Quantum Intermediate Representation Specification

Version 0.1

Introduction

This specification defines an intermediate representation for compiled quantum computations. The intent is that a quantum computation written in any language can be compiled into this representation as a common intermediate *lingua franca*. The intermediate representation would then be an input to a code generation step that is specific to the target execution platform, whether simulator or quantum hardware.

We see compilation as having three high-level phases:

1. A language-specific phase that takes code written in some quantum language, performs language-specific transformations and optimizations, and compiles the result into this intermediate format.
2. A generic phase that performs transformations and analysis of the code in the intermediate format.
3. A target-specific phase that performs additional transformations and ultimately generates the instructions required by the execution platform in a target-specific format.

By defining our representation within the popular open-source LLVM framework, we enable users to easily write code analyzers and code transformers that operate at this level, before the final target-specific code generation.

Role of This Specification

The representation defined in this specification is intended to be the target representation for language-specific compilers. In particular, a language-specific compiler may rely on the existence of the various utility and quantum functions defined in this specification.

It is neither required nor expected that any particular execution target actually implement every runtime function specified here. Rather, it is expected that the target-specific compiler will translate the functions defined here into the appropriate representation for the target, whether that be code, calls into target-specific libraries, metadata, or something else.

This applies to quantum functions as well as classical functions. We do not intend to specify a gate set that all targets must support, nor even that all targets use the gate model of computation. Rather, the quantum functions in this document specify the interface that language-specific compilers should meet. It is the role of the target-specific compiler to translate the quantum functions into an appropriate computation that meets the computing model and capabilities of the target platform.

Executable Code Generation Considerations

There are several areas where a code generator may want to significantly deviate from a simple rewrite of basic intrinsics to target machine code:

- The intermediate representation assumes that the runtime does not perform garbage collection, and thus carefully tracks stack versus heap allocation and reference counting for heap-allocated structures. A runtime that provides full garbage collection may wish to remove the reference count field from several intermediate representation structures and elide calls to `quantum.rt.free` and the various `unreference` functions.
- Depending on the characteristics of the target architecture, the code generator may prefer to use different representations for the various types defined here. For instance, on some architectures it will make more sense to represent small types as bytes rather than as single or double bits.
- The primitive quantum operations provided by a particular target architecture may differ significantly from the intrinsics defined in this specification. It is expected that code generators may significantly rewrite sequences of quantum intrinsics into sequences that are optimal for the specific target.

Identifiers

Identifiers in LLVM begin with a prefix, '@' for global symbols and '%' for local symbols, followed by the identifier name. Names must be in 8-bit ASCII, and must start with a letter or one of the special characters '$', '_', '-', and '.'; the rest of the characters in the name must be either one of those characters or a digit. It is possible to include other ASCII characters in names by surrounding the name in quotes and using '\xx' to represent the hex encoding of the character. LLVM has no analog to namespaces or similar named scopes that are present in many modern languages.

To the extent possible, symbols in the QIR should have identifiers that match the identifier used in the source language. The identifiers of local symbols should be converted to LLVM by merely adding the '%' prefix. Anonymous local variables generated by the compiler can be represented as %0, %1, etc., as is usual in LLVM.

Similarly, global symbols should have their identifiers converted by adding the '@' prefix. If the source language provides a named scoping mechanism, such as Python modules or Q# namespaces, then the fully-qualified name of the global symbol should be used.

Data Type Representation

We define LLVM representations for a variety of classical and quantum data types. In each case, we specify an LLVM type alias for the underlying type. Compilers should always emit the type alias so that the semantic type can be determined by LLVM-level compiler passes.

Unit

The `Unit` data type is not represented as an LLVM type because its primary use is as a return type, and all callables use a tuple as the return type. If there is a need to represent the `Unit` type for some purpose, a tuple type with no contained elements should be used.

The one possible value of `Unit`, `()`, is represented as a null tuple pointer.

Simple Types

The simple types are those whose values are fixed-size and do not contain pointers. They are represented as follows:

Type	LLVM Representation	Comments
`Int`	`%Int = type i64`	
`Double`	`%Double = double`	
`Bool`	`%Bool = i8`	0 is false, 1 is true.
`Result`	`%Result = i8`	0 is Zero, 1 is One.
`Pauli`	`%Pauli = i8`	0 is PauliI, 1 is PauliX, 3 is PauliY, and 2 is PauliZ.
`Qubit`	`%Qubit = i8 addrspace(2) *`	A qubit is logically a pointer into an alternate address space. The only meaningful operation on this type, other than passing as an argument to a quantum instruction, is equality comparison.
`Range`	`%Range = {%Int, %Int, %Int}`	In order, these are the start, step, and end of the range. When passed as a function argument or return value, ranges should be passed by value.

The following global constants are defined for use with the `%Result` and `%Pauli` types:

```
@Result.One = %Result 1
@Result.Zero = %Result 0

@Pauli.I = %Pauli 0
@Pauli.X = %Pauli 1
@Pauli.Y = %Pauli 3
@Pauli.Z = %Pauli 2
```

Variably-Sized Types

The variably-sized types are those whose values vary in size but do not contain pointers. Values of these types are represented as pointers to LLVM structures. The structures may be allocated on the heap or be located in static memory for compile-time constants.

Each structure starts with an `i32` reference count for memory management use. Next there is an `i32` size field, followed by the actual data. The reference count field is managed using the library functions described below. For static (compile-time constant) values of these types, the reference count should be set to -1. The `reference` and `unreference` functions recognize this value and will act appropriately, not updating the reference count or attempting to release the value.

Type	LLVM Representation	Comments
String	`%String = type { i32, i32, [0 x i8] }`	The byte array is the UTF-8 representation of the string. The size field is the size in bytes, not characters.

Type	LLVM Representation	Comments
BigInt	`%BigInt = type { i32, i32, [0 x i64]}`	The size field is the size in i64s.

