = new Cell(Cell.TAG_INTEGER,1);
Cell arg2 = new Cell(Cell.TAG_INTEGER,2);
Cell op = new Cell("(+)");
Cell exp = new Cell(new Cell(op,arg1),arg2);
```

Tuples. The following illustrates the creation of 3-tuple {`1,x,'IQL'`}:

```
Cell t1 = new Cell(Cell.TAG_INTEGER,1);
Cell t2 = new Cell(Cell.TAG_VARIABLE, "x");
Cell t3 = new Cell(Cell.TAG_STRING,"IQL");
Cell tCons = new Cell("Tuple3");
Cell tuple = new Cell(new Cell(new Cell(tCons,t1),t2),t3);
```

Collections. Collections are created similarly:

```
Cell nil = new Cell("Nil");
Cell i1 = new Cell(Cell.TAG_INTEGER,1);
Cell i2 = new Cell(Cell.TAG_INTEGER,2);
Cell i3 = new Cell(Cell.TAG_INTEGER,3);
Cell ci3 = new Cell(new Cell(new Cell("Cons"),i3),nil);
Cell ci2 = new Cell(new Cell(new Cell("Cons"),i2),ci3);
Cell ci1 = new Cell(new Cell(new Cell("Cons"),i1),ci2);
Cell list = ci1;
```

Alternatively, one could use method `ASG.toASGCollection(List l, int t)`, where `t` is one of the constants `ASG.COLLECTION_LIST`, `ASG.COLLECTION_SET` or `ASG.COLLECTION_BAG`:

```
List l = new ArrayList();
l.add(i1); l.add(i2); l.add(i3);
Cell list = ASG.toASGCollection(l, ASG.COLLECTION_LIST).root();
```

Comprehensions. The following code creates comprehension $[x|\{x\} \leftarrow \langle\langle A \rangle\rangle]$:

```
Cell gen = new Cell(new Cell(new Cell("Gen"),
                             new Cell("x"),
                    new Cell(Cell.TAG_SCHEME,new SchemeInfo("<<A>>")));
Cell comp = new Cell(new Cell(new Cell("Comp"),
                              gen),
                     new Cell(new Cell("Tuple1"),
                              new Cell("x")));
```

4.2 Creating an AQP instance

```
QueryProcessor qp = new QueryProcessor();
```

A client program is free to create as many AQP instances as desired, so as to parallelise query processing. No two instances of AQP interact or interfere with each other in any way.

4.3 Using `QueryProcessorConfiguration`

Default Configuration:

```
QueryProcessorConfiguration qpc = new QueryProcessorConfiguration();
```

Specifying a custom implementation for an AQP component:

```
qpc.setEvaluator("mypackage.myEvaluator");
```

Using the `FragmentProcessor` as the annotation provider:

```
qpc.setAnnotator("uk.ac.bbk.dcs.automed.qproc.annotate.FragmentProcessor");
```

Changing the default reformulation semantics:

```
qpc.setReformulationSemantics(QueryProcessorConfiguration.REFORMULATION_GAV);
```

Changing the default `Range` semantics:

```
qpc.setRangeSemantics(QueryProcessorConfiguration.RANGE_LOWER);
```

Adding a custom Function Table:

```
FunctionTable[] ft = {new XMLFunctionTable()};
qpc.addFunctionTable(ft);
```

Setting SQL as the input query language for AQP:

```
qpc.setQueryLanguage(QueryProcessorConfiguration.QUERY_LANGUAGE_SQL);
```

Setting XQuery as the input query language for AQP:

```
qpc.setQueryLanguage(QueryProcessorConfiguration.QUERY_LANGUAGE_XQUERY);
```

Switching off optimisation:

```
qpc.setOptimisationState(false); // true is default
```

Switching off the cache of the evaluator:

```
qpc.setEvaluationProviderCacheStatus(false); // true is default
```

4.4 Retrieving data

For a query to be evaluated, the wrapper subtrees within an ASG have to be replaced with actual data. An AutoMed wrapper is able to retrieve data from its corresponding data source by using the `executeIQL` method. This method translates an IQL query into the data source's query language, submits it for evaluation to the data source, and returns the result in the form of an `ASG` collection — list, set or bag.

The following code shows the 'top to bottom' processing of an IQL query with respect to a given target schema and a set of data source schemas:

```
Schema target = Schema.getSchema("GLOBAL");
Schema[] sources = new Schema[2];
sources[0] = Schema.getSchema("LOCAL1");
sources[1] = Schema.getSchema("LOCAL2");
String query = "<<person,pname>>";
String integrationSemantics = "(++)";
try {
  ASG g = new ASG(query);
  QueryProcessor qp = new QueryProcessor();
  QueryProcessorConfiguration qpc = new QueryProcessorConfiguration();
  qp.process(q, target, sources, integrationSemantics, qpc);
  System.out.print("Result: ");
  g.println();
} catch (QProcException e) {
  if ((e.getCause() instanceof LAVQueryReformulationException) &&
     (qpc.getReformulationSemantics() ==
      QueryProcessorConfiguration.REFORMULATION_LAV_INV_RULES)){
      // LAV query reformulation may not succeed always.
      // This is a valid limitation of LAV reformulation
      System.out.println("###LAV Issue:");
  } else {
      // This error should not occur.
      System.out.println("###Other Issue:");
  }
  qpc.printConfiguration();
  e.printStackTrace();
}
```

5 Example Integration Settings

A number of example integration settings have been developed to help demonstrate and test query processing in AutoMed. Currently, there are three such examples:

- University Database Integration example located in the examples directory of the CVS. The following classes need to be run in the given order: UniversityDatabaseWrapping, UniversityDatabseIntegration, UniversityDatabaseQuerying. Schema information and instructions on installing the participating databases can be found on `http://www.doc.ic.ac.uk/~pjm/databases/index.html`.

- Student Database Integration example (adapted from [4]). This is located in the `/query_processing/ student` package in the examples directory of the CVS. The following classes need to be run in the given order: StudentDatabaseWrapping, StudentDatabaseIntegration, StudentDatabaseQuerying. The schemas of the databases participating in this integration are listed in StudentSchema.txt.

- Hospital Database Integration example (adapted from [3]). This is located in the `/query_processing/ hospital` package in the examples directory of the CVS. The following classes need to be run in the given order: HospitalDBWrapping, HospitalDBIntegration, HospitalDBQuerying. The

schemas of the databases participating in this integration are listed in file HospitalSchema.txt.

6 Changes

The latest version of the query processor contains a number of improvements and new features:

New features:

- Support for LAV and BAV query reformulations (earlier only GAV reformulation was available).
- Modular Query Reformulator architecture
- Customisable Query Processor configuration
- Support for incremental query processing

Changes from the earlier version:

- Support for logging
- Added test units for a range of the AQP components
- Datatypes instead of constants; functions have type signatures.
- `LogicalOptimiser` component.
- `QueryAnnotator` component (replacing the `FragmentProcessor`).
- `AQP` component (coordinating query processing components).
- Cache system for the `CallToWrapper` IQL function.
- Evaluator can handle multiple function tables.
- `ident` transformations now come in four flavours.
- Support for sets and bags.
- New constructor: `Null`.
- SQL and XQuery support.
- Comprehension translation is now correct.
- Variable unification is now correct.
- Infix versions of all binary IQL built-in functions.
- Optimised implementations for the built-in functions (reduction of arguments to weak-head normal form where appropriate instead of always evaluating arguments).
- Set of built-in functions augmented (see Appendix A).
- IQL functions now have a consistent way of handling `Void`, `Any` and `Null`.

7 Future Work

It is envisaged that the next version of the query processor will contain the following features:

- **Type checker.** The type system infrastructure is one of the new features of version 1.2 of the query processor of the AutoMed toolkit; the next step is to provide a `TypeChecker` component.
- **Parallel evaluation.** Currently, all IQL queries are evaluated in a serial fashion. The parallelisation of the query processor would give a significant performance boost.
- **Translation.** Increase high-level query language support.
- **Streaming.** Support for more IQL functions.

References

[1] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.

[2] O. M. Duschka and M. R. Genesereth. Answering recursive queries using views. In *Proc of PODS'97*, pages 109–116. ACM Press, 1997.

[3] P.J. McBrien and A. Poulovassilis. Defining Peer-to-Peer Data Integration using Both as View Rules. In *Proc. of DBISP2P'03*, pages 91–107. Springer-Verlag, 2003.

[4] P.J. McBrien and A. Poulovassilis. P2P query reformulation over Both-as-View data transformation rules. In *Proc. of DBISP2P'06*, page TBC. Springer-Verlag, 2006.

[5] S. L. Peyton-Jones. *Implementing Functional Programming Languages.* Prenctice-Hall International, 1992.

[6] A. Poulovassilis. The AutoMed Intermediate Query Langauge. AutoMed Technical Report 2, June 2001.

[7] A. Poulovassilis and L. Zamboulis. A tutorial on the IQL query language. AutoMed Technical Report 28, July 2008.

[8] A. Poulovassilis S. Courtenage. Combining inheritance and parametric polymorphism in a functional database language. In *Proc. BNCOD'95*, pages 24–46, 1995.

[9] L. Zamboulis and A. Poulovassilis. Using AutoMed for XML data transformation and integration. In *Proc. DIWeb'04 (at CAiSE'04)*, 2004.

A Built-In Functions

Note that below 'occurs(x,xs)' denotes the number of occurrences of a value 'x' in a collection 'xs'. Note also the following polymorphic type signatures for the `Void, Any` and `Null` constructors:

