The Automed Intermediate Query Language
Automed Working Document 2, Version 1

Alexandra Poulovassilis
Dept. of Computer Science, Birkbeck College
Malet Street, London WC1E 7HX

June 28, 2001

1 The Language

The Automed Intermediate Query Language (AIQL) is a typed, functional language [1]. It supports a number of primitive types, such as booleans, strings and numbers, as well as product, function, set, bag and list types. In particular, $(t_1, \ldots, t_n)$ is an n-product type for any types $t_1, \ldots, t_n$, *List* t is a list type for any type t, *Bag* t is a bag type for any type t, *Set* t is a set type for any type t, and $t_1 \to t_2$ is the type of functions from t_1 to t_2.

Enumerated sets, bags and lists are delimited by $\{\ldots\}$, $\{|\ldots|\}$ and $[\ldots]$ brackets, respectively. The set-union, bag-union and list-append operators are $union$, $bagUnion$ and $++$[1].

The usual arithmetic and comparison operators are built into AIQL. The 3-argument conditional function if is also built-in. In this document, we sometimes use $if\ e_1\ then\ e_2\ else\ e_3$ as syntactic sugar for $if\ e_1\ e_2\ e_3$. The negation operator *not* is defined in terms of if, as are the conjunction and disjunction operators *and* and *or* (see [8, 9] for details of these definitions)[2]. So these operators are not built-in, but for our purposes here they can be regarded as if they were.

In this document we use the notation $e :: t$ to indicate that an expression e has type t. We use letters from the start of the alphabet to indicate type variables in type expressions.

2 Collection types

A *collection type* is one constructed by a unary type name, T, for which there exist a constant $Empty :: T\ a$ and two functions

$$\begin{array}{lll} single & :: & a \to T\ a \\ \_ \oplus \_ & :: & T\ a \to T\ a \to T\ a \end{array}$$

such that $\oplus$ has $Empty$ as a left and right identity.

AIQL's set, bag and list types are collection types:

- For sets, $T = Set$, $Empty = \{\}$, $\oplus = union$ and $single = \lambda x.\{x\}$.
- For bags, $T = Bag$, $Empty = \{||\}$, $\oplus = bagUnion$ and $single = \lambda x.\{|x|\}$.
- For lists, $T = List$, $Empty = [\,]$, $\oplus = ++$ and $single = \lambda x.[x]$.

For each collection type, T, AIQL also supports a built-in operator, $fold :: (a \to b) \to (b \to b \to b) \to b \to T\ a \to b$, which 'folds' a binary function *op* into a collection and has the following semantics[3]:

[1] $union$ and $bagUnion$ are called $\cup$ and $\cup_B$ in [8, 9].
[2] *and* and *or* correspond to the $\wedge$ and $\vee$ operators defined in [8, 9].
[3] $fold$ corresponds to the ϕ operator of [8, 9].

$$
\begin{aligned}
fold\ f\ op\ e\ \bot &= \bot \\
fold\ f\ op\ e\ Empty &= e \\
fold\ f\ op\ e\ (single\ e') &= f\ e' \\
fold\ f\ op\ e\ (s\ \oplus\ s') &= op\ (fold\ f\ op\ e\ s)\ (fold\ f\ op\ e\ s')
\end{aligned}
$$

Note that the operator name $fold$ is overloaded, there being in fact a family of $fold_T$ functions, one for each collection type T. The type-checker is able to distinguish which one is intended from the collection type of the 4th argument passed to $fold$.

In order for $fold$ to be well-defined, there are some provisos on the parameters op and e that can be passed to it:

- In order for an application $fold\ f\ op\ e\ s$ to be well-defined for any set s, op must be commutative, associative and idempotent, and must have e as a left and right identity.
- In order for $fold\ f\ op\ e\ b$ to be well-defined for any bag b, op must be commutative and associative, and must have e as a left and right identity.
- In order for $fold\ f\ op\ e\ l$ to be well-defined for any list l, op must be associative, and must have e as a left and right identity.

