Modules

Modules provide a way of organizing QL code by grouping together related types, predicates, and other modules.

You can import modules into other files, which avoids duplication, and helps structure your code into more manageable pieces.

Defining a module ¶

There are various ways to define modules—here is an example of the simplest way, declaring an explicit module named `Example` containing a class `OneTwoThree`:

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

The name of a module can be any identifier that starts with an uppercase or lowercase letter.

`.ql` or `.qll` files also implicitly define modules. For more information, see "Kinds of modules."

You can also annotate a module. For more information, see of "Overview of annotations."

Note that you can only annotate explicit modules. File modules cannot be annotated.

Kinds of modules

File modules

Each query file (extension `.ql`) and library file (extension `.qll`) implicitly defines a module. The module has the same name as the file, but any spaces in the file name are replaced by underscores (`_`). The contents of the file form the body of the module.

Library modules

A library module is defined by a `.qll` file. It can contain any of the elements listed in Module bodies below, apart from select clauses.

For example, consider the following QL library:

OneTwoThreeLib.qll

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

This file defines a library module named `OneTwoThreeLib`. The body of this module defines the class `OneTwoThree`.

Query modules

A query module is defined by a `.ql` file. It can contain any of the elements listed in Module bodies below.

Query modules are slightly different from other modules:

- A query module can't be imported.
- A query module must have at least one query in its namespace. This is usually a select clause, but can also be a query predicate.

For example:

OneTwoQuery.ql

```
import OneTwoThreeLib

from OneTwoThree ott
where ott = 1 or ott = 2
select ott
```

This file defines a query module named `OneTwoQuery`. The body of this module consists of an import statement and a select clause.

Explicit modules

You can also define a module within another module. This is an explicit module definition.

An explicit module is defined with the keyword `module` followed by the module name, and then the module body enclosed in braces. It can contain any of the elements listed in "Module bodies" below, apart from select clauses.

For example, you could add the following QL snippet to the library file **OneTwoThreeLib.qll** defined above:

```
...
module M {
  class OneTwo extends OneTwoThree {
    OneTwo() {
      this = 1 or this = 2
    }
  }
}
```

This defines an explicit module named `M`. The body of this module defines the class `OneTwo`.

Parameterized modules

Parameterized modules are QL's approach to generic programming. Similar to explicit modules, parameterized modules are defined within other modules using the keywork `module`. In addition to the module name, parameterized modules declare one or more parameters between the name and the module body.

For example, consider the module `M`, which takes two predicate parameters and defines a new predicate that applies them one after the other:

```
module M<transformer/1 first, transformer/1 second> {
  bindingset[x]
  int applyBoth(int x) {
    result = second(first(x))
  }
}
```

Parameterized modules cannot be directly referenced. Instead, you instantiate a parameterized module by passing arguments enclosed in angle brackets (`<` and `>`) to the module. Instantiated parameterized modules can be used as a module expression, identical to explicit module references.

For example, we can instantiate `M` with two identical arguments `increment`, creating a module containing a predicate that adds 2:

```
bindingset[result] bindingset[x]
int increment(int x) { result = x + 1 }

module IncrementTwice = M<increment/1, increment/1>;

select IncrementTwice::applyBoth(40) // 42
```

The parameters of a parameterized module are (meta-)typed with signatures.

For example, in the previous two snippets, we relied on the predicate signature `transformer`:

```
bindingset[x]
signature int transformer(int x);
```

The instantiation of parameterized modules is applicative. That is, if you instantiate a parameterized module twice with identical arguments, the resulting object is the same. This is particularly relevant for type definitions inside parameterized modules as classes or via newtype, because the duplication of such type definitions would result in incompatible types.

The following example instantiates module `M` inside calls to predicate `foo` twice. The first call is valid but the second call generates an error.

```
bindingset[this]
signature class TSig;

module M<TSig T> {
  newtype A = B() or C()
}

string foo(M<int>::A a) { ... }

select foo(M<int>::B()),  // valid: repeated identical instantiation of M does not
duplicate A, B, C
       foo(M<float>::B()) // ERROR: M<float>::B is not compatible with M<int>::A
```

Module parameters are dependently typed, meaning that signature expressions in parameter definitions can reference preceding parameters.

For example, we can declare the signature for `T2` dependent on `T1`, enforcing a subtyping relationship between the two parameters:

```
signature class TSig;

module Extends<TSig T> { signature class Type extends T; }

module ParameterizedModule<TSig T1, Extends<T1>::Type T2> { ... }
```

Dependently typed parameters are particularly useful in combination with parameterized module signatures.

Module bodies

The body of a module is the code inside the module definition, for example the class `OneTwo` in the explicit module `M`.

In general, the body of a module can contain the following constructs:

- Import statements
- Predicates
- Types (including user-defined classes)
- Aliases
- Explicit modules
- Select clauses (only available in a query module)

Importing modules

The main benefit of storing code in a module is that you can reuse it in other modules. To access the contents of an external module, you can import the module using an import statement.

When you import a module this brings all the names in its namespace, apart from private names, into the namespace of the current module.

Import statements

Import statements are used for importing modules. They are of the form:

```
import <module_expression1> as <name>
import <module_expression2>
```

Import statements are usually listed at the beginning of the module. Each import statement imports one module. You can import multiple modules by including multiple import statements (one for each module you want to import). An import statement can also be annotated with `private`.

You can import a module under a different name using the `as` keyword, for example `import javascript as js`.

The `<module_expression>` itself can be a module name, a selection, or a qualified reference. For more information, see "Name resolution."

For information about how import statements are looked up, see "Module resolution" in the QL language specification.

Built-in modules

QL defines a `QlBuiltins` module that is always in scope. Currently, it defines a single parameterized sub-module `EquivalenceRelation`, that provides an efficient abstraction for working with (partial) equivalence relations in QL.

Equivalence relations

The built-in `EquivalenceRelation` module is parameterized by a type `T` and a binary base relation `base` on `T`. The symmetric and transitive closure of `base` induces a partial equivalence relation on `T`. If every value of `T` appears in `base`, then the induced relation is an equivalence relation on `T`.

The `EquivalenceRelation` module exports a `getEquivalenceClass` predicate that gets the equivalence class, if any, associated with a given `T` element by the (partial) equivalence relation induced by `base`.

The following example illustrates an application of the `EquivalenceRelation` module to generate a custom equivalence relation:

```
class Node extends int {
  Node() { this in [1 .. 6] }
}

predicate base(Node x, Node y) {
  x = 1 and y = 2
  or
  x = 3 and y = 4
}

module Equiv = QlBuiltins::EquivalenceRelation<Node, base/2>;

from int x, int y
where Equiv::getEquivalenceClass(x) = Equiv::getEquivalenceClass(y)
select x, y
```

Since `base` does not relate `5` or `6` to any nodes, the induced relation is a partial equivalence relation on `Node` and does not relate `5` or `6` to any nodes either.

The above select clause returns the following partial equivalence relation:

x	y
1	1
1	2
2	1
2	2

x	y
3	3
3	4
4	3
4	4

© 2023 GitHub, Inc. Terms Privacy