Name resolution

The QL compiler resolves names to program elements.

As in other programming languages, there is a distinction between the names used in QL code, and the underlying QL entities they refer to.

It is possible for different entities in QL to have the same name, for example if they are defined in separate modules. Therefore, it is important that the QL compiler can resolve the name to the correct entity.

When you write your own QL, you can use different kinds of expressions to refer to entities. Those expressions are then resolved to QL entities in the appropriate namespace.

In summary, the kinds of expressions are:

- **Module expressions**
  - These refer to modules.
  - They can be simple names, qualified references (in import statements), selections, or instantiations.
- **Type expressions**
  - These refer to types.
  - They can be simple names or selections.
- **Predicate expressions**
  - These refer to predicates.
  - They can be simple names or names with arities (for example in an alias definition), or selections.
- **signature expressions**
  - These refer to module signatures, type signatures, or predicate signatures.
  - They can be simple names, names with arities, selections, or instantiations.

Names

To resolve a simple name (with arity), the compiler looks for that name (and arity) in the namespaces of the current module.

In an import statement, name resolution is slightly more complicated. For example, suppose you define a query module `Example.ql` with the following import statement:

```
import javascript
```

The compiler first checks for a library module `javascript.qll`, using the steps described below for qualified references. If that fails, it checks for an explicit module named `javascript` defined in the module namespace of `Example.ql`.

Qualified references

A qualified reference is a module expression that uses `.` as a file path separator. You can only use such an expression in import statements, to import a library module defined by a relative path.

For example, suppose you define a query module `Example.ql` with the following import statement:

```
import examples.security.MyLibrary
```

To find the precise location of this library module, the QL compiler processes the import statement as follows:

1. The `.`s in the qualified reference correspond to file path separators, so it first looks up `examples/security/MyLibrary.qll` from the directory containing `Example.ql`.
2. If that fails, it looks up `examples/security/MyLibrary.qll` relative to the query directory, if any. The query directory is the first enclosing directory containing a file called `qlpack.yml`. (Or, in legacy products, a file called `queries.xml`.)
3. If the compiler can't find the library file using the above two checks, it looks up `examples/security/MyLibrary.qll` relative to each library path entry. The library path is usually specified using the `libraryPathDependencies` of the `qlpack.yml` file, though it may also depend on the tools you use to run your query, and whether you have specified any extra settings. For more information, see "Library path" in the QL language specification.

If the compiler cannot resolve an import statement, then it gives a compilation error.

Selections

You can use a selection to refer to a module, type, or predicate inside a particular module. A selection is of the form:

```
<module_expression>::<name>
```

The compiler resolves the module expression first, and then looks for the name in the namespaces for that module.

Example

Consider the following library module:

CountriesLib.qll

```
class Countries extends string {
  Countries() {
    this = "Belgium"
    or
    this = "France"
    or
```

```
      this = "India"
    }
  }

  module M {
    class EuropeanCountries extends Countries {
      EuropeanCountries() {
        this = "Belgium"
        or
        this = "France"
      }
    }
  }
```

You could write a query that imports `CountriesLib` and then uses `M::EuropeanCountries` to refer to the class `EuropeanCountries`:

```
import CountriesLib

from M::EuropeanCountries ec
select ec
```

Alternatively, you could import the contents of `M` directly by using the selection `CountriesLib::M` in the import statement:

```
import CountriesLib::M

from EuropeanCountries ec
select ec
```

That gives the query access to everything within `M`, but nothing within `CountriesLib` that isn't also in `M`.

Namespaces

When writing QL, it's useful to understand how namespaces (also known as environments) work.

As in many other programming languages, a namespace is a mapping from **keys** to **entities**. A key is a kind of identifier, for example a name, and a QL entity is a module, a type, or a predicate.

Each module in QL has six namespaces:

- The **module namespace**, where the keys are module names and the entities are modules.
- The **type namespace**, where the keys are type names and the entities are types.
- The **predicate namespace**, where the keys are pairs of predicate names and arities, and the entities are predicates.
- The **module signature namespace**, where the keys are module signature names and the entities are module signatures.
- The **type signature namespace**, where the keys are type signature names and the entities are type signatures.
- The **predicate signature namespace**, where the keys are pairs of predicate signature names and arities, and the entities are predicate signatures.

The six namespaces of any module are not completely independent of each other:

- No keys may be shared between the **module namespace** and the **module signature namespace**.
- No keys may be shared between the **type namespace** and the **type signature namespace**.
- No keys may be shared between the **module namespace** and the **type signature namespace**.
- No keys may be shared between the **type namespace** and the **module signature namespace**.
- No keys may be shared between the **predicate namespace** and the **predicate signature namespace**.
- No keys may be shared between the **module signature namespace** and the **type signature namespace**.

