PQL/Java language definition, v0.2 (draft)

Christoph Reichenbach, Yannis Smaragdakis, Neil Immerman

March 31, 2012

This document defines the domain language 'PQL/Java'. The purpose of PQL/Java is to facilitate parallelisable queries that can be embedded in Java programs.

1 Integration with Java

PQL/Java defines a number of additional keywords. To integrate with Java, these keywords assume the meaning defined in this document iff the source file they are defined in contains the following import statement:

```
import static edu.umass.pql.Query;
```

In the absence of this statement, the syntax and semantics of a PQL/Java program are identical to those of a regular Java program.

Keywords. PQL/Java defines the following keywords:
query **reduce** **forall** **exists** **over**

2 Syntax

Any Java expression (*JAVA-EXPR*) may contain a query (*QUERY*) as a subexpression. A query in turn follows the following production:

QUERY ::= ⟨*QUANT-EXPR*⟩ | id | ⟨*JAVA-EXPR*⟩ | ⟨*QEXPR*⟩

That is, it may be a quantifier expression (*QUANT-EXPR*) that quantifies one or more logical variables (such as `forall x, y : a[x] > b[y]`), a single identifier that references such a logical variable (such as `x` or `y` in the above example for *QUANT-EXPR*), or one of two unquantified expressions: an arbitrary Java expression (which may contain side effects but no logical variables) or a Q-Expression (*QEXPR*), which may contain logical variables and sub-queries but no side effects.

That the semantics of Q-Expressions and Java-expressions are identical whenever they produce the same expression.

Note that since Java expressions may contain subqueries, it is possible to nest multiple queries in the same expression, though these must not share variables.

In the following, we first describe quantifier expressions, then Q-expressions.

2.1 Quantifier expressions

Quantifier expressions take one of the following forms:

$$\begin{aligned} \textit{QUANT-EXPR} \quad ::= \quad & \langle \textit{QUANT} \rangle \ \langle \textit{ID} \rangle \ \text{':'} \ \langle \textit{QUERY} \rangle \\ | \ & \textbf{query} \ \text{'('} \ \langle \textit{MATCH} \rangle \ \text{')'} \ \text{':'} \ \langle \textit{QUERY} \rangle \\ | \ & \textbf{reduce} \ \text{'('} \ \mathsf{id} \ \text{')'} \ \langle \textit{ID} \rangle \ [\ \textbf{over} \ \langle \textit{ID-SEQ} \rangle \ ] : \ \langle \textit{QUERY} \rangle \end{aligned}$$

The first form of quantification is universal or existential quantification: here, *QUANT* may be either **forall** or **exists**. The second form of quantification, a `container` **query**, constructs maps, sets, or arrays. The third and final form is a general-purpose reduction operation. We describe these forms in more detail below.

2.1.1 Universal and existential quantifications

Universal or existential quantification extends over an identifier *ID*, which can be one of the following:

$$\textit{ID} \quad ::= \quad \mathsf{id} \ | \ \langle \textit{JAVA-TY} \rangle \ \ \mathsf{id}$$

For now, consider an example of the second form:

```
forall int x : x == x
```

This tests whether all `x` that are of type **int** are equal to themselves. This particular should always evaluate to **true**. Similarly,

```
exists int x : x == 0.5
```

will test whether there exists an integer `x` that is equal to 0.5; this test will evaluate to **false**.

We refer to the identifier occuring in the `ID` construct as the *query variable*. If a java type (*JAVA-TY*) is present, the query variable is *explicitly typed*. These two cases behave differently, as we discuss in Section 3.2. Informally, the difference is that for queries

```
/* A */ forall int x : a[x] > 0
/* B */ forall x : a[x] > 0
```

the compiler will infer the static type for Example B, and also infer that it should only consider values for `x` that occur in the domain of `a`, whereas for Example A, it will consider all $2^3 2$ possible **int** values for `x`, irrespectively of the size of `a`. Unless `a` is a special-case `java.util.Map`, this will always yield an array index out of bounds exception.

Section 5 describes extended syntax through which multiple identifiers may be quantified by the same existential or universal clause.

2.1.2 Container queries

A container query has the syntax

query '(' ⟨*MATCH*⟩ ')' ':' ⟨*QUERY*⟩

where a *MATCH* is one of the following:

MATCH ::= 'Set' '.' 'contains' '(' ⟨*CM*⟩ ')'
| 'Map' '.' 'find' '(' ⟨*CM*⟩ ')' '==' ⟨*CM*⟩ [**default** ⟨*QUERY*⟩]
| 'Array' '[' ⟨*ID*⟩ ']' '==' ⟨*CM*⟩

For now, assume that *CM* is an identifier. The first of the above productions then constructs a set, as in the following example:

```
query (Set.contains(x)): x == 0
```

This would construct a set of integers containing precisely the number zero.