3 Converting between collections

We can use $fold$ to define a conversion function from lists to sets:

$$list\_to\_set \quad = fold\ (\lambda x.\{x\})\ (union)\ \{\}$$

and also a conversion function from sets to lists:

$$set\_to\_list \quad = fold\ (\lambda x.[x])\ (\lambda xs.\lambda ys.remove\_duplicates\ (sort\ (xs ++ ys)))\ [\,]$$

We can also convert between bags and lists:

$$
\begin{aligned}
list\_to\_bag \quad &= fold\ (\lambda x.\{|x|\})\ (bagUnion)\ \{|\,|\} \\
bag\_to\_list \quad &= fold\ (\lambda x.[x])\ (\lambda xs.\lambda ys.(sort\ (xs ++ ys)))\ [\,]
\end{aligned}
$$

and between bags and sets:

$$
\begin{aligned}
bag\_to\_set \quad &= fold\ (\lambda x.\{x\})\ (union)\ \{\} \\
set\_to\_bag \quad &= fold\ (\lambda x.\{|x|\})\ (\lambda xs.\lambda ys.unique\ (xs\ bagUnion\ ys))\ \{|\,|\}
\end{aligned}
$$

Here, $unique$ is a duplicate-removing operation on bags (see [8] for details of how this is defined).

4 Iteration operations

$fold_T$ can be used to define a function $flatmap_T$ that applies a function f of type $a \to T\ b$ to each element of a collection s of type $T\ a$ and merges the results using $\oplus_T$:

$$
\begin{aligned}
flatmap_T &:: (a \to T\ b) \to T\ a \to T\ b \\
flatmap_T\ f\ s &= fold_T\ f\ \oplus_T\ Empty_T\ s
\end{aligned}
$$

Using $flatmap_T$, functions map_T, $filter_T$ and $join_T$ can be defined which generalise relational projection, selection and join operations, respectively:

$$
\begin{aligned}
map_T &:: (a \to b) \to T\ a \to T\ b \\
map_T\ f\ s &= flatmap_T\ (single_T \circ f)\ s
\end{aligned}
$$

$$
\begin{aligned}
filter_T &:: (a \to Bool) \to T\ a \to T\ a \\
filter_T\ f\ s &= flatmap_T\ (\lambda x.if\ (f\ x)\ then\ (single_T\ x)\ else\ Empty_T)\ s
\end{aligned}
$$

$$
\begin{aligned}
join_T &:: (a \to b \to c) \to T\ a \to T\ b \to T\ c \\
join_T\ f\ r\ s &= flatmap_T\ (\lambda x.flatmap_T\ (\lambda y.f\ x\ y)\ s)\ r
\end{aligned}
$$

Again, the function names $flatmap$, map, $filter$ and $join$ can be overloaded in practice, and the type-checker is able to distinguish which one is intended from the collection type of arguments passed to these functions.

5 Aggregation operations

Using $fold$, the maximum and minimum element of a collection s can be found, where id is the identity function $\lambda x.x$:

$$\begin{array}{lcl} max\ s &=& fold\ \ id\ \ (\lambda x.\lambda y.if\ (x > y)\ x\ y)\ \ minNum\ \ s \\ min\ s &=& fold\ \ id\ \ (\lambda x.\lambda y.if\ (x < y)\ x\ y)\ \ maxNum\ \ s \end{array}$$

Cardinality and summation functions can be defined over bags and lists (but not directly over sets since $+$ is not idempotent):

$$\begin{array}{lcl} count\ s &=& fold\ \ (\lambda x.1)\ \ (+)\ \ 0\ \ s \\ sum\ s &=& fold\ \ id\ \ (+)\ \ 0\ \ s \end{array}$$

Sets can be counted or summed by first converting them to bags and then applying $count$ or sum. A function avg can also be defined in terms of $fold$:

$$\begin{array}{lcl} avg\ s &=& let\ (sum, count) = fold\ \ (\lambda x.(x,1))\ \ (\lambda(x1,y1).\lambda(x2,y2).(x1+x2, y1+y2))\ \ (0,0)\ \ s \\ && in\ \ sum/count \end{array}$$

We can use $fold$ to define a membership operator, in, over collections:

$$x\ in\ s \ =\ fold\ \ ((=)\ x)\ \ (or)\ \ False\ \ s$$

More generally, $fold$ can be used to define an existential quantifier over collections, where p is a predicate:

$$some\ p\ s \ =\ fold\ \ p\ \ (or)\ \ False\ \ s$$

Similarly, a universal quantifier:

$$all\ p\ s \ =\ fold\ \ p\ \ (and)\ \ True\ \ s$$

6 Grouping operations

A grouping operation

$$group\_T \ ::\ T\ (a,b) \rightarrow T\ (a, T\ b)$$

can be defined in terms of $fold$ — see [9] for details.