There is no relation between names in namespaces of different modules. For example, two different modules can define a predicate `getLocation()` without confusion. As long as it's clear which namespace you are in, the QL compiler resolves the name to the correct predicate.

Global namespaces

The namespaces containing all the built-in entities are called **global namespaces**, and are automatically available in any module. In particular:

- The **global module namespace** has a single entry `QlBuiltins`.
- The **global type namespace** has entries for the primitive types `int`, `float`, `string`, `boolean`, and `date`, as well as any database types defined in the database schema.
- The **global predicate namespace** includes all the built-in predicates, as well as any database predicates.
- The **global signature namespaces** are empty.

In practice, this means that you can use the built-in types and predicates directly in a QL module (without importing any libraries). You can also use any database predicates and types directly—these depend on the underlying database that you are querying.

Local namespaces

In addition to the global module, type, and predicate namespaces, each module defines a number of local module, type, and predicate namespaces.

For a module `M`, it is useful to distinguish between its **privately declared**, **publically declared**, **exported**, and **visible** namespaces. (These are described generically, but remember that there is always one for each of modules, module signatures, types, type signatures, predicates, and predicate signatures.)

- The **privately declared** namespaces of `M` contain all entities and aliases that are declared—that is, defined—in `M` and that are annotated as `private`.
- The **publically declared** namespaces of `M` contain all entities and aliases that are declared—that is, defined—in `M` and that are not annotated as `private`.
- The **exported** namespaces of `M` contain
    1. all entries from the **publically declared** namespaces of `M`, and

2. for each module `N` that is imported into `M` with an import statement that is not annotated as `private`: all entries from the **exported** namespaces of `N` that do not have the same name as any of the entries in the **publically declared** namespaces of `M`, and
3. for each module signature `S` that is implemented by `M`: an entry for each module signature default predicate in `S` that does not have the same name and arity as any of the entries in the **publically declared** predicate namespace of `M`.

- The **visible** namespaces of `M` contain
  1. all entries from the **exported** namespaces of `M`, and
  2. all entries from the **global** namespaces, and
  3. all entries from the **privately declared** namespace of `M`, and
  4. for each module `N` that is imported into `M` with an import statement that is annotated as `private`: all entries from the **exported** namespaces of `N` that do not have the same name as any of the entries in the **publically declared** namespaces of `M`.
  5. if `M` is nested within a module `N`: all entries from the **visible** namespaces of `N` that do not have the same name as any of the entries in the **publically declared** namespaces of `M`, and
  6. all parameters of `M`.

This is easiest to understand in an example:

OneTwoThreeLib.qll

```
import MyFavoriteNumbers

class OneTwoThree extends int {
  OneTwoThree() {
    this = 1 or this = 2 or this = 3
  }
}

private module P {
  class OneTwo extends OneTwoThree {
    OneTwo() {
      this = 1 or this = 2
    }
  }
}
```

The module `OneTwoThreeLib` **publically declares** the class `OneTwoThree` and **privately declares** the module `P`.

It **exports** the class `OneTwoThree` and anything that is exported by `MyFavoriteNumbers` (assuming `MyFavoriteNumbers` does not export a type `OneTwoThree`, which would not be **exported** by `OneTwoThreeLib`).

Within it, the class `OneTwoThree` and the module `P` are **visible**, as well as anything exported by *MyFavoriteNumbers*` (assuming `MyFavoriteNumbers` does not export a type `OneTwoThree`, which would not be **visible** within `OneTwoThreeLib`).

Example

Let’s see what the module, type, and predicate namespaces look like in a concrete example:

For example, you could define a library module `Villagers` containing some of the classes and predicates that were defined in the QL tutorials:

Villagers.qll

```
import tutorial

predicate isBald(Person p) {
  not exists(string c | p.getHairColor() = c)
}

class Child extends Person {
  Child() {
    this.getAge() < 10
  }
}

module S {
  predicate isSouthern(Person p) {
    p.getLocation() = "south"
  }

  class Southerner extends Person {
    Southerner() {
      isSouthern(this)
    }
  }
}
```

Module namespace

The module namespace of `Villagers` has entries for:

- The module `S`.
- Any modules exported by `tutorial`.

The module namespace of `S` also has entries for the module `S` itself, and for any modules exported by `tutorial`.

Type namespace

The type namespace of `Villagers` has entries for:

- The class `Child`.
- The types exported by the module `tutorial`.
- The built-in types, namely `int`, `float`, `string`, `date`, and `boolean`.

The type namespace of `S` has entries for:

- All the above types.
- The class `Southerner`.

Predicate namespace

The predicate namespace of `Villagers` has entries for:

- The predicate `isBald`, with arity 1.
- Any predicates (and their arities) exported by `tutorial`.
- The built-in predicates.

The predicate namespace of `S` has entries for:

- All the above predicates.
- The predicate `isSouthern`, with arity 1.