```
Void::List a
    ::Bag a
    ::Set a
Any ::List a
    ::Bag a
    ::Set a
Null::a
```

Generally, `Null` has semantics 'unknown' (and hence the same semantics as `Any` if used in place of a collection (list, set or bag)). `Void` has the semantics of an empty collection. `Any` has the semantics of 'unknown collection'.

A.1 Arithmetic Operators

name	type	behaviour
+	$Integer \rightarrow Integer \rightarrow Integer$ $Float \rightarrow Float \rightarrow Float$ $Float \rightarrow Integer \rightarrow Float$ $Integer \rightarrow Float \rightarrow Float$	add
-	$Integer \rightarrow Integer \rightarrow Integer$ $Float \rightarrow Float \rightarrow Float$ $Float \rightarrow Integer \rightarrow Float$ $Integer \rightarrow Float \rightarrow Float$	subtract
*	$Integer \rightarrow Integer \rightarrow Integer$ $Float \rightarrow Float \rightarrow Float$ $Float \rightarrow Integer \rightarrow Float$ $Integer \rightarrow Float \rightarrow Float$	multiply
/	$Integer \rightarrow Integer \rightarrow Integer$ $Float \rightarrow Float \rightarrow Float$ $Float \rightarrow Integer \rightarrow Float$ $Integer \rightarrow Float \rightarrow Float$	divide
div	$Integer \rightarrow Integer \rightarrow Integer$	integer division
mod	$Integer \rightarrow Integer \rightarrow Integer$	modulo

Each of the arithmetic functions returns `Null` if either of its arguments is `Null`; also `/` and `div` return `Null` if their second argument is 0.

A.2 Comparison operators

name	type	behaviour
=	$a \rightarrow a \rightarrow Boolean$	returns whether its two arguments are identical
<>	$a \rightarrow a \rightarrow Boolean$	returns whether its two arguments are not identical
>	$a \rightarrow a \rightarrow Boolean$	returns whether its first argument is alphanumerically greater than its second argument
<	$a \rightarrow a \rightarrow Boolean$	returns whether its first argument is alphanumerically less than its second argument
>=	$a \rightarrow a \rightarrow Boolean$	returns whether its first argument is alphanumerically greater than or equal to its second argument
<=	$a \rightarrow a \rightarrow Boolean$	returns whether its first argument is alphanumerically less than or equal to its second argument
like	$String \rightarrow String \rightarrow Boolean$	returns whether its first argument matches the regular expression that is given as the second argument (implementation is equivalent to the ANSI SQL operator)

Note that these operators implement syntactic equality and alphanumeric ordering for constructors such as `Void, Any, Null`. These operators always return either `True` or `False` except in the case when one or both of their arguments is one of the `Const-i` constructors discussed in [4].

A.3 Boolean Operators

<table>
<tr><th>name</th><th>type</th><th>behaviour</th></tr>
<tr><td>and</td><td>$Boolean \rightarrow Boolean \rightarrow Boolean$</td><td>(e1 and e2) returns the conjunction of e1 and e2; here is its definition:
<table>
<tr><td></td><td>True</td><td>False</td><td>Null</td></tr>
<tr><td>True</td><td>True</td><td>False</td><td>Null</td></tr>
<tr><td>False</td><td>False</td><td>False</td><td>False</td></tr>
<tr><td>Null</td><td>Null</td><td>False</td><td>Null</td></tr>
</table></td></tr>
<tr><td>or</td><td>$Boolean \rightarrow Boolean \rightarrow Boolean$</td><td>(e1 or e2) returns the disjunction of e1 and e2; here is its definition:
<table>
<tr><td></td><td>True</td><td>False</td><td>Null</td></tr>
<tr><td>True</td><td>True</td><td>True</td><td>True</td></tr>
<tr><td>False</td><td>True</td><td>False</td><td>Null</td></tr>
<tr><td>Null</td><td>True</td><td>Null</td><td>Null</td></tr>
</table></td></tr>
<tr><td>not</td><td>$Boolean \rightarrow Boolean$</td><td>(not e) returns the negation of e; returns Null if e is Null</td></tr>
<tr><td>if</td><td>$Boolean \rightarrow a \rightarrow a \rightarrow a$</td><td>(if e e1 e2) returns e1 if e is True, e2 if e is False and Null if e is Null</td></tr>
</table>

A.4 Collection Functions

<table>
<tr><th>name</th><th>type(s)</th><th>behaviour</th></tr>
<tr><td>++</td><td>(Set a) → (Set a) → (Set a)</td><td>returns the set-union of two sets</td></tr>
<tr><td></td><td>(Bag a) → (Bag a) → (Bag a)</td><td>returns the bag-union of two bags
i.e. xs ++ ys returns a bag zs
such that for any value t:
occurs(t,zs) = occurs(t,xs)+occurs(t,ys)</td></tr>
<tr><td></td><td>(List a) → (List a) → (List a)</td><td>appends two lists</td></tr>
<tr><td></td><td></td><td>The table below defines its behaviour:
<table>
<tr><td>++</td><td>e2</td><td>Void</td><td>Null</td><td>Any</td></tr>
<tr><td>e1</td><td>e1++e2</td><td>e1</td><td>Null</td><td>Any</td></tr>
<tr><td>Void</td><td>e2</td><td>[]</td><td>Null</td><td>Any</td></tr>
<tr><td>Null</td><td>Null</td><td>Null</td><td>Null</td><td>Null</td></tr>
<tr><td>Any</td><td>Any</td><td>Any</td><td>Null</td><td>Any</td></tr>
</table></td></tr>
<tr><td>−−</td><td>(Set a) → (Set a) → (Set a)</td><td>returns the set-difference of two sets</td></tr>
<tr><td></td><td>(Bag a) → (Bag a) → (Bag a)</td><td>returns the bag-difference of
two bags i.e. xs – ys returns
a bag zs such that for any value t:
occurs(t,zs) = max(0,occurs(t,xs)-occurs(t,ys))</td></tr>
<tr><td></td><td>(List a) → (List a) → (List a)</td><td>returns the difference of its two
list arguments, treating them as
bags and producing the output
in sorted order</td></tr>
<tr><td></td><td></td><td>The table below defines its behaviour:
<table>
<tr><td>--</td><td>e2</td><td>Void</td><td>Null</td><td>Any</td></tr>
<tr><td>e1</td><td>e1 -- e2</td><td>e1</td><td>Null</td><td>Any</td></tr>
<tr><td>Void</td><td>[]</td><td>[]</td><td>Null</td><td>Any</td></tr>
<tr><td>Null</td><td>Null</td><td>Null</td><td>Null</td><td>Null</td></tr>
<tr><td>Any</td><td>Any</td><td>Any</td><td>Null</td><td>Any</td></tr>
</table></td></tr>
<tr><td>intersect</td><td>(Set a) → (Set a) → (Set a)</td><td>returns the intersection of two sets</td></tr>
<tr><td></td><td>(Bag a) → (Bag a) → (Bag a)</td><td>returns the intersection of two bags
i.e. intersect xs ys returns a bag zs
such that for any value t:
occurs(t,zs) = min(occurs(t,xs),occurs(t,ys))</td></tr>
<tr><td></td><td>(List a) → (List a) → (List a)</td><td>returns the intersection of two lists,
treating them as bags, and returning
a sorted list as output</td></tr>
<tr><td></td><td></td><td>The table below defines its behaviour:
<table>
<tr><td>intersect</td><td>e2</td><td>Void</td><td>Null</td><td>Any</td></tr>
<tr><td>e1</td><td>e1 intersect e2</td><td>[]</td><td>Null</td><td>Any</td></tr>
<tr><td>Void</td><td>[]</td><td>[]</td><td>Null</td><td>Any</td></tr>
<tr><td>Null</td><td>Null</td><td>Null</td><td>Null</td><td>Null</td></tr>
<tr><td>Any</td><td>Any</td><td>Any</td><td>Null</td><td>Any</td></tr>
</table></td></tr>
</table>

<table>
<tr><th>name</th><th>type(s)</th><th>behaviour</th></tr>
<tr><td>union</td><td>$(Set\ a) \rightarrow (Set\ a) \rightarrow (Set\ a)$</td><td>returns the set-union of two sets</td></tr>
<tr><td></td><td>$(Bag\ a) \rightarrow (Bag\ a) \rightarrow (Bag\ a)$</td><td>union xs ys returns a bag zs
such that for any value t:
occurs(t,zs) = max(occurs(t,xs),occurs(t,ys))</td></tr>
<tr><td></td><td>$(List\ a) \rightarrow (List\ a) \rightarrow (List\ a)$</td><td>returns the ‘union’ of its two list
arguments, treating them as bags and
producing the output in sorted order</td></tr>
<tr><td></td><td></td><td>The table below defines its behaviour:
<table>
<tr><td>union</td><td>e2</td><td>Void</td><td>Null</td><td>Any</td></tr>
<tr><td>e1</td><td>e1 union e2</td><td>e1</td><td>Null</td><td>Any</td></tr>
<tr><td>Void</td><td>e2</td><td>[]</td><td>Null</td><td>Any</td></tr>
<tr><td>Null</td><td>Null</td><td>Null</td><td>Null</td><td>Null</td></tr>
<tr><td>Any</td><td>Any</td><td>Any</td><td>Null</td><td>Any</td></tr>
</table></td></tr>
<tr><td>flatmap</td><td>$(a \rightarrow List\ b) \rightarrow (List\ a) \rightarrow (List\ b)$
$(a \rightarrow Bag\ b) \rightarrow (Bag\ a) \rightarrow (Bag\ b)$
$(a \rightarrow Set\ b) \rightarrow (Set\ a) \rightarrow (Set\ b)$</td><td>applies a collection-valued function
to each element of a collection and
combines the results using ++;
returns <code>Void/Any/Null</code> if its argument
is <code>Void/Any/Null</code> respectively</td></tr>
<tr><td>foldl</td><td>$(a \rightarrow b) \rightarrow (b \rightarrow b \rightarrow c)$
$\rightarrow (List\ c) \rightarrow (List\ a) \rightarrow (List\ c)$
$(a \rightarrow b) \rightarrow (b \rightarrow b \rightarrow c)$
$\rightarrow (Bag\ c) \rightarrow (Bag\ a) \rightarrow (Bag\ c)$
$(a \rightarrow b) \rightarrow (b \rightarrow b \rightarrow c)$
$\rightarrow (Set\ c) \rightarrow (Set\ a) \rightarrow (Set\ c)$</td><td>foldl f op e xs applies f to each
element of the collection xs and
combines the result using op; e is
returned if xs is empty;
returns <code>Void/Any/Null</code> if its fourth argument
is <code>Void/Any/Null</code> respectively</td></tr>
<tr><td>foldr</td><td colspan="2">same as foldl except that foldl processes the collection starting from its first