The second construction builds a map:

```
query (Map.find(x) == y): range(1, 10).contains(x) && y == x*x
```

This would construct a map of all numbers from 1 to 10 to their squares. `range(1, 10)` here is a logical constant and a Java expression, denoting a set of all integers from 1 through 10.

Note that the above does not provide mappings for other numbers in the set; dereferencing them and treating them as integers has the same effect as trying to implicitly unbox a **null** in Java.

Maps may contain a default clause:

```
query (Map.find(x) == y default -1): range(1, 10).contains(x) && y == x*x
```

This would construct the same map as above, except that all numbers outside of the range 1 through 10 are mapped to `-1`.

The third construct builds arrays. For example,

```
query (Array[x] == y): range(1, 10).contains(x) && y == x*x
```

is the same as our map construction without defaults, with one exception: the missing array index (0) is filled in with the default value for the relevant type (i.e., 0 for integers). Thus, this will construct an 11-element array containing $0, 1, \ldots, 81, 100$.

In the above, we assumed that *CM* are always identifiers. This is not necessarily the case, since they may also be logical constants (Java expressions) or constructor invocations:

CM ::= ⟨*JAVA-EXPR*⟩ | ⟨*ID*⟩
| **new** ⟨*JAVA-TY*⟩ '(' [⟨*CM-SEQ*⟩] ')'

In the above, *CM-SEQ* denotes a comma-separated sequence of *CM*.

For example, if we have a datatype 'IntPair' that contains two integer values, we can construct a set of pairs of values and their negations as follows:

```
query (Set.contains(new IntPair(x, y))) : range(1, 100).contains(x) && y == -x
```

2.1.3 Reductions

The last query construct are reductions, which follow the syntax below:

reduce '(' id ')' ⟨*ID*⟩ [**over** ⟨*ID-SEQ*⟩] : ⟨*QUERY*⟩

In the above, *ID-SEQ* denotes a comma-separated sequence of *ID*.

The first parameter to a reduction is always a reduction operator, such as the built-in `sumInt`, `sumLong` and `sumDouble` operators. For example,

```
reduce (sumInt) x : set.contains(x)
```

This will sum up all numbers contained in `set` after coercing them to **int**.

Sometimes reduction requires additional free variables. We obtain these using the keyword **over**:

```
reduce (sumDouble) x over y : set.contains(y) && x == 1.0 / y
```

This will sum up the inverses of all numbers contained in the set.

2.2 Q-expressions

Specifically, a *QEXPR* has the following form:

QEXPR ::= '(' ⟨*QUERY*⟩ ')'
| ⟨*QUERY*⟩ ⟨*BINOP*⟩ ⟨*QUERY*⟩
| ⟨*QUERY*⟩ **instanceof** ⟨*JAVA-TY*⟩
| ⟨*UNOP*⟩ ⟨*QUERY*⟩
| ⟨*QUERY*⟩ '?' ⟨*QUERY*⟩ ':' ⟨*QUERY*⟩
| ⟨*QUERY*⟩ '.' 'find' '(' ⟨*QUERY*⟩ ')'
| ⟨*QUERY*⟩ '[' ⟨*QUERY*⟩ ']'
| ⟨*QUERY*⟩ '.' 'contains' '(' ⟨*QUERY*⟩ ')'
| ⟨*QUERY*⟩ '.' id
| ⟨*QUERY*⟩ '.' **length**
| ⟨*QUERY*⟩ '.' **size** '(' ')'

That is, it combines queries through a number of operators. In the above, *BINOP* is taken from a set of binary operators listed below (these have the same semantics as in Java):

- `||`: logical disjunction
- `&&`: logical conjunction
- `|`: bitwise disjunction (this is identical to logical disjunction for booleans)
- `&`: bitwise disjunction (this is identical to logical disjunction for booleans)
- `^`: bitwise exclusive-or
- `\%`: modulo

- `*`: multiplication
- `+`: addition
- `-`: subtraction
- `/`: division
- `>`: greater-than comparison
- `<`: less-than comparison
- `<=`: less-than or equal comparison
- `>=`: greater-than or equal comparison
- `!=`: inequality (for objects, this is reference inequality)
- `==`: equality (for objects, this is reference equality)
- `<<`: logical shift-left
- `>>`: logical shift-right, carrying the sign
- `>>>`: logical shift-right, not carrying the sign
- `=>`: logical implication (such that '`a => b`' is equivalent to '`(!a) || b`', if `!` is logical negation). This is the only new operator.