Note that $group$ is more restrictive than GROUP BY in SQL since it only operates on collections of pairs. But AIQL supports nested tuples, projection over the components thereof, and tuple construction (see, for example, the query in Section 7.1) so this single operation suffices to capture GROUP BY.

Using $group$, a generic "group-and-compute" function, gc, can be defined that takes an aggregation function, $aggr$, as an argument:

$$gc\ aggr\ s \ =\ map\ (\lambda(x, ys).(x, aggr\ ys))\ (group\ s)$$

This operation captures aggregation over GROUP BY in SQL.

Note that SQL's $ORDER\ BY$ and $DISTINCT$ operations can be captured by converting the set or bag to a list and applying to it the *sort* and *remove_duplicates* functions, respectively.

7 Comprehensions

The syntax of *comprehensions* [2] is

$$[e \mid Q_1;\ \ldots;\ Q_n]$$

e is an expression, termed the *head* of the comprehension. Q_1 to Q_n are *qualifiers*, where $n \geq 0$. Each qualifier is either a *filter* or a *generator*. A filter is a boolean-valued expression. A generator has syntax $p \leftarrow s$, where p is a pattern and s is a collection-valued expression.

For the moment, we can assume that the collections in the generators of a comprehension all have the same collection type, T say (it is however possible to relax this and allow iteration over different collection types under certain conditions — see [9] for details). A comprehension then returns a collection of type $T\ t$, where t is the type of its head expression.

Comprehensions add no extra expressiveness to the language as they can be translated into applications of $flatmap$ and if:

$$\begin{array}{lcl} [e \mid p \leftarrow s; Q] & \equiv & flatmap_T\ (\lambda p.[e \mid Q])\ s \\ [e \mid e'; Q] & \equiv & if\ e'\ then\ [e \mid Q]\ else\ Empty_T \\ [e \mid\ ] & \equiv & single_T\ e \end{array}$$

However, comprehensions give a more useful intermediate query formalism than the lower-level $flatmap$ and if operations because (a) they can directly represent the common iteration operations of joining, selecting and projecting (see for example the query in Section 7.1), and (b) there is a rich set of algebraic equivalences that has been developed for comprehensions that can be used to optimise them (see [8, 9, 6, 5] for a discussion).

7.1 An Example Comprehension Query

Consider a database containing information about music sales. The following extensionally-defined collections are maintained in the database:

$$\begin{array}{lcl} categories & :: & Bag\ (Category, Group) \\ recordings & :: & Bag\ (RecNo, Group, Margin, Sales) \end{array}$$

These give for each group with whom the company has a contract, the category (e.g. Folk, Rock, Jazz) of the group, and for each recording a recording number, the group that made the recording, the profit margin and the total sales.

Our example query involves finding the total sales per category and group, for those categories which satisfy a stated predicate, p.

A naive formulation of this query is to project the group and sales attributes from *recordings* (in the local definition of $r1$ below), join the result with *categories* (in the local definition of $r2$ below), select the tuples which satisfy p, and then apply *gc sum* to the resulting bag:

$$\begin{array}{lll} let & r1 = & [(g,s) \mid (r,g,m,s) \leftarrow recordings]; \\ & r2 = & [((c,g),s) \mid (g,s) \leftarrow r1;\ (c,g') \leftarrow categories;\ g = g'] \\ in & \multicolumn{2}{l}{gc\ sum\ [((c,g),s) \mid ((c,g),s) \leftarrow r2;\ p\ c]} \end{array}$$

7.2 Algebraic Optimisation of Comprehension Queries

In the above query, we can push *gc sum* before the selection $p\ c$:

$$\begin{array}{lll} let & r1 = & [(g,s) \mid (r,g,m,s) \leftarrow recordings]; \\ & r2 = & [((c,g),s) \mid (g,s) \leftarrow r1;\ (c,g') \leftarrow categories;\ g = g'] \\ in & \multicolumn{2}{l}{[((c,g),s) \mid ((c,g),s) \leftarrow gc\ sum\ r2;\ p\ c]} \end{array}$$