element (left-recursion) and foldr starting from its last element (right-recursion)</td></tr>
<tr><td>map</td><td>$(a \rightarrow b) \rightarrow (List\ a) \rightarrow (List\ b)$
$(a \rightarrow b) \rightarrow (Bag\ a) \rightarrow (Bag\ b)$
$(a \rightarrow b) \rightarrow (Set\ a) \rightarrow (Set\ b)$</td><td>applies a function to each element of a collection;
returns <code>Void/Any/Null</code> if its argument is
<code>Void/Any/Null</code> respectively</td></tr>
<tr><td>member</td><td>$(List\ a) \rightarrow a \rightarrow Boolean$
$(Bag\ a) \rightarrow a \rightarrow Boolean$
$(Set\ a) \rightarrow a \rightarrow Boolean$</td><td>returns whether its second argument is a member
of its first argument; returns <code>Null</code> if its first
argument is <code>Any/Null</code>; returns <code>False</code> if its
first argument is <code>Void</code></td></tr>
<tr><td>single</td><td>$a \rightarrow (List\ a)$</td><td>creates a singleton list</td></tr>
<tr><td>singleBag</td><td>$a \rightarrow (Bag\ a)$</td><td>creates a singleton bag</td></tr>
<tr><td>singleSet</td><td>$a \rightarrow (Set\ a)$</td><td>creates a singleton set</td></tr>
<tr><td>sub</td><td>$(Set\ a) \rightarrow (Set\ a) \rightarrow Boolean$</td><td>returns whether its first argument is
a subset of its second argument</td></tr>
<tr><td></td><td>$(Bag\ a) \rightarrow (Bag\ a) \rightarrow Boolean$</td><td>returns whether its first argument is
a sub-bag of its second argument</td></tr>
<tr><td></td><td>$(List\ a) \rightarrow (List\ a) \rightarrow Boolean$</td><td>treats its two arguments as bags and
returns whether its first argument is
a sub-bag of its second argument</td></tr>
<tr><td></td><td></td><td>returns <code>Null</code> if either argument
is <code>Void/Any/Null</code></td></tr>
<tr><td>choose</td><td>$a \rightarrow a \rightarrow a$</td><td>non-deterministically returns
one of its two arguments</td></tr>
</table>

A.5 Aggregation Functions

name	type	behaviour
avg	$(List\ Integer) \rightarrow Float$ $(List\ Float) \rightarrow Float$ $(Bag\ Integer) \rightarrow Float$ $(Bag\ Float) \rightarrow Float$ $(Set\ Integer) \rightarrow Float$ $(Set\ Float) \rightarrow Float$	returns the average of a collection of numbers; returns Null if its argument is an empty collection or `Void/Any/Null`
count	$(List\ a) \rightarrow Integer$ $(Bag\ a) \rightarrow Integer$ $(Set\ a) \rightarrow Integer$	returns the cardinality of any collection; returns Null if its argument is Any/Null; returns 0 if its argument is Void or empty
max	$(List\ a) \rightarrow a, (Bag\ a) \rightarrow a,$ $(Set\ a) \rightarrow a$	returns the maximum element of any first-order collection, according to $>$; returns `Void/Any/Null` if its argument is `Void/Any/Null` respectively
min	$(List\ a) \rightarrow a, (Bag\ a) \rightarrow a,$ $(Set\ a) \rightarrow a$	returns the minimum element of any first-order collection, according to $<$; returns `Void/Any/Null` if its argument is `Void/Any/Null` respectively
sum	$(ListInteger) \rightarrow Integer$ $(ListFloat) \rightarrow Float$ $(BagInteger) \rightarrow Integer$ $(BagFloat) \rightarrow Float$ $(SetInteger) \rightarrow Integer$ $(SetFloat) \rightarrow Float$	returns the sum of any numeric collection; returns `Null` if its argument is `Any/Null`; returns 0 if its argument is `Void` or empty
group	$(List\ (Product2\ a\ b)) \rightarrow$ $(List\ (Product2\ a\ (List\ b)))$ $(Bag\ (Product2\ a\ b)) \rightarrow$ $(Bag\ (Product2\ a\ (Bag\ b)))$ $(Set\ (Product2\ a\ b)) \rightarrow$ $(Set\ (Product2\ a\ (Set\ b)))$	groups a collection of pairs according to their first component and returns a collection of pairs whose second component is a collection; returns `Void/Any/Null` if its argument is `Void/Any/Null` respectively
gc	$((List\ b) \rightarrow c) \rightarrow (List\ (Product2\ a\ b)) \rightarrow (List\ (Product2\ a\ c))$ $((Bag\ b) \rightarrow c) \rightarrow (Bag\ (Product2\ a\ b)) \rightarrow (Bag\ (Product2\ a\ c))$ $((Set\ b) \rightarrow c) \rightarrow (Set\ (Product2\ a\ b)) \rightarrow (Set\ (Product2\ a\ c))$	groups a collection of pairs (its second argument) according to their first component, and applies an aggregation function (its first argument) to the second component of each group; returns `Void/Any/Null` if its argument is `Void/Any/Null` respectively
sort	$(List\ a) \rightarrow (List\ a)$	sorts a list of values; returns `Void/Any/Null` if its argument is `Void/Any/Null` respectively
distinct	$(List\ a) \rightarrow (List\ a),$ $(Bag\ a) \rightarrow (Bag\ a)$	removes duplicates from a list or bag, outputting a sorted collection; returns `Void/Any/Null` if its argument is `Void/Any/Null` respectively

A.6 Type Conversion Functions

There are two types of conversion functions, those used for converting between primitive data types, and those used for converting between collections.

name	type(s)	behaviour
toString	$Integer \rightarrow String$ $Float \rightarrow String$ $DateTime \rightarrow String$	converts an input Integer/Float/DateTime into a String
toInteger	$String \rightarrow Integer$ $Float \rightarrow Integer$	converts an input String/Float into an Integer (float conversion uses only the integer part)
toFloat	$Integer \rightarrow Float$ $String \rightarrow Float$	converts an input String/Integer into a Float
toDateTime	$String \rightarrow DateTime$	converts an input String into a DateTime (does *NOT* check whether the format is correct)

Each of the conversion functions for primitive data types returns `Null` if its argument is `Null`.

name	type	behaviour
list2bag	$(List\ a) \rightarrow (Bag\ a)$	converts a list to a bag
bag2list	$(Bag\ a) \rightarrow (List\ a)$	converts a bag to a list, producing the output in sorted order according to $>$
list2set	$(List\ a) \rightarrow (Set\ a)$	converts a list to a set, removing duplicates
set2list	$(Set\ a) \rightarrow (List\ a)$	converts a set to a list, producing the output in sorted order according to $>$
bag2set	$(Bag\ a) \rightarrow (Set\ a)$	converts a bag to a set, removing duplicates
set2bag	$(Set\ a) \rightarrow (Bag\ a)$	converts a set to a bag

Each of the conversion functions for collections returns `Null/Void/Any` if its argument is `Null/Void/Any`.

A.7 Projection over Tuples

name	type(s)	behaviour
get1	$(Product1\ a) -> a$	returns the first element of a singleton
get1	$(Product2\ a\ b) -> a$	returns the first element of a pair
get1	$(Product3\ a\ b\ c) -> a$	returns the first element of a triple
...	...	...
get2	$(Product2\ a\ b) -> b$	returns the second element of a pair
get2	$(Product3\ a\ b\ c) -> b$	returns the second element of a triple
get2	$(Product4\ a\ b\ c\ d) -> b$	returns the second element of a 4-tuple
...	...	...

These functions return `Null` if their argument is `Null`. geti has been implemented up to $i = 5$, but is easily extendable to the tuple size limit, which is currently 40.

A.8 String Functions

name	type(s)	behaviour
concat	$(List\ String) \rightarrow String$	returns the concatenation of the strings of the input list
split	$String \rightarrow String \rightarrow (List\ String)$	splits the first string argument around matches of the regular expression defined by the second string argument and returns a list containing the e.g. (split 'boo:and:foo' ':') returns ['boo','and','foo']
indexOf	$String \rightarrow Integer$	returns the index within the first string argument of the first occurrence of the second string argument; e.g. (indexOf 'AutoMed' 'M') returns 5
lastIndexOf	$String \rightarrow Integer$	returns the index within the first string argument of the rightmost occurrence of the second string argument; e.g. (indexOf 'Birkbeck' 'k') returns 8
length	$String \rightarrow Integer$	returns the length of the input string argument
lowerCase	$String \rightarrow String$	converts all of the characters of the input string argument to lower case
upperCase	$String \rightarrow String$	converts all of the characters of the input string argument to upper case
substring	$String \rightarrow Integer \rightarrow Integer \rightarrow String$	returns a substring of the input string, based on the input indexes; e.g. (substring 'AutoMed' 5 7) returns 'Me'

Each of the string functions returns `Null` if any of its arguments is `Null`. Also, `concat` returns `Null` if its first argument is `Void`.

A.9 Date Functions

name	type(s)	behaviour
getMonth	$String \rightarrow Integer$ $Integer \rightarrow String$	Provides a mapping between number and word representation for months. Possible integer inputs are 1..12 and 01..12. Possible string inputs are 'January'..'December' and 'Jan'..'Dec' (case-insensitive, include single quotes). String output is January..December. Integer output is 01..12.
now	$\rightarrow DateTime$	returns the current date and time as an IQL dateTime value

Each of the date functions returns `Null` if any of its arguments is `Null`, or if given an invalid input, e.g. 15 or 'Janary'.

A.10 Other Functions

name	type(s)	behaviour
id	$a \rightarrow a$	the identity function - returns its argument

B IQL Syntax

B.1 IQL Lexer