Strings

The following utility functions are provided by the classical runtime to support strings:

Function	Signature	Description
quantum.rt.string_init	`void(%String*)`	Initializes a heap-allocated string by setting the reference count to 1.
quantum.rt.string_reference	`void(%String*)`	Increments the reference count of a string.
quantum.rt.string_unreference	`void(%String*)`	Decrements the string's reference count and releases the string if the count is now zero.
quantum.rt.string_concatenate	`%String* (%String*, %String*)`	Creates a new heap-allocated string that is the concatenation of the two argument strings.
quantum.rt.string_equal	`i1(%String*, %String*)`	Returns true if the two strings are equal, false otherwise.

The following utility functions support converting values of other types to strings. In every case, the returned string is allocated on the heap; the string can't be allocated by the caller because the length of the string depends on the actual value.

Function	Signature	Description
quantum.rt.int_to_string	`%String*(%Int)`	Returns a string representation of the integer.
quantum.rt.double_to_string	`%String*(%Double)`	Returns a string representation of the double.
quantum.rt.bool_to_string	`%String*(%Bool)`	Returns a string representation of the Boolean.
quantum.rt.result_to_string	`%String*(%Result)`	Returns a string representation of the result.
quantum.rt.pauli_to_string	`%String*(%Pauli)`	Returns a string representation of the Pauli.
quantum.rt.qubit_to_string	`%String*(%Qubit)`	Returns a string representation of the qubit.
quantum.rt.range_to_string	`%String*(%Range)`	Returns a string representation of the range.
quantum.rt.bigint_to_string	`%String*(%BigInt*)`	Returns a string representation of the big integer.

Big Integers

The following utility functions are provided by the classical runtime to support big integers. Note that all returned big integers will be allocated on the heap.

Function	Signature	Description
quantum.rt.bigint_init	`void(%BigInt*)`	Initializes a heap-allocated big integer by setting the reference count to 1.
quantum.rt.bigint_reference	`void(%BigInt*)`	Increments the reference count of a big integer.
quantum.rt.bigint_unreference	`void(%BigInt*)`	Decrements the big integer's reference count and releases the big integer if the count is now zero.
quantum.rt.bigint_negate	`%BigInt* (%BigInt*)`	Returns the negative of the big integer.
quantum.rt.bigint_add	`%BigInt* (%BigInt*, %BigInt*)`	Adds two big integers and returns their sum.
quantum.rt.bigint_subtract	`%BigInt* (%BigInt*, %BigInt*)`	Subtracts the second big integer from the first and returns their difference.
quantum.rt.bigint_multiply	`%BigInt* (%BigInt*, %BigInt*)`	Multiplies two big integers and returns their product.
quantum.rt.bigint_divide	`%BigInt* (%BigInt*, %BigInt*)`	Divides the first big integer by the second and returns their quotient.
quantum.rt.bigint_modulus	`%BigInt* (%BigInt*, %BigInt*)`	Returns the first big integer modulo the second.
quantum.rt.bigint_power	`%BigInt* (%BigInt*, i64)`	Returns the big integer raised to the integer power.
quantum.rt.bigint_bitand	`%BigInt* (%BigInt*, %BigInt*)`	Returns the bitwise-AND of two big integers.
quantum.rt.bigint_bitor	`%BigInt* (%BigInt*, %BigInt*)`	Returns the bitwise-OR of two big integers.
quantum.rt.bigint_bitxor	`%BigInt* (%BigInt*, %BigInt*)`	Returns the bitwise-XOR of two big integers.
quantum.rt.bigint_bitnot	`%BigInt* (%BigInt*)`	Returns the bitwise complement of the big integer.

Function	Signature	Description
quantum.rt.bigint_shiftleft	`%BigInt* (%BigInt*, i64)`	Returns the big integer arithmetically shifted left by the integer amount of bits.
quantum.rt.bigint_shiftright	`%BigInt* (%BigInt*, i64)`	Returns the big integer arithmetically shifted right by the integer amount of bits.
quantum.rt.bigint_equal	`i1(%BigInt*, %BigInt*)`	Returns true if the two big integers are equal, false otherwise.
quantum.rt.bigint_greater	`i1(%BigInt*, %BigInt*)`	Returns true if the first big integer is greater than the second, false otherwise.
quantum.rt.bigint_greater_eq	`i1(%BigInt*, %BigInt*)`	Returns true if the first big integer is greater than or equal to the second, false otherwise.

Container Types

Container types are those whose values contain other values. For instance, a tuple may contain several values of various types. The container types defined here are tuples and arrays.

Values of these types are represented as handles to LLVM structures, where the handles may be simple pointers or may contain additional information. The structures may be allocated on the heap or be located in static memory for compile-time constants. The handles should be treated as scalars and stored directly in LLVM variables or as direct fields of other data structures.

Each structure has an `i32` reference count for memory management use. The reference count field is managed using the library functions described in each section below. For static (compile-time constant) values of these types, the reference count should be set to -1. The `reference` and `unreference` functions recognize this value and will act appropriately, not updating the reference count or attempting to release the value.

Structures also contain a pointer to a destructor function; that is, to a function that is used to release values that are pointed to by this value. This function is generated by the compiler based on the specifics of the complex type. The precise details of the destructor function varies depending on the specific value type, although the signature of the destructor function is always `void(i8*)`, where the argument passed to the function is a pointer to the value to be destroyed. Values that the compiler knows don't require destructor functions should have a null pointer in this field.

We define an LLVM type for a pointer to a destructor:

```
%Destructor = type void(i8*)*
```

Tuples and User-Defined Types

Tuple data, including values of user-defined types, is represented as an LLVM structure type containing the reference count and destructor fields described above as a standard header, followed by the tuple fields. We define an LLVM type for the tuple header:

```
%TupleHeader = type {i32, %Destructor}
```

For instance, a tuple containing two integers, `(Int, Int)`, would be represented in LLVM as `type {%TupleHeader, i64, i64}`.

Tuple handles are simple pointers to the underlying data structure. To satisfy LLVM's strong typing, tuple pointers are passed as pointers to the initial tuple header field, and then cast to pointers to the correct data structures by the receiver:

```
%TuplePointer = type %TupleHeader*
```

The destructor for a tuple will get called with a pointer to the tuple structure as argument when the tuple is being released. The destructor should recursively unreference any variably-sized or container components of the tuple. If the tuple only contains simple types, then the destructor pointer should be null.