We can then push *gc sum* into the join, to obtain an expression which projects the group and sales attributes from *recordings*, computes for each group their total sales, joins the result with *categories*, and finally selects the tuples which satisfy p:

$$\begin{array}{lll} let & r1 & = [(g,s) \mid (r,g,m,s) \leftarrow recordings]; \\ & r2 & = [((c,g),s) \mid (g,s) \leftarrow gc\ sum\ r1;\ (c,g') \leftarrow categories;\ g = g'] \\ in & \multicolumn{2}{l}{[((c,g),s) \mid ((c,g),s) \leftarrow r2;\ p\ c]} \end{array}$$

We can then push the filter $(p\ c)$ into the second comprehension and then promote it past $g = g'$:

$$\begin{array}{lll} let & r1 & = [(g,s) \mid (r,g,m,s) \leftarrow recordings]; \\ & r2 & = [((c,g),s) \mid (g,s) \leftarrow gc\ sum\ r1;\ (c,g') \leftarrow categories;\ p\ c;\ g = g'] \\ in & \multicolumn{2}{l}{[((c,g),s) \mid ((c,g),s) \leftarrow r2]} \end{array}$$

Finally, we can eliminate the third comprehension:

$$\begin{array}{lll} let & r1 & = [(g,s) \mid (r,g,m,s) \leftarrow recordings]; \\ & r2 & = [((c,g),s) \mid (g,s) \leftarrow gc\ sum\ r1;\ (c,g') \leftarrow categories;\ p\ c;\ g = g'] \\ in & r2 & \end{array}$$

8 Why a Collections+Comprehensions Language for Automed?

There are three main reasons. Firstly, comprehensions provide a unifying query formalism for relational, OO and deductive languages (see [10, 7, 4, 8, 9, 6]). Informally, the head of a set comprehension corresponds to the SELECT clause of an SQL or an OQL query, the generators correspond to the FROM clause, and the filters to the WHERE clause. We can also express Datalog predicates using comphrehensions. For example, given a set $parent :: Set(Person, Person)$ and a set $mother :: Set(Person, Person)$, the following equations define a set $father$ and a (recursively-defined) set anc:

$$\begin{array}{ll} father & = [t \mid t \leftarrow parent;\ not\ (t\ in\ mother)] \\ anc & = parent\ union\ [(a,d) \mid (a,d') \leftarrow anc;\ (a',d) \leftarrow anc; a' = d'] \end{array}$$

Secondly, there has also been much work on optimising collection and comprehension languages, by us and by others (see [8, 9, 6, 5] for reviews of this work). Thirdly, by supporting the generic $fold$ operator, the language is extensible.

This kind of language is thus a good candidate for an intermediate query language for heterogeneous database architectures.

A further reason for using this language for the Automed project is that this functionality is already available within the PFL functional database prototype. Thus, we may be able to leverage this existing software to provide the Automed global query processing functionality.

Comprehension languages have been used before for querying heterogeneous data sources, for example in the BioKleisli system [3]. What will be novel about our work in Automed will be the query translation process, which will be based on the schema transformations set up by the Schema Transformation and Integration Tool.

References

[1] R.Bird and P.Wadler, *Introduction to functional programming*, Prentice-Hall, 1988.

[2] P.Buneman, L.Libkin, D.Suciu, V.Tannen and L.Wong, Comprehension Syntax, *ACM SIGMOD Record*, 23(1), pp 87-96, 1994.

[3] S.B.Davidson et al, BioKleisli: A Digital Library for Biomedical Researchers, *Int. J. on Digital Libraries*, 1(1), pp 36-53, 1997.

[4] L.Fegaras and D.Maier, Towards an Effective Calculus for Object Query Languages, *SIGMOD'95*, pp 47-58, 1995.

[5] L.Fegaras, Optimizing Queries with Object Updates, *JIIS*, 12(2-3), pp 219-242, 1999.

[6] T.Grust and M.H.Scholl, How to Comprehend Queries Functionally, *JIIS*, 12(2-3), pp 191-218, 1999.

[7] A.Poulovassilis and C.Small, A Functional Programming Approach to Deductive Databases, *VLDB'91*, pp 491-500, 1991.

[8] A.Poulovassilis and C.Small, Algebraic query optimisation for database programming languages, *The VLDB Journal*, 5(2), pp 119-132, 1996.

[9] A.Poulovassilis and C.Small, Formal Foundations for Optimising Aggregation Functions in Database Programming Languages. *DBPL'97*, Springer-Verlag LNCS 1369, pp 299-318.

[10] P.W.Trinder, Comprehensions, a Query Notation for DBPLs. *DBPL'91*, pp 55-68.