```
package uk.ac.bbk.dcs.automed.qproc;
import java_cup.runtime.*;
%%
%class QLexer
%cup
%eofval{
return (new Symbol(QSym.EOF,""));
%eofval}

BANNEDTOKEN = "BCons"|"SCons"
BLSB = "B["
SLSB = "S["
LLSB = "L["
UNDERSCORE = "_"
INTTOKEN = [-]?[0-9]*
FLOATTOKEN = [-]?([0-9]+)(".")([0-9]+)
STRTOKEN = \'([^\']|"'")*\'
INFIXOP = "<>"|"<="|">="|"++"|"--"|"+"|"-"|"*"|"/"|"="|"<"|">"|
          "div"|"mod"
SPECIAL = "Comp"
DATETIMETOKEN = ("dt '")([1-9])([0-9])([0-9])([0-9])("-")
                ([0-1])([0-9])("-")([0-3])([0-9])(" ")
                ([0-2])([0-9])(":")([0-5])([0-9])(":")
                ([0-5])([0-9])("'")
VARTOKEN = [a-z][A-Za-z0-9_$.]*
CONSTOKEN = [A-Z][A-Za-z0-9_$.]*
SYSVARTOKEN = ("$")([A-Za-z0-9_$.]*)
NN_WHITESPACE = [\ \t\b\012]+