Many languages provide immutable tuples, along with operators that allow a modified copy of an existing tuple to be created. In QIR, this is implemented by creating a new copy of the existing tuple on the heap, and then modifying the newly-created tuple in place. In some cases, if the compiler knows that the existing tuple is not used after the creation of the modified copy, it is possible to avoid the copy and modify the existing tuple as long as there are no other references to the tuple. The `tuple_is_writable` utility function is used to test if the tuple pointer is the only reference.

The following utility functions are provided by the classical runtime to support tuples and user-defined types:

Function	Signature	Description
quantum.rt.tuple_init_stack	`void(%TuplePointer)`	Initializes a stack-allocated tuple by setting the reference count to -1 and the destructor to null. Any code required to recursively unreference contained components should be generated as in-line clean-up code.
quantum.rt.tuple_init_heap	`void(%TuplePointer, %Destructor)`	Initializes a heap-allocated tuple by setting the reference count to 1 and filling in the destructor.
quantum.rt.tuple_reference	`void(%TuplePointer)`	Increments the reference count of the tuple.
quantum.rt.tuple_unreference	`void(%TuplePointer)`	Decrements the tuple's reference count; calls its destructor and releases the tuple if the reference count is now zero.
quantum.rt.tuple_is_writable	`i1(%TuplePointer)`	Returns true if it is safe to modify the contents of the tuple; that is, it returns true if the reference count of the tuple is equal to 1.

Arrays

Array data is stored as an LLVM structure with the actual element data stored in a variably-sized byte array:

```
%ArrayData = type {i32, %Destructor, i32, i64, [0 x i8]}
```

The first `i32` is the reference count, and the `%Destructor` is the destructor for the array. The second `i32` is the size of each array element in bytes. The `i64` is the total number of elements in the array. Finally, the byte array holds the actual elements for the array. Pointers to elements will get cast to the actual element type by code that accesses the array.

The handle for an array is more complicated than a simple pointer. It holds the information that allows a list of array indices to be translated into a specific element offset:

```
%ArrayPointer = type {%ArrayData*, i32, [0 x i64]}
```

The `i32` is the number of dimensions in the array. The list of `i64`s provides information on the slice represented by this handle. Specifically, the first element of the list is an offset into the array data, and the next `d` elements (where `d` is the number of dimensions in the array) are the strides for each array. Finally, the last `d` elements are the lengths of the corresponding dimensions; these are used to validate that a set of dimension indices are within the array's bounds. Thus, the list has `2d+1` elements.

To access an element of the underlying array, the element index into the `ArrayData` is calculated by multiplying each dimension's index by the corresponding stride, adding the results together, and then adding the offset. For example, for a 3-dimensional array with offset 7 and strides 24, 8, and 1, the element at `[3, 2, 5]` would be at element `7 + 3x24 + 2x8 + 5x1 = 100` in the underlying `%ArrayData`.

Array destructors are invoked slightly differently than tuple destructors. Because every element of an array is the same and requires the same processing, the `unreference` routine for arrays invokes the destructor on each element of the array in turn, rather than on the entire array. This allows the "boilerplate" loop over elements to exist in one place, rather than being replicated in every destructor.

Many languages provide immutable arrays, along with operators that allow a modified copy of an existing array to be created. In QIR, this is implemented by creating a new copy of the existing arrays on the heap, and then modifying the newly-created arrays in place. In some cases, if the compiler knows that the existing arrays is not used after the creation of the modified copy, it is possible to avoid the copy and modify the existing arrays as long as there are no other references to the arrays. The `arrays_is_writable` utility function is used to test if the array handle is the only reference.

There are two special operations on arrays:

- An array *slice* is specified by providing a dimension to slice on and a `%Range` to slice with. This is equivalent to the current Q# array slicing feature, generalized to multi-dimensional arrays.
- An array *projection* is specified by providing a dimension to project on and an `i64` index value to project to. The resulting array has one fewer dimension than the original array, and is the segment of the original array with the given dimension fixed to the given index value.

Both slicing and projecting are implemented by creating a new `%ArrayPointer% with updated dimension count, offset, stride, and length information.

The following utility functions are provided by the classical runtime to support arrays:

Function	Signature	Description
quantum.rt.array_create	`void(%ArrayPointer, i32, %Destructor, i32, [0 x i64])`	Creates a heap-allocated array by setting the reference count to 1 and filling in the element size and the destructor. The second i32 is the dimension count. The list of `i64`s contains the length of each dimension. This routine allocates and initializes an `%ArrayData` structure and fills in the supplied `%ArrayPointer` structure.
quantum.rt.array_copy	`void(%ArrayPointer, %ArrayPointer)`	Initializes the first `%ArrayPointer` to be the same as the second, and increments the reference count of the underlying `%ArrayData`.
quantum.rt.array_get_dim	`i32(%ArrayPointer)`	Returns the number of dimensions in the array.
quantum.rt.array_get_length	`%Int(%ArrayPointer, i32)`	Returns the length of a dimension of the array. The `i32` is the zero-based dimension to return the length of.
quantum.rt.array_validate	`i1(%ArrayPointer, [0 x i64])`	Returns true if the list of indices is valid for the array and false otherwise.
quantum.rt.array_get_element	`i8*(%ArrayPointer, [0 x i64])`	Returns a pointer to the indicated element of the array.
quantum.rt.array_slice	`void(%ArrayPointer, $ArrayPointer, i32, %Range)`	Initializes the first `%ArrayPointer` to be a slice of the second. The `i32` indicates which dimension the slice is on, and the `%Range` specifies the slice. The reference count of the underlying `%ArrayData` is incremented.
quantum.rt.array_project	`void(%ArrayPointer, $ArrayPointer, i32, i64)`	Initializes the first `%ArrayPointer` to be a projection of the second. The `i32` indicates which dimension the projection is on, and the `i64` specifies the specific index value to project. The new `ArrayPointer` will have one fewer dimension than the original. The reference count of the underlying `%ArrayData` is incremented.