The **instanceof** operator also works as in Java, meaning that `a` **instanceof** `java.lang.Set` evaluates to true iff the object `a` has a dynamic type that is a subtype of `java.lang.Set`.

UNOP expands to the three unary Java operators, `-` (arithmetic negation), `!` (logical negation), and `~` (bitwise inversion).

The ternary operator $-?-:-$ is identical to Java's if-then-else operator for expressions (though PQL/Java provide syntactic sugar for the same construct, cf. Section 5). For example, `list.size() > 0 ? 1 : -1` evaluates to `1` iff the size of the list `list` is greater than zero, and to `-1` otherwise.

Q-expressions of the form `q.find(x)` or `q[x]` are equivalent. Both denote a map or array lookup (depending on the static type of `q`) and evaluate to the element indexed by `x`. For example, `myArray.find(3)` for an integer array `myArray` would obtain the 4th element of the array. This operation raises the same exceptions as regular Java array accesses might raise.

Q-expressions of the form `q.contains(x)` test whether a given Java set `q` contains element `x`. They evaluate to **true** or **false** and again copy the exception behaviour of Java.

Q-expressions of the form `q.f` are projections that obtain the contents of field `f`. Field `f` must be accessible from the context the query originates in according to the rules of reflective field access in Java (i.e., the field may be private and in a different class, but field access via the Java reflection API must be permitted).

Q-expressions of the form `q.length` or `q.size()` are equivalent. Both evaluate to the size of an array, `java.util.Collection`, or `java.util.Map`, as determined by the static type of `q`.

Figure 1 summarises the language syntax. We define the semantics in the next section, except for constructs inherited from Java:

JAVA-EXPR	A Java expression
JAVA-TY	A Java type
id	A Java identifier

3 Static Semantics

3.1 Types

Type checking is as for Java. All types are implicit, unless annotated explicitly in $\langle ID \rangle$. Boxing/unboxing conversions are implicit; failed conversion are described in the dynamic semantics.

reductions, the type of the body is the type of the value being reduced over (e.g., int for a sum of integers). This means that summing a set of values requires an explicit contains check:

reduce(addInt) x : set.contains(x)

3.2 Domains

Logical variables can either be declared 'with type' as in

forall **int** `i : ...`

or 'without type', as in

forall `i : ...`

These declarations specify different semantics. The explicitly typed variant has the obvious semantics, for example, **forall** **int** `i : a[i]` will raise an 'index out of bounds' exception if `a` is an array. Explicitly typed variables conceptually iterate over all *viable* values of their type τ, where *viable* is defined as follows:

- If τ is an enumeration, the viable values of τ are all members of the enumeration, as per `java.util.EnumSet.allOf`.
- If τ is an ordinal type such as **int** or **boolean**, the viable values of τ are all possible values for τ permitted by the Java programming language.
- If τ is a floating-point type, i.e., **float** or **double**, then the viable values for τ are all the values of that type that exist in live objects on the Java heap.
- Otherwise, τ is a reference type, and the viable values for τ are all the objects of that type that exist in live objects on the Java heap.

$QUERY$ ::= $\langle QUANT\text{-}EXPR \rangle$ | id | $\langle JAVA\text{-}EXPR \rangle$ | $\langle QEXPR \rangle$

$QUANT\text{-}EXPR$ ::= $\langle QUANT \rangle$ $\langle ID \rangle$ ':' $\langle QUERY \rangle$
| **query** '(' $\langle MATCH \rangle$ ')' ':' $\langle QUERY \rangle$
| **reduce** '(' id ')' $\langle ID \rangle$ [**over** $\langle ID\text{-}SEQ \rangle$] : $\langle QUERY \rangle$

$OVER$ ::= ε | **over** $\langle ID\text{-}SEQ \rangle$

$QUANT$ ::= **forall** | **exists**

$QEXPR$ ::= '(' $\langle QUERY \rangle$ ')'
| $\langle QUERY \rangle$ $\langle BINOP \rangle$ $\langle QUERY \rangle$
| $\langle QUERY \rangle$ **instanceof** $\langle JAVA\text{-}TY \rangle$
| $\langle UNOP \rangle$ $\langle QUERY \rangle$
| $\langle QUERY \rangle$ '?' $\langle QUERY \rangle$ ':' $\langle QUERY \rangle$
| $\langle QUERY \rangle$ '.' 'find' '(' $\langle QUERY \rangle$ ')'
| $\langle QUERY \rangle$ '[' $\langle QUERY \rangle$ ']'
| $\langle QUERY \rangle$ '.' 'contains' '(' $\langle QUERY \rangle$ ')'
| $\langle QUERY \rangle$ '.' id
| $\langle QUERY \rangle$ '.' **length**
| $\langle QUERY \rangle$ '.' **size** '(' ')'