"let" { return (new Symbol(QSym.Let)); }
"in"  { return (new Symbol(QSym.In)); }
"equal"   { return (new Symbol(QSym.Equal)); }
";"   { return (new Symbol(QSym.SemiColon)); }
"<-"  { return (new Symbol(QSym.LArrow)); }
"->"  { return (new Symbol(QSym.RArrow)); }
","   { return (new Symbol(QSym.Comma)); }
"|"   { return (new Symbol(QSym.Bar)); }
"<<"   { return (new Symbol(QSym.LDAB)); }
">>"   { return (new Symbol(QSym.RDAB)); }
"["   { return (new Symbol(QSym.LSB)); }
"]"   { return (new Symbol(QSym.RSB)); }
"("   { return (new Symbol(QSym.LRB)); }
")"   { return (new Symbol(QSym.RRB)); }
"lambda" { return (new Symbol(QSym.Lambda)); }
"{"   { return (new Symbol(QSym.LCB)); }
"}"   { return (new Symbol(QSym.RCB)); }
```

```
":"  { return (new Symbol(QSym.Colon)); }

{BANNEDTOKEN} { throw new RuntimeException("Lexical error:
                      token BCons/SCons not allowed."); }
{BLSB} { return (new Symbol(QSym.BLSB,  yytext()  ) ); }
{SLSB} { return (new Symbol(QSym.SLSB,  yytext()  ) ); }
{LLSB} { return (new Symbol(QSym.LLSB,  yytext()  ) ); }
{UNDERSCORE} {return (new Symbol(QSym.UnderScore,yytext()));}
{SPECIAL} { return (new Symbol(QSym.Special,  yytext()  ) ); }
{INFIXOP} { return (new Symbol(QSym.OpToken,  yytext()  ) ); }
{DATETIMETOKEN}
     {return (new Symbol(QSym.DateTimeToken,yytext()));}
{VARTOKEN} { return (new Symbol(QSym.VarToken , yytext())); }
{CONSTOKEN} { return (new Symbol(QSym.ConsToken,yytext())); }
{STRTOKEN} { return (new Symbol(QSym.StrToken , yytext())); }
{INTTOKEN} { return (new Symbol(QSym.IntToken , yytext())); }
{FLOATTOKEN} {return (new Symbol(QSym.FloatToken , yytext()));}
{SYSVARTOKEN} {return (new Symbol(QSym.SysVarToken, yytext()));}
{NN_WHITESPACE} { }
. { throw new
  RuntimeException("Lexical error in column " + yy_buffer_index +
               ": character " + yytext() + " is not allowed."); }
\n { }
\r { }
```

B.2 IQL Parser

```
package uk.ac.bbk.dcs.automed.qproc;
import java.util.*;

terminal VarToken, Special, ConsToken, StrToken, SysVarToken,
OpToken, DateTimeToken, IntToken, FloatToken, Let, Equal, In,
SemiColon, LArrow, RArrow, Comma, Bar, LLSB,BLSB,SLSB,LSB, RSB,
LRB, RRB, LCB, RCB, Lambda, LDAB, RDAB, Colon, UnderScore;

non terminal query, expr, seq, quals, qual, prefix_op, scheme,
scheme_seq, scheme_element, schemePrefix, schemeSuffix,
schemaName, modelName, consName, VarOrCons;

precedence left IntToken, Let, Equal, In, LArrow, RArrow,
SemiColon, Comma, Bar, LLSB,BLSB,SLSB,LSB, RSB, LRB, RRB,
LCB, RCB, Lambda, VarToken, StrToken;

precedence left OpToken;

query ::=   expr:e1
            {: RESULT =  e1; :}
          | Let VarToken:e1 Equal query:e2 In query:e3
            {: RESULT = new Cell( new Cell( new Cell(
                 new Cell(Cell.TAG_SPECIAL, "Let"),
```

```
                    new Cell(Cell.TAG_VARIABLE, (String)(e1)) ),
                    e2), e3);
          :}
        | query:e1 OpToken:e2 query:e3
          {:
            String op = "("+(String)(e2)+")";
            RESULT = new Cell( new Cell(
               new Cell(Cell.TAG_FUNCTION,op), e1), e3);
          :}
        | query:e1 expr:e2
          {: RESULT = new Cell(e1, e2);
          :}
        | query:e1 RArrow query:e2
          {:
             RESULT = new Cell( new Cell(
               new Cell(Cell.TAG_CONSTRUCTOR,"->"), e1), e2);
          :}
        ;

expr ::=   Special:e1
           {: RESULT = new Cell(Cell.TAG_SPECIAL,(String)e1); :}
         | DateTimeToken:e1
           {: RESULT = new Cell(Cell.TAG_DATETIME,(String)e1); :}
         | IntToken:e1
           {: RESULT = new Cell(Cell.TAG_INTEGER,
                       new Integer(((String)e1)).intValue() ); :}
         | FloatToken:e1
           {: RESULT = new Cell(Cell.TAG_FLOAT,
                  new Float(((String)e1+'f')).floatValue() ); :}
         | StrToken:e1
           {: RESULT = new Cell(Cell.TAG_STRING, (String)e1); :}
         | ConsToken:e1
           {: RESULT=new Cell(Cell.TAG_CONSTRUCTOR,(String)e1);:}
         | VarToken:e1
           {: RESULT = new Cell((String)e1); :}
         | SysVarToken:e1
           {: RESULT=new Cell(Cell.TAG_SYSVARIABLE,(String)e1);:}
         | scheme:e1
           {: RESULT = new Cell(Cell.TAG_SCHEME,
                                new SchemeInfo((String)e1)); :}
         | schemePrefix:e1 scheme:e2 schemeSuffix:e3
           {:
             String prefix = (String)e1;
             String schemaName =
                    prefix.substring(prefix.indexOf(":")+1,
                    prefix.indexOf(":",prefix.indexOf(":")+1));
             String modelName =
                    prefix.substring(prefix.indexOf(":",1)+1,
                    prefix.indexOf(":",prefix.indexOf(":",1)+1));
             String consName =
```