Function	Signature	Description
quantum.rt.array_reference	`void(%ArrayPointer)`	Increments the reference count of the array. This is almost never needed because new `%ArrayPointer`s are typically created with one of the above four routines, all of which update the reference count properly.
quantum.rt.array_unreference	`void(%ArrayPointer)`	Decrements the array's reference count; calls its destructor and releases the array if the reference count is now zero.
quantum.rt.array_is_writable	`i1(%ArrayPointer)`	Returns true if it is safe to modify the contents of the tuple; that is, it returns true if the reference count of the tuple is equal to 1.

Callables

We use to term *callable* to mean a subroutine in the source language. Different source languages use different names for this concept.

Code

Callables may have up to four different implementations to handle different combinations of functors. Each implementation is represented as an LLVM function that takes three tuple pointers as input and returns no output; that is, as an LLVM `void(%TuplePointer, %TuplePointer, %TuplePointer)`. The first input is the capture tuple, which will be null for top-level callables. The second input is the argument tuple. The third input points to the result tuple, which will be allocated by the caller.

We define the LLVM type `%CallableImpl = type void(%TuplePointer, %TuplePointer, %TuplePointer)` for convenience.

We use a caller-allocates strategy for the result tuple because this allows us to avoid a heap allocation in many cases. If the callee allocates space for the result tuple, that space has to be on the heap because a stack-based allocation would be released when the callee returns. The caller can usually allocate the result tuple on the stack, or reuse the result tuple pointer it received for tail calls.

The names of the implementation should be the namespace-qualified name of the callable with periods, followed by a hyphen ('-'), followed by `body` for the default implementation, `adj` for the adjoint implementation, `ctl` for the controlled implementation, and `ctladj` for the controlled adjoint implementation.

For instance, for a callable named `Some.Namespace.Symbol` with all four implementations, the compiler should generate the following in LLVM:

```
define void Some.Namespace.Symbol-body (%TuplePointer capture, %TuplePointer args,
%TuplePointer result)
{
  ; code goes here
}
```

```
define void Some.Namespace.Symbol-adj (%TuplePointer capture, %TuplePointer args,
%TuplePointer result)
{
  ; code goes here
}

define void Some.Namespace.Symbol-ctl (%TuplePointer capture, %TuplePointer args,
%TuplePointer result)
{
  ; code goes here
}

define void Some.Namespace.Symbol-ctladj (%TuplePointer capture, %TuplePointer
args, %TuplePointer result)
{
  ; code goes here
}
```

Each implementation should start with a prologue that decomposes the argument and capture tuples. Depending on the result type, it should end with an epilogue that fills in the result tuple. If the result tuple contains another container such as an array, it may be simpler and more efficient to fill in the sub-container in-line in the implementation, rather than creating a separate array and then copying it to the result tuple.

Implementation Table

For each defined callable, a four-entry table is created with pointers to these four labels; implementations that don't exist for a specific callable have a null pointer in that place. The table is defined as a global constant with the namespace-qualified name, with periods, of the callable.

For the example above, the following would be generated:

```
@Some.Namespace.Symbol = constant [4 x %CallableImpl*]
  [
    %CallableImpl* @Some.Namespace.Symbol-body,
    %CallableImpl* @Some.Namespace.Symbol-adj,
    %CallableImpl* @Some.Namespace.Symbol-ctl,
    %CallableImpl* @Some.Namespace.Symbol-ctladj
  ]
```

We define the LLVM type `%CallableImplTable = type [4 x %CallableImpl*]` for convenience.

External Callables

Callables may be specified as external; that is, they are declared in the quantum source, but are defined in an external component that is statically or dynamically linked with the compiled quantum code. For such callables, the compiler should generate a normal callable in LLVM, and generate a function body that unwraps the argument tuple, calls the external function, and then wraps the return value into a result tuple.

Generating the proper linkage is the responsibility of the target-specific compilation phase.

Generics

QIR does not provide support for generic or type-parameterized callables. It relies on the language-specific compiler to generate a new, uniquely-named callable for each combination of concrete type parameters. The LLVM representation treats these generated callables as the actual callables to generate code for; the original callables with open type parameters are not represented in LLVM.

Callable Values

Callable values are represented by a pointer to a structure with the following format:

Field	Data Type	Description
Specialization index	`i8`	Identifies the specialization to use when invoking this callable.
Implementation table	`%CallableImplTable*`	Points to the implementation table described above.
Controlled count	`i8`	Counts how many times the Controlled functor has been applied to this callable.
Capture tuple	`%TuplePointer`	A pointer to the tuple of captured values. This is null if the callable has no captured values.

The specialization index and controlled count are used and maintained by the callable utility functions to implement functors.

For convenience, the LLVM Callable type is defined:

```
%Callable = type { i8, %CallableImplTable*, i8, %TuplePointer }
```

To create a callable value, allocate space for the value on the stack or heap and use the `callable_init` utility function to initialize it from an implementation table and capture tuple or the `callable_copy` utility function to initialize it as a copy of another callable value.

Invoking a Callable

An explicit call to a top-level callable may be translated into a direct LLVM call instruction to the callable's body implementation:

```
call void @Symbol-body (%TuplePointer null, %TuplePointer %arg-tuple,
%TuplePointer %res-tuple)
```

The capture tuple pointer is null because top-level named callables never have a capture tuple.

Similarly, an explicit call to the adjoint of a named callable may be translated as:

```
call void @Symbol-adj (%TuplePointer null, %TuplePointer %arg-tuple, %TuplePointer
%res-tuple)
```

Calls to the controlled or controlled adjoint specializations of a named callable follow the same pattern; in these cases, the argument tuple should point to a two-tuple whose first element is the array of control qubits and whose second element is a tuple of the remaining arguments.

To invoke a callable value represented as a `%Callable*`, the `callable_invoke` utility function should be used. This function uses the information in the callable value to invoke the proper implementation with the appropriate parameters.

