Proposal: Method Calling Convention

1 Synopsis

To encode frame conditions, the pillar of conditionals, a straightforward approach is compile statements in a new frame as a new Dafny `method`. Here'e a challenge:

`qregs` are expanded into its pure data representation such as `seq<nat>` and `seq<seq<nat>>` after compilation without using heap-allocated objects. A method where the users assumed that they passed a reference to the `qreg` object needs to emit a return value for the post-state of the `qreg` passed in. The type of the `qreg` passed to the method could change through the execution of the body, the return value needs to be calculated based on the final type accordingly.

1.1 Example

The following method

```
method GHZ(q : qreg[10])
  requires { q[0  .. 10] : nor ↦ ⊗ i . (0) }
  ensures  { q[0  .. 10] : en01 ↦ ∑ j ∈ [0 .. 2] . ⊗ k ∈ [0 .. 10] . ( j ) }
{
  ...
}
```

should be compiled into something like

```
method GHZ_Compiled(q_in : seq<nat>) returns (q_out : seq<seq<nat>>)
  requires { q_in[0  .. 10] : ⟦ nor ↦ ⊗ i . (0) ⟧ }
  ensures  { q_out[0  .. 10] : ⟦ en01 ↦ ∑ j ∈ [0 .. 2] . ⊗ k ∈ [0 .. 10] . ( j ) ⟧ }
{
  ...
}
```

2 Proposal

Methods can in fact have multiple parameter, therefore it's important to have a uniformed order to compile those input and return parameters.

2.1 Method Type, Reworked

To get a deterministc description of methods, the new method type is defined as followed.

```
data MethodType = MethodType
  { mtSrcParams :: [MethodElem]
  , mtSrcReturns :: [MethodElem]
  , mtInstantiate :: Map.Map Var Range -> [(Partition, QTy)]
  }

data MethodElem
  = MTyPure Var Ty
  | MTyQuantum Var Exp'
  deriving (Show, Eq, Ord)
```

The spotlight here is `mtInstantiate` which takes a correspondence between `qreg` variables used in definition and the `qreg` slices passed by the caller and returns partition-level typing requirements to be satisfied. The instantiator is deterministic, and therefore can determine the order of quantum states to be passed into the compiled method.

2.2 Input

```
(define-metafunction qafny
  call-conv : x ... -> x ...
  [(call-conv (n x ...))
   (n x' ...)
   (where (x' ...) (call-conv x ...))
   (where n (not (is-qreg n)))]
  [(call-conv (q x ...))
   (x' ... y' ...)
   (where (x' ...) (call-conv x ...))
   (where (y' ...) (call-conv-q/env q))])
```

where `call-conv-q/env` resolves emit symbols based on the `type` information encoded in the **pre**-condition. Passed qubits are compiled as emit variables corresponding to each range decided by the type information provided by the instantiatd instance. For convenience, those variables are put after non-quantum parameters.

2.3 Output

```
(define-metafunction qafny
  call-conv-ret : x ... -> x ...
  [(call-conv-ret (n x ...))
   (x' ...)
   (where (x' ...) (call-conv-ret x ...))
   (where n (not (is-qreg n)))]
  [(call-conv-ret (q x ...))
   (y' ... x' ...)
   (where (x' ...) (call-conv-ret x ...))
   (where (y' ...) (call-conv-q/env q))])
```

where `call-conv-q/env` resolves emit symbols based on the `type` information encoded in the **post**-condition.

3 Implementation

3.1 Pass `qreg`

How to pass a `qreg` argument? Pass the `qreg` seems suffices, but it's not enough because chances are that method requires less qubits than what the caller could provide. Therefore, the syntax should allow the caller to pass a slice of qubits, which can be of exactly the same syntax as passing a range.