```
            prefix.substring(prefix.indexOf(":",
            prefix.indexOf(":",prefix.indexOf(":",1)+1))+1,
                           prefix.lastIndexOf(":"));
      String type = (String)e3; //syntax is ('type')
      SchemeInfo si = new SchemeInfo((String)e2);
      if(!schemaName.equals(""))
        si.setSchema(schemaName);
      if(!modelName.equals(""))
        si.setModelName(modelName);
      if(!consName.equals(""))
        si.setConstructName(consName);
      if(!type.equals(""))
        si.setType(type);
      RESULT = new Cell(Cell.TAG_SCHEME, si);
    :}
  | LSB query:e1 Bar quals:e2 RSB
    {:
       Cell n = (Cell)(e1);
       ArrayList list = (ArrayList)(e2);
       for(int i=list.size()-1; i>=0; i--) {
         n = new Cell( new Cell(
              new Cell(Cell.TAG_SPECIAL,"LComp"),
              list.get(i) ), n );
       }
       RESULT = n;
        // RESULT = new Cell( new Cell( new Cell("LComp"),
       // e1), (ASG.toASGList((ArrayList)(e2))).root());
    :}
  | LLSB query:e1 Bar quals:e2 RSB
    {:
       Cell n = (Cell)(e1);
       ArrayList list = (ArrayList)(e2);
       for(int i=list.size()-1; i>=0; i--) {
         n = new Cell( new Cell(
              new Cell(Cell.TAG_SPECIAL,"LComp"),
              list.get(i) ), n );
       }
       RESULT = n;
        // RESULT = new Cell( new Cell( new Cell("LComp"),
       //   e1),(ASG.toASGList((ArrayList)(e2))).root());
    :}
  | SLSB query:e1 Bar quals:e2 RSB
    {:
       Cell n = (Cell)(e1);
       ArrayList list = (ArrayList)(e2);
       for(int i=list.size()-1; i>=0; i--) {
         n = new Cell( new Cell(
              new Cell(Cell.TAG_SPECIAL,"SComp"),
              list.get(i) ), n );
       }
```

```
                RESULT = n;
                 // RESULT = new Cell( new Cell( new Cell("SComp"),
                //   e1), (ASG.toASGList((ArrayList)(e2))).root() );
              :}
            | BLSB query:e1 Bar quals:e2 RSB
              {:
                Cell n = (Cell)(e1);
                ArrayList list = (ArrayList)(e2);
                for(int i=list.size()-1; i>=0; i--)
                {
                  n = new Cell( new Cell(
                           new Cell(Cell.TAG_SPECIAL,"BComp"),
                           list.get(i) ), n );
                }
                RESULT = n;
                 // RESULT = new Cell( new Cell( new Cell("BComp"),
                //   e1), (ASG.toASGList((ArrayList)(e2))).root() );
              :}
            | LSB RSB {: RESULT = ASG.emptyList().root(); :}
            | LLSB RSB {: RESULT = ASG.emptyList().root(); :}
            | SLSB RSB {: RESULT = ASG.emptySet().root(); :}
            | BLSB RSB {: RESULT = ASG.emptyBag().root(); :}
            | LSB seq:e1 RSB
              {:
                RESULT = (ASG.toASGList((ArrayList)(e1))).root();
              :}
            | LCB seq:e1 RCB
              {:
                RESULT = (ASG.toASGTuple((ArrayList)(e1))).root();
              :}
            | LRB query:e1 RRB {: RESULT = e1; :}
            | Lambda expr:e1 expr:e2
              {: RESULT = new Cell(Cell.TAG_LAMBDA,e1,e2);
              :}
            | prefix_op:e1
              {: RESULT = new Cell(Cell.TAG_FUNCTION,(String)e1); :}
            ;

seq ::=     seq:e1 Comma query:e2
            {:
              ((ArrayList)(e1)).add(e2);
              RESULT = e1;
            :}
          | query:e1
            {:
              ArrayList alist = new ArrayList();
              alist.add(e1);
              RESULT = alist;
            :}
          ;
```

```
quals ::=   qual:e1 SemiColon quals:e2
            {:
              ((ArrayList)(e2)).add(0,e1);
              RESULT = e2;
            :}
          | qual:e1
            {:
              ArrayList alist = new ArrayList();
              alist.add(e1);
              RESULT = alist;
            :}
          ;

qual ::=    query:e1
            {: RESULT = e1; :}
          | query:e1 LArrow query:e2
            {: RESULT = new Cell( new Cell(
                          new Cell(Cell.TAG_CONSTRUCTOR, "Gen"),
                 e1), e2); :}
          ;

scheme ::= LDAB scheme_seq:e1 RDAB
               {: RESULT = "<<"+e1+">>"; :}
               ;

scheme_seq ::=   scheme_element:e1
                 {: RESULT = e1; :}
               | scheme_seq:e1 Comma scheme_element:e2
                 {:
                   RESULT = e1+","+e2;
                 :}
                 ;

scheme_element ::=   UnderScore:e1 {: RESULT = e1; :}
                   | VarToken:e1 {: RESULT = e1; :}
                   | StrToken:e1 {: RESULT = e1; :}
                   | IntToken:e1 {: RESULT = e1; :}
                   | ConsToken:e1 {: RESULT = e1; :}
                   | SysVarToken:e1 {: RESULT = e1; :}
                   | scheme:e1 {: RESULT = e1; :}
                     ;

schemePrefix ::=   Colon schemaName:e1 Colon modelName:e2
                                       Colon consName:e3 Colon
                   {: RESULT = ":"+(String)e1+":"+(String)e2
                                +":"+(String)e3+":"; :}
                 | Colon schemaName:e1 Colon modelName:e2 Colon
                                                         Colon
```

```
                    {: RESULT = ":"+(String)e1+":"+(String)e2
                                  +":"+":"; :}
                  | Colon schemaName:e1 Colon Colon consName:e2
                                                          Colon
                    {: RESULT = ":"+(String)e1+":"+":"
                                  +(String)e2+":"; :}
                  | Colon Colon modelName:e1 Colon consName:e2
                                                         Colon
                    {: RESULT = ":"+":"+(String)e1+":"
                                  +(String)e2+":"; :}
                  | Colon schemaName:e1 Colon Colon Colon
                    {:
                      RESULT = ":"+(String)e1+":"+":"+":";
                    :}
                  | Colon Colon modelName:e1 Colon Colon
                    {:
                      RESULT = ":"+":"+(String)e1+":"+":";
                    :}
                  | Colon Colon Colon consName:e1 Colon
                    {:
                      RESULT = ":"+":"+":"+(String)e1+":";
                    :}
                  | Colon Colon Colon Colon
                    {:
                      RESULT = "::::";
                    :}
                    ;

schemeSuffix ::= Colon {: RESULT = ""; :}
                 | Colon StrToken:e1 {: RESULT = e1; :}
                 ;

schemaName ::= VarOrCons:e1 {: RESULT = e1; :} ;

modelName ::= VarOrCons:e1 {: RESULT = e1; :} ;

consName ::= VarOrCons:e1 {: RESULT = e1; :} ;