Implementing Functors

The Adjoint and Controlled functors are important for expressing quantum algorithms. They are implemented by the `callable_adjoint` and `callable_control` utility functions, which update a `%Callable` in place by applying the Adjoint or Controlled functors, respectively.

To support cases where the original, unmodified `%Callable` is still needed after functor application, the `callable_copy` routine may be used to create a new copy of the original `%Callable`; the functor may then be applied to the new `%Callable`. For instance, to implement the following:

```
let f = someOp;
let g = Adjoint f;
// ... code that uses both f and g ...
```

The following snippet of LLVM code could be generated:

```
%f = alloca %Callable
call %quantum.rt.callable_init(%Callable* %f, %CallableImplTable* @someOp,
%TuplePointer null)
%g = alloca %Callable
call %quantum.rt.callable_copy(%g, %f)
call %quantum.rt.callable_adjoint(%g)
```

Implementing Lambdas

The language-specific compiler should generate a new top-level callable of the appropriate type with implementation provided by the anonymous body of the lambda; this is known as "lifting" the lambda. A unique name should be generated for this callable. Lifted callables can support functors, just like any other callable. The language-specific compiler is responsible for determining the set of functors that should be supported by the lifted callable and generating code for them accordingly.

At the point where a lambda is created as a value in the code, a new callable data structure should be created with the appropriate contents. Any values referenced inside the lambda that are defined in a scope external to the lambda should be added to the lambda's capture tuple.

For instance, in the following Q# code, assuming a possible syntax for lambda expressions:

```
let x = 1;
let y = 2;
let f = lambda (z) { return x + y + z; };
```

The capture tuple for the callable value stored in `f` will consist of a tuple of two integers, `(1, 2)`. The code generated for the `let f =` line would do the following:

1. Build a tuple on the stack or heap from the current values of `x` and `y`. The tuple must be allocated on the heap if there is a possibility of the resulting callable being returned from this code.
2. Allocate a `%Callable` structure on the stack or heap. This structure should be allocated in the same place as the capture tuple.
3. Call the `callable_init` library routine with the callable pointer, the implementation table for the compiler-generated callable, and the capture pointer.
4. Assign the address of the `%Callable` structure to the local variable `%f`.

`callable_init` initializes the new callable structure with the provided parameters, a specialization index of 0, and a control count of 0.

Implementing Partial Application

A partial application is a form of closure value; that is, it is a lambda value, although the source syntax is different.

Partial applications should be rewritten into lambdas by the language-specific compiler. The lambda body may include additional code that performs argument tuple construction before calling the underlying callable. For instance, the following Q# partial application:

```
f(x, _)
```

could be rewritten as:

```
lambda (y) {
  let z = (x, y);
  return f(z);
}
```

Utility Functions

The following utility functions are provided by the classical runtime to support callables:

Function	Signature	Description
quantum.rt.callable_init	`void(%Callable*, %CallableImplTable*, %TuplePointer)`	Initializes the callable with the provided function table and capture tuple.
quantum.rt.callable_copy	`void(%Callable*, %Callable*)`	Initializes the first callable to be the same as the second callable.
quantum.rt.callable_invoke	`void(%Callable*, %TuplePointer, %TuplePointer)`	Invokes the callable with the provided argument tuple and fills in the result tuple.
quantum.rt.callable_adjoint	`void(%Callable*)`	Updates the callable bu applying the Adjoint functor.
quantum.rt.callable_control	`void(%Callable*)`	Updates the callable by applying the Controlled functor.

Metadata

Representing Source-Language Attributes

Many languages allow attributes to be placed on callable and type definitions. For instance, in Q# attributes are compile-time constant values of specific user-defined types that themselves have the `Microsoft.Quantum.Code.Attribute` attribute.

The language compiler should represent these attributes as LLVM metadata associated with the callable or type. For callables, the metadata representing the attribute should be attached to the LLVM global symbol that defines the implementation table for the callable. The identifier of the metadata node should be "!quantum.", where "!" is the LLVM standard prefix for a metadata value, followed by the namespace-qualified name of the attribute. For example, a callable `Your.Op`with two attributes, `My.Attribute(6, "hello")` and `Their.Attribute(2.1)`, applied to it would be represented in LLVM as follows:

```
@Your.Op = constant [4 x %CallableImpl*]
  [
    %CallableImpl* @Your.Op-body,
    %CallableImpl* @Your.Op-adj,
    %CallableImpl* @Your.Op-ctl,
    %CallableImpl* @Your.Op-ctladj
  ], !quantum.My.Attribute {i64 6, !"hello\00"},
     !quantum.Their.Attribute {double 2.1}
```

LLVM does not allow metadata to be associated with structure definitions, so there is no direct way to represent attributes attached to user-defined types. Thus, attributes on types are represented as named (module-level) metadata, where the metadata node's name is "quantum." followed by the namespace-qualified name of the type. The metadata itself is the same as for callables, but wrapped in one more level of

LLVM structure in order to handle multiple attributes on the same structure. For example, a type `Your.Type`with two attributes, `My.Attribute(6, "hello")` and `Their.Attribute(2.1)`, applied to it would be represented in LLVM as follows:

```
!quantum.Your.Type = !{ !{!"quantum.My.Attribute\00", i64 6, !"hello\00"},
                        !{ !"quantum.Their.Attribute\00", double 2.1} }
```

Standard LLVM Metadata

Debugging Information

Compilers are strongly urged to follow the recommendations in Source Level Debugging with LLVM.

Branch Prediction

Compilers are strongly urged to follow the recommendations in LLVM Branch Weight Metadata.

Other Compiler-Generated Metadata

> **TODO**: what quantum-specific "well-known" metadata do we want to specify?