$BINOP$ ::= '||' | '&&' | '|' | '&' | '^' | '%' | '*' | '+' | '-'
| '/' | '>' | '<' | '<=' | '>=' | '!=' | '==' | '<<'
| '>>' | '>>>' | '=>'

$UNOP$ ::= '!' | '~' | '-'

$MATCH$ ::= 'Set' '.' 'contains' '(' $\langle CM \rangle$ ')'
| 'Map' '.' 'find' '(' $\langle CM \rangle$ ')' '==' $\langle CM \rangle$ [**default** $\langle QUERY \rangle$]
| 'Array' '[' $\langle ID \rangle$ ']' '==' $\langle CM \rangle$

CM ::= $\langle JAVA\text{-}EXPR \rangle$ | id
| **new** $\langle JAVA\text{-}TY \rangle$ '(' [$\langle CM\text{-}SEQ \rangle$] ')'

$CM\text{-}SEQ$ ::= $\langle CM \rangle$ | $\langle CM\text{-}SEQ \rangle$ ',' $\langle CM \rangle$

ID ::= $\langle ID \rangle$ | $\langle JAVA\text{-}TY \rangle$ id

$ID\text{-}SEQ$ ::= $\langle ID \rangle$ | $\langle ID\text{-}SEQ \rangle$ ',' $\langle ID \rangle$

Figure 1: PQL/J syntax

By contrast, a local variable without explicit type is subject to *domain inference*. This means that the variable's type and bounds are inferred from the body of the quantification. The precise rules will be described in a later version of the specification. For now, the following shall give an intuition (Δ denotes a set):

$$\frac{\Delta, i \vdash B :: \delta}{\Delta \vdash \ \textbf{forall}\ i : B :: \delta}$$

$$\frac{}{\Delta, i \vdash \mathsf{a[i]} :: \mathsf{range(0,\ a.length\ -\ 1).contains(i)}}$$

where `range(a, b)` computes the set of all integers from `a` to `b` (inclusive):

```
public static java.util.Set<Integer> range(int min, int max);
```

Whenever a quantification has the judgement

$$Q : B :: \delta$$

we rewrite it to

$$Q : \delta \implies B$$

3.3 Semantic constraints

4 Dynamic Semantics

Exceptions in the query body are propagated to the outside. There is no guarantee about the order in which exceptions are delivered. Failed lookups in maps are translated into `java.lang.IndexOutOfBoundsException`s.

Queries such as

```
query(Map.find(int a) == int b) : a == 1
              && range(1,2).contains(b)
```

in which we construct a multi-map that is not a partial map are invalid and raise a query failure through the `edu.umass.pql.AmbiguousMapException`.

Map defaults Maps constructed with the **default** keyword are total maps that produce the default value unless they have been bound differently. Such maps cannot produce a

5 Extended syntax

The language further defines the following syntactic shortcuts, expressed as rewritings:

```
RW0:
 'if' <QUERY.0> 'then' <QUERY.1> 'else' <QUERY.2>
 ==>
  <QUERY.0> '?' <QUERY.1> ':' <QUERY.2>

RW1:
 <QUANT> <ID> <ID-SEQ> ':' <QUERY>
 ==>
  <QUANT> <ID> ':' <QUANT> <ID-SEQ> <QUERY>

RW2:
 'reduce' '(' X ')' ':' S
 ==>
  'reduce' '(' X ')' <JAVA-TY> id ':' id 'where' S '.' 'contains' '(' id ')'  // id fresh

RW3:
 'count' '(' X ')' Y ':' S
 ==>
  'reduce' '(' 'sum' ')' Y ':' '1' 'where' S

RW4
  Map[<CM>]  ==>  Map.find(<CM>):

RW5
  Array.find(<ID>)  ==>  Array[<ID>]
```

6 Reduction operator definition

Reduction operators must be associative and commutative, and provide a neutral element. In Java, they must be static methods annotated with exactly one of `@DefaultValueInt(d)`, `@DefaultValueLong(d)` or `@DefaultValueDouble(d)`, where `d` gives their neutral element, or not annotated at all, in which case their neutral element is **null**.