VarOrCons ::=   VarToken:e1 {: RESULT = e1; :}
              | ConsToken:e1 {: RESULT = e1; :}
               ;

prefix_op      ::=   LRB OpToken:e1 RRB
                     {:
                       StringBuffer sb = new StringBuffer();
                       String s = "("+((String)(e1))+")";
                       RESULT =s;
                     :}
                     ;
```

C Wrapper JavaCC grammars

C.1 Schemes.jj

```
/* JavaCC grammar defining schemes. */

options { STATIC = false; }

PARSER_BEGIN(Schemes)

package uk.ac.bbk.dcs.automed.qproc.annotate.grammars.schemes;

import uk.ac.bbk.dcs.automed.qproc.annotate.grammars.QueryParser;

public class Schemes extends QueryParser { }

PARSER_END(Schemes)

SKIP : { " " | "\t" | "\n" | "\r" }

TOKEN : {
  < VARTOKEN: ["a"-"z"] ( ["a"-"z","A"-"Z","0"-"9","_","$","."] )* >  |
  < CONSTOKEN: ["A"-"Z"] ( ["a"-"z","A"-"Z","0"-"9","_","$","."] )* > |
  < STRTOKEN: ["'"] (~["'"])* ["'"] >                    |
  < NUMTOKEN: (["0"-"9"])+ | (["0"-"9"])+ ["."] (["0"-"9"])+ >      |
  < COMMA: "," >
}

void parse() : {   } {
  simpleScheme()
}

void simpleScheme() : {   } {
  pureScheme() | (":" [schemaName()] ":" [modelName()] ":"
                [consName()] ":" pureScheme() ":" [type()])
}

void pureScheme() : {   } {
  "<<" schemeSeq() ">>"
}

void schemaName() : {   } {
  varOrCons()
}

void modelName() : {    } {
  varOrCons()
}

void consName() : { } {
  varOrCons()
}

void type() : { } {
```

```
  <STRTOKEN>
}

void varOrCons() : {    } {
  <VARTOKEN> | <CONSTOKEN>
}

void schemeSeq() : {   } {
  schemeElement() [ <COMMA> schemeSeq() ]
}

void schemeElement() : {   } {
   <VARTOKEN> [ ":" <VARTOKEN> ] | <STRTOKEN> | <NUMTOKEN>
}
```

C.2 SimpleCompAppend.jj

```
/* JavaCC grammar defining simple comprehensions,
 * also supporting simple/nested append operator. */

options { STATIC = false; }

PARSER_BEGIN(SimpleCompAppend)

package uk.ac.bbk.dcs.automed.qproc.annotate.grammars.simpleCompAppend;

import uk.ac.bbk.dcs.automed.qproc.annotate.grammars.QueryParser;

public class SimpleCompAppend extends QueryParser { }

PARSER_END(SimpleCompAppend)

SKIP : { " " | "\t" | "\n" | "\r" }

TOKEN : {
 < COMP: "[" >     |
 < LCOMP: "L[" >   |
 < SCOMP: "S[" >   |
 < BCOMP: "B[" >   |
 < VARTOKEN: ["a"-"z"] ( ["a"-"z","A"-"Z","0"-"9","_","$","."] )* >    |
 < CONSTOKEN: ["A"-"Z"] (["a"-"z","A"-"Z","0"-"9","_","$","."]|"%"])* > |
 < WRAPPERTOKEN: "$wrapper" >      |
 < SYSVARTOKEN: "$" ( ["a"-"z","A"-"Z","0"-"9","_","$"] )* >       |
 < STRTOKEN: ["'"] (~["'"])* ["'"] >                    |
 < NUMTOKEN: (["0"-"9"])+ | (["0"-"9"])+ ["."] (["0"-"9"])+ >      |
 < COMPARISONTOKEN: "(=)"|"(<>)"|"(>)"|"(<)"|"(>=)"|"(<=)"|
                    "(div)"|"(mod)" > |
 < BOOLTOKEN: "True" | "False">                     |
 < COMMA: "," >    |
 < APPEND: "(++)">
}

void parse() : {   } {
  simpleSchemeOrSimpleComp() | append()
```

```
}

void append() : {  } {
  "(" <APPEND> ( simpleSchemeOrSimpleComp() | append())
               ( simpleSchemeOrSimpleComp() | append()) ")"
}

void simpleSchemeOrSimpleComp() : { } {
  simpleScheme() | simpleComprehension()
}

void simpleComprehension() : {   } {
  (<COMP>|<LCOMP>|<BCOMP>|<SCOMP>) simpleTuple() "|" simpleQuals() "]"
}

void constant() : {   } {
  <STRTOKEN> | <NUMTOKEN> | <BOOLTOKEN>
}

void simpleTuple() : {   } {
  ["{"] seqOfVars() ["}"]
}

void variable() : {} {
  <VARTOKEN> | <SYSVARTOKEN>
}

void seqOfVars() : {   } {
  variable() ( <COMMA> variable() )*
}

void simpleQuals() : {   } {
  ( simpleFilter()";" )? simpleGenerator()
               ( ";" ( simpleGenerator() | simpleFilter() ) )*
}

void simpleGenerator() : {   } {
  simpleTuple() "<-"  ( simpleScheme() | simpleList() )
}

void simpleFilter() : {   } {
   "(" <COMPARISONTOKEN> ( constant() | variable())
                         ( constant() | variable()) ")"
}

void simpleList() : {   } {
  "[" simpleListTokens() "]"
}

void simpleListTokens() : {   } {
  ( ( constant() ( <COMMA> constant() )* ) |
  ( listTuple() ( <COMMA> listTuple() )* ) ) ?
}
```

```
void listTuple() : {   } {
  ["{"] constant() ( <COMMA> constant() )* ["}"]
}

void simpleScheme() : {   } {
  pureScheme() | (":" [schemaName()] ":" [modelName()] ":"
                      [consName()] ":" pureScheme() ":" [type()])
}

void pureScheme() : {   } {
  "<<" schemeSeq() ">>"
}

void schemaName() : {   } {
  varOrCons()
}

void modelName() : {    } {
  varOrCons()
}

void consName() : { } {
  varOrCons()
}

void type() : { } {
  <STRTOKEN>
}

void varOrCons() : {    } {
  <VARTOKEN> | <CONSTOKEN>
}

void schemeSeq() : {   } {
  schemeElement() [ <COMMA> schemeSeq() ]
}

void schemeElement() : {   } {
   ( <VARTOKEN> | <CONSTOKEN> )
       [ ":" ( <VARTOKEN> | <CONSTOKEN> ) ] | <STRTOKEN> | <NUMTOKEN>
}
```

C.3 IQLforSQL.jj

```
/* JavaCC grammar for BBKSQLWrapper */

options {
STATIC = false;
DEBUG_PARSER = false;
DEBUG_TOKEN_MANAGER = false;
}

PARSER_BEGIN(IQLForSQLQueryParser)
```

```
package uk.ac.bbk.dcs.automed.qproc.annotate.grammars.sql;

import uk.ac.bbk.dcs.automed.qproc.annotate.grammars.QueryParser;

public class IQLForSQLQueryParser extends QueryParser {
}

PARSER_END(IQLForSQLQueryParser)