Quantum Instruction Set and Runtime

Standard Operations

As recommended by the LLVM documentation, we do not define new LLVM instructions for standard quantum operations. Instead, we define a set of quantum functions that may be used by language-specific compilers and that should be recognized and appropriately dealt with by target-specific compilers:

Function	Signature	Description
quantum.x	`void(%Qubit)`	Pauli X gate.
quantum.y	`void(%Qubit)`	Pauli Y gate.
quantum.z	`void(%Qubit)`	Pauli Z gate.
quantum.h	`void(%Qubit)`	Hadamard gate.
quantum.s	`void(%Qubit)`	S (phase) gate.
quantum.t	`void(%Qubit)`	T gate.
quantum.cnot	`void(%Qubit, %Qubit)`	CNOT gate.
quantum.rz	`void(%Double, %Qubit)`	Z rotation; the `double` is the rotation angle.
quantum.r1	`void(%Double, %Qubit)`	Rotation around the

Function	Signature	Description
quantum.mz	`%Result(%Qubit)`	Measure in the Z basis.
quantum.ccnot	`void(%Qubit, %Qubit, %Qubit)`	Toffoli gate.
quantum.swap	`void(%Qubit, %Qubit)`	SWAP gate.
quantum.r	`void(%Pauli, %Double, %Qubit)`	General single-qubit rotation.
quantum.m	`%Result(%Pauli, %Qubit)`	General single-qubit measurement.
quantum.m2	`%Result(%Pauli, %Qubit, %Pauli, %Qubit)`	General two-qubit joint measurement.
quantum.measure	`%Result(i64, [0 x %Pauli], [0 x %Qubit])`	General multi-qubit joint measurement. The `i64` is the count of entries in both the `%Pauli` and `%Qubit` arrays.
quantum.exp	`void(i64, [0 x %Pauli], double, [0 x %Qubit])`	Applies the exponential of a multi-qubit Pauli operator. The `i64` is the count of entries in both the `%Pauli` and `%Qubit` arrays. The `double` multiplies the Pauli operator.
quantum.expfrac	`void(i64, [0 x %Pauli], i64, i64, [0 x %Qubit])`	Applies the exponential of a multi-qubit Pauli operator. The `i64` is the count of entries in both the `%Pauli` and `%Qubit` arrays. The Pauli operator is multiplied by the second `i64` divided by q raised to the power given by the third `i64`.
quantum.i	`void(%Qubit)`	Applies the identity gate to a qubit.

Qubit Management Functions

We define the following functions for managing qubits:

Function	Signature	Description
quantum.alloc	`void(i64, [0 x %Qubit]*)`	Fill in a pre-allocated array with newly allocated qubits. The `i64` is the number of qubits to allocate.
quantum.borrow	`void(i64, [0 x %Qubit]*, i32, [0 x %Qubit])`	Borrow an array of qubits. The first two arguments are as for `quantum.alloc`. The remaining count and array passed in are the qubits currently in scope that are not safe to be borrowed.
quantum.release	`void(i64, [0 x %Qubit]*)`	Release the allocated qubits in the array.

Function	Signature	Description
quantum.return	`void(i64, [0 x %Qubit]*)`	Return the borrowed qubits in the array.

Borrowing qubits means supplying qubits that are guaranteed not to be otherwise accessed while they are borrowed. The code that borrows the qubits guarantees that the state of the qubits when returned is identical, including entanglement, to their state when borrowed. It is always acceptable to satisfy `borrow` by allocating new qubits.

The array of in-scope qubits passed to `borrow` is used by the runtime to safely select qubits that will not be accessible during the scope of the borrowing. The compiler is responsible for inserting the code required to compute this array. We may change the signature of `borrow` to take an array of arrays instead, as this may allow less runtime copying of qubits. For in-scope callable values, it is necessary to determine the qubits, if any, that appear in the callable's capture tuple. See Capture Tuples, below, for a description of how this is done.

Allocated qubits are not guaranteed to be in any particular state. If a language guarantees that allocated qubits will be in a specific state, the compiler should insert the code required to set the state of the qubits returned from `alloc`.

It will likely be useful to provide usage hints to `alloc` and `borrow`. Since we don't know yet what form these hints may take, we leave them out for now.

Classical Runtime

Memory Management

The quantum runtime is not expected to provide garbage collection. Rather, the compiler should generate code that generates proper allocation for values on the stack or heap, and ensure that values are properly unreferenced when they go out of scope.

Stack versus Heap Allocation

We assume that the source language does not provide any mechanism for mutable values that persist across call values. That is, this discussion assumes that the source language provides no feature analogous to C `static` variables or to class static members as in C++, Java, or C#.

Any value that the compiler can prove will not be part of the return value can thus always be allocated on the stack using the LLVM `alloca` intrinsic. Values that might be part of the return value must be allocated on the heap; if such a value is determined at run time to no longer be a possible part of the return value, it may be explicitly released before the return. Similarly, values that require too much memory space to put on the stack can be allocated on the heap and explicitly released after their last use.

Values passed as arguments to a callable should not be released by the callable.

Values that are returned from a callable must not be allocated on the callee's stack. The calling code can rely on this, and can apply the same logic as above to either pass the value to the next caller or release the value after its last use.

We define the following functions for allocating and releasing heap memory, They should provide the same behavior as the standard C library functions malloc and free.

Function	Signature	Description
quantum.rt.heap_alloc	`i8*(i32)`	Allocate a block of memory on the heap.
quantum.rt.heap_free	`void(i8*)`	Release a block of allocated heap memory.

Reference Counting

The reference count of a heap-allocated structure should be incremented whenever a new long-lived reference to the structure is created and decremented whenever such a reference is overwritten or goes out of scope. A reference is long-lived if it is potentially part or all of the return value of the current callable.

In some cases, it may be possible to avoid incrementing the reference count of a structure when a new alias of the structure is created, as long as that alias is local and not part of the return value. For instance, in the following somewhat artificial Q# snippet:

```
function First<'T>(tuple: ('T1, 'T2)) : 'T1
{
  let x = tuple;
  let (first, second) = x;
  return first;
}
```

There is no need to increment the reference count for `tuple` when `x` is initialized, nor to decrement it at the function exit when `x` goes out of scope.

Example: Teleport

The following Q# code runs and tests a teleport operation. The implementation is somewhat artificial in order to provide a richer example.

```
namespace Microsoft.Quantum.Tutorial
{
    open Microsoft.Quantum.Intrinsic;

    operation PrepareBellState(q1 : Qubit, q2 : Qubit) : Unit is Adj + Ctl {
        H(q1);
        CNOT(q1, q2);
    }

    operation Teleport (nrReps : Int, (target : Qubit, msg : Qubit), initialize :
(Qubit => Unit is Adj)) : Result[] {
        mutable results = new Result[nrReps];

        for (iter in 1..nrReps) {
            using (source = Qubit()) {
                initialize(source);
                PrepareBellState(source, target);
                Adjoint PrepareBellState(source, msg);

                let measureZ = Measure([PauliZ], _);

                if (measureZ([source]) == One) {
                    Z(target);
                }
                if (measureZ([msg]) == One) {
                    X(target);
                }

                Adjoint initialize(target);

                set results w/= iter <- M(target);
            }
        }
        return results;
    }

    operation RunTests (nrReps : Int) : Bool {
        let angle = 0.5;
        let initialize = Rx(angle, _);
        mutable success = true;

        using ((target, msg) = (Qubit(), Qubit())) {
            let results = Teleport(nrReps, (target, msg), initialize);
            for (r in results) {
                if (r != Zero) {
                    set success = false;
                }
            }
        }
        return success;
    }
}
```

Here is the translation to LLVM using the representation from this document. We assume that quantum instrinsics such as H and CNOT are inlined. We are not using singleton-tuple equivalence, so that tuples of a single element are explicitly allocated as a one-element LLVM structure. We assume for now that arrays are represented as an LLVM structure containing a reference count, an element count, and a zero-length (variable-length) array, and are passed by reference but are immutable.

Comments are added for explanation and would not be generated by the compiler.

Note that this is hand-coded LLVM and is probably not valid.

```
define void Microsoft.Quantum.Tutorial.PrepareBellState-body(%TuplePointer
%capture, %TuplePointer %args, %TuplePointer %result)
```

```
{
entry:
    %0 = type { %TupleHeader, %Qubit, %Qubit }
    %1 = bitcast %TuplePointer %args to %0*
    ; %2 points to the first qubit
    %2 = getelementptr %Qubit*, %0* %1, i64 0, i32 1
    %q1 = load %Qubit, %Qubit* %2
    ; %3 points to the second qubit
    %3 = getelementptr %Qubit*, %0* %1, i64 0, i32 2
    %q2 = load %Qubit, %Qubit* %3
    call void quantum.h(%Qubit %q1)
    call void quantum.cnot(%Qubit %q1, %Qubit %q2)
    ret void
}

define void Microsoft.Quantum.Tutorial.PrepareBellState-adj(%TuplePointer
%capture, %TuplePointer %args, %TuplePointer %result)
{
entry:
    %0 = type { %TupleHeader, %Qubit, %Qubit }
    %1 = bitcast %TuplePointer %args to %0*
    ; %2 points to the first qubit
    %2 = getelementptr %Qubit*, %0* %1, i64 0, i32 1
    %q1 = load %Qubit, %Qubit* %2
    ; %3 points to the second qubit
    %3 = getelementptr %Qubit*, %0* %1, i64 0, i32 2
    %q2 = load %Qubit, %Qubit* %3
    call void quantum.cnot(%Qubit %q1, %Qubit %q2)
    call void quantum.h(%Qubit %q1)
    ret void
}

define void Microsoft.Quantum.Tutorial.PrepareBellState-ctl(%TuplePointer
%capture, %TuplePointer %args, %TuplePointer %result)
{
    ; ...
}

define void Microsoft.Quantum.Tutorial.PrepareBellState-ctladj(%TuplePointer
%capture, %TuplePointer %args, %TuplePointer %result)
{
    ; ...
}

@Microsoft.Quantum.Tutorial.PrepareBellState = constant %CallableImplTable
  [
    %CallableImpl* @Microsoft.Quantum.Tutorial.PrepareBellState-body,
    %CallableImpl* @Microsoft.Quantum.Tutorial.PrepareBellState-adj,
    %CallableImpl* @Microsoft.Quantum.Tutorial.PrepareBellState-ctl,
    %CallableImpl* @Microsoft.Quantum.Tutorial.PrepareBellState-ctladj
  ]

; This is a generated operation that implements the partial application of Measure
inside Teleport
```

```
; Note that, in principle, because the captured value is a compile-time constant,
the language-specific
; phase could simply insert the constant value and ignore the capture tuple.
; We don't perform this optimization here in order to provide an example of
capture tuple usage.
define void Microsoft.Quantum.Tutorial.Teleport-lambda1-body(%TuplePointer
%capture, %TuplePointer %args, %TuplePointer %result)
{
entry:
    ; Get the Pauli array value from the capture tuple
    %0 = type { %TupleHeader, %ArrayPointer }
    %1 = bitcast %TuplePointer %capture to %0*
    %2 = getelementptr %ArrayPointer*, %0* %1, i64 0, i32 1
    %3 = load %ArrayPointer, %ArrayPointer* %2

    ; Get the qubit array from the args tuple
    ; Since the type is the same as for the capture tuple, we can reuse the type
    %4 = bitcast %TuplePointer %args to %0*
    %5 = getelementptr %ArrayPointer*, %0* %4, i64 0, i32 1
    %6 = load %ArrayPointer, %ArrayPointer* %5

    ; Prepare the argument tuple for the call to Measure
    %7 = type { %TupleHeader, %ArrayPointer, %ArrayPointer }
    %8 = alloca %7
    %9 = bitcast %7* %8 to %TuplePointer
    call void quantum.rt.tuple_init_stack(%TuplePointer %9)
    %10 = getelementptr %ArrayPointer*, %7* %8, i64 0, i32 1
    store %ArrayPointer %3, %ArrayPointer* %10
    %11 = getelementptr %ArrayPointer*, %7* %8, i64 0, i32 2
    store %ArrayPointer %7, %ArrayPointer* %11

    ; Call Measure. Note the re-use of the result tuple; this is essentially a
tail call
    call void Microsoft.Quantum.Intrinsics.Measure-body(%TuplePointer null,
%TuplePointer %9, %TuplePointer %result)

    ret void
}

@Microsoft.Quantum.Tutorial.Teleport-lambda1 = constant %CallableImplTable
  [
    %CallableImpl* @Microsoft.Quantum.Tutorial.Teleport-lambda1-body,
    %CallableImpl* null,
    %CallableImpl* null,
    %CallableImpl* null
  ]

define void Microsoft.Quantum.Tutorial.Teleport-body(%TuplePointer %capture,
%TuplePointer %args, %TuplePointer %result)
{
entry:
    %0 = type { %TupleHeader, %Int, %TuplePointer, %Callable* }
    %1 = bitcast %TuplePointer %args to %0*
    ; %2 points to the repetition count
    %2 = getelementptr %Int*, %0* %1, i64 0, i32 1
    %nrReps = load %Int, %Int* %2
    ; %3 points to the pointer to the target/msg tuple, %4 to the tuple itself,
and %6 to the tuple properly typed
    ; Then, %7 points to target and %8 to msg
    %3 = getelementptr %TuplePointer*, %0* %1, i64 0, i32 2
    %4 = load %TuplePointer, %TuplePointer* %3
    %5 = type { %TupleHeader,  %Qubit, %Qubit }
    %6 = bitcast %TuplePointer %4 to %5*
    %7 = getelementptr %Qubit*, %5* %6, i64 0, i32 1
    %target = load %Qubit, %Qubit* %7
    %8 = getelementptr %Qubit*, %5* %6, i64 0, i32 2
    %msg = load %Qubit, %Qubit* %8
    ; %9 points to the initialize pointer
    %9 = getelementptr %Callable**, %0* %1, i64 0, i32 3
    %initialize = load %Callable*, %Callable** %9

    ; Get the results array handle from the result tuple and initialize it as a 1-
dimensional array of %Results.
    ; We rrepresent each %Result as the low-order bit of a byte, rather than
worrying about packing bits into bytes
    ; more densely. If such packing is required, it could be added.
    %10 = type { %TupleHeader, %ArrayPointer }
    %11 = bitcast %TuplePointer %result to %10*
    %12 = getelementptr %ArrayPointer*, %10* %11, i64 0, i32 1
    %results = load %ArrayPointer, %ArrayPointer* %12
    call void quantum.rt.array_create(%ArrayPointer %results, i32 1, %Destructor
null, i32 1, [ %Int %nrReps ])

preheader-1:
    ; Here we've constant-folded, rather than creating and then disassembling the
    ; 1..nrReps Range. We've also constant-folded below, so we skip the various
start and step symbols here.
    br label header-1

header-1:
    %iter = phi %Int [ 1, %preheader-1 ], [ %nextindex-1, %exiting-1 ]
    ; We've pre-done the select here since we know the loop direction statically
    %13 = icmp sle %iter, %nrReps
    br %13, label body-1, label exit-1

body-1:
    ; Implement the using statement
    %14 = alloca [1 x %Qubit]
    %15 = bitcast [1 x %Qubit]* %14 to [0 x %Qubit]*
    call void @quantum.alloc(i64 1, [0 x %Qubit]* %15)
    %16 = getelementptr %Qubit*, [1 x %Qubit]* %14, i64 0
    %source = load %Qubit, %Qubit* %16

    ; Create the argument tuple for initialize
    %17 = type { %TupleHeader, %Qubit }
    %18 = alloca %17
    %19 = bitcast %17* %18 to %TuplePointer
    call void quantum.rt.tuple_init_stack(%TuplePointer %19)
    %20 = getelementptr %Qubit*, %17* %18, i64 0, i32 1
    store %Qubit %source, %Qubit* %20
    ; Call through the initialize callable pointer, with a null result pointer
    ; because the initialize callable returns Unit
    call void quantum.rt.invoke_callable(%Callable* %initialize, %TuplePointer
%19, %TuplePointer null)

    ; Create the argument tuple for PrepareBellState
    %21 = type { %TupleHeader, %Qubit, %Qubit }
    %22 = alloca %21
    %22 = bitcast %21* %22 to %TuplePointer
    call quantum.rt.tuple_init_stack(%TuplePointer %22)
    %23 = getelementptr %Qubit*, %21* %22, i64 0, i32 1
    store %Qubit %source, %Qubit* %23
    %24 = getelementptr %Qubit*, %21* %22, i64 0, i32 2
    store %Qubit %target, %Qubit* %24
    ; Call PrepareBellState, optimizing to just call it directly
    call void Microsoft.Quantum.Tutorial.PrepareBellState-body(%TuplePointer null,
%TuplePointer %22, %TuplePointer null)

    ; Reuse the PrepareBellState argument tuple for Adjoint PrepareBellState
    store %Qubit %msg, %Qubit* %24
    ; Call Adjoint PrepareBellState, optimizing to just call it directly
    call void Microsoft.Quantum.Tutorial.PrepareBellState-adj(%TuplePointer null,
%TuplePointer %22, %TuplePointer null)

    ; Generate the callable value for the partial application, starting with the
capture tuple
    ; Note that we've already declared %10 as the LLVM type for a tuple containing
a single array,
    ; so we just reuse that type definition here and below.
    ; This could obviously be hoisted from the loop, but we omit that optimization
here to make the
    ; LLVM code easier to match with the Q# source.
    %25 = alloca %10
    %26 = bitcast %10* %25 to %TuplePointer
    call void quantum.rt.tuple_init_stack(%TuplePointer %26)
    %27 = extractvalue %10* %25, i32 1
    call void quantum.rt.array_create(%ArrayPointer %27, i32 1, %Destructor null,
i32 1, [ i64 1 ])
    %28 = call i8* quantum.rt.array_get_element(%ArrayPointer %27, [i64 0])
    ; We know there are no other handles to this array, so we can write directly
to the element
    store %Pauli @Pauli.Z, i8* %28
    %measureZ = alloca %Callable
    call void quantum.rt.callable_init(%Callable* %measureZ,