SKIP : { " " | "\t" | "\n" | "\r" }

TOKEN : {
 < LSB: "[" >   |
 < LCOMP: "L[" >   |
 < SCOMP: "S[" >   |
 < BCOMP: "B[" >   |
 < DISTINCT: "distinct" >   |
 < AGGREGATETOKEN: "count" | "sum" | "avg" | "max" | "min" > |
 < COLLECTION: "(++)" | "union" >|
 < CONCATTOKEN: "concat" > |
 < BOOLTOKEN: "True" | "False">|
 < TOSTRINGTOKEN: "toString" > |
 < COMPARISONTOKEN: "(=)" | "(<>)" | "(>)" | "(<)" | "(>=)" | "(<=)" > |
 < MEMBERTOKEN: "member" > |
 < LIKETOKEN: "like" > |
 < WRAPPERTOKEN: "$wrapper" >    |
 < STRTOKEN: ["'"] (~["'"])* ["'"] >|
 < NUMTOKEN: (["0"-"9"])+ | (["0"-"9"])+ ["."] (["0"-"9"])+ > |
 < LRB: "("> |
 < RRB: ")"> |
 < COMMA: "," >  |
 < VARTOKEN: ["a"-"z"] ( ["a"-"z","A"-"Z","0"-"9","_","$","."] )* > |
 < NULLTOKEN: "Null" >    |
 < CONSTOKEN: ["A"-"Z"] (["a"-"z","A"-"Z","0"-"9","_","$","."," %"])* > |
 < SYSVARTOKEN: "$" ( ["a"-"z","A"-"Z","0"-"9","_","$"] )* >
}

void parse() : {   } {
  query()
  |
  ( <LRB> query() <RRB> )
}

void query() : {   } {
  simpleSchemeOrComp() | collectionOrAggregate()
}

void simpleSchemeOrComp() : {   } {
  simpleScheme() | comprehension()
}

void collectionOrAggregate() : {   } {
  (collection() | aggregate())
}
```

```
void aggregate() : {   } {
  <AGGREGATETOKEN>
  ( LOOKAHEAD(2)
    ( <LRB> comprehension() <RRB> )
    |
    comprehension()
    |
    ( <LRB> collection() <RRB> )
    |
    collection()
  )
}

void collection() : {  } {
  <COLLECTION> ( collection() |
                 ( simpleSchemeOrComp() simpleSchemeOrComp() ) )
}

void comprehension() : {   } {
  ( (<LSB> | <LCOMP> | <BCOMP> | <SCOMP>)
        ( nestedTuple() | varOrSysVar() | concat() ) "|" quals() "]" )
  |
  ( <DISTINCT> (<LSB> | <LCOMP> | <BCOMP> | <SCOMP>)
        ( nestedTuple() | varOrSysVar() | concat() ) "|" quals() "]" )
}

void subComp() : {   } {
  comprehension()
  |
  ( <LRB> comprehension() <RRB> )
}

////////////////////
// Filters syntax //
////////////////////

void filter() : {   } {
   <LRB> ( comparisonFilter() | likeFilter() | concatFilter() ) <RRB>
}

void comparisonFilter() : {   } {
    <COMPARISONTOKEN>
    (
      (
      (constant() | varOrSysVar() | filter())
                (constant() | varOrSysVar() | filter() | <NULLTOKEN>)
     ) |
     (
       (simpleTuple()) (simpleTuple() | <NULLTOKEN>)
     )
   )
}
```

```
void memberFilter() : {   } {
<MEMBERTOKEN> ( LOOKAHEAD(40) simpleSchemeOrComp() | nestedList() )
                                ( varOrSysVarOrCons() | simpleTuple())
}

void likeFilter() : {   } {
<LIKETOKEN> varOrSysVar() <STRTOKEN>
}

void concatFilter() : {   } {
( <CONCATTOKEN> <LSB> ( varOrSysVarOrCons() | toStringMethod() )
     ( <COMMA>  ( varOrSysVarOrCons() | toStringMethod() ) )* "]" )
}

//////////////////////
// Qualifiers syntax //
//////////////////////

void quals() : {   } {
  ( filter()";" )? generator() ( ";" ( generator() | filter() ) )*
}

void generator() : {   } {
  ( varOrSysVar() | nestedTuple() ) "<-"
        ( simpleScheme() | ( LOOKAHEAD(60) subComp() | nestedList() ) )
}

//////////////////
// Tuples syntax //
//////////////////

void nestedTuple() : {   } {
  "{" nestedTupleSeq() "}"
}

void nestedTupleSeq() : {   } {
  seqOfVarsAndTuples()
}

void seqOfVarsAndTuples() : {   } {
  patternTupleItem() ( <COMMA> patternTupleItem() )*
}

void patternTupleItem() : {   } {
  constant() | varOrSysVar() | nestedTuple() | concat()
}

void simpleTuple() : {   } {
  "{" simpleTupleSeq() "}"
}

void simpleTupleSeq() : {   } {
  (varOrSysVar() | constant())
    ( <COMMA> (varOrSysVar() | constant()) )*
```

```
}

void concat() : {   } {
  ( <LRB> )?
    <CONCATTOKEN> <LSB> ( varOrSysVarOrCons() | toStringMethod() )
    ( <COMMA>  ( varOrSysVarOrCons() | toStringMethod() ) )* "]"
  ( <RRB> )?
}

void toStringMethod() : {   } {
  ( <LRB> )? <TOSTRINGTOKEN> varOrSysVar() ( <RRB> )?
}

////////////////////////
// Nested List syntax //
////////////////////////

void nestedList() : {   } {
  "[" nestedListTokens() "]"
}

void nestedListTokens() : {   } {
  ( ( constant() ( <COMMA> constant() )* ) | ( nestedListTuple()
    ( <COMMA> nestedListTuple() )* ) ) ?
}

void nestedListTuple() : {   } {
  "{" nestedListTupleItem() ( <COMMA> nestedListTupleItem() )* "}"
}

void nestedListTupleItem() : {   } {
  constant() | nestedTuple()
}

//////////////////////////
// Simple scheme syntax //
//////////////////////////

void simpleScheme() : {   } {
  pureScheme() | (":" [peerName()] ":" [schemaName()] ":" [modelName()]
                  ":" [consName()] ":" pureScheme() ":" [type()])
}

void pureScheme() : {   } {
  "<<" schemeSeq() ">>"
}

void peerName() : { } {
  varOrCons()
}

void schemaName() : { } {
  varOrCons()
}
```

```
void modelName() : { } {
  varOrCons()
}

void consName() : { } {
  varOrCons()
}

void type() : { } {
  <STRTOKEN>
}

void varOrCons() : { } {
  <VARTOKEN> | <CONSTOKEN>
}

void schemeSeq() : {   } {
  schemeElement() [ <COMMA> schemeSeq() ]
}

void schemeElement() : {   } {
   pureScheme()|"_"|<VARTOKEN>|<CONSTOKEN>|<STRTOKEN>|<NUMTOKEN>
}

////////////
// Other //
////////////

void varOrSysVarOrCons() : {} {
  <VARTOKEN> | <SYSVARTOKEN> | constant()
}

void varOrSysVar() : {} {
  <VARTOKEN> | <SYSVARTOKEN>
}

void constant() : {   } {
  <STRTOKEN> | <NUMTOKEN> | <BOOLTOKEN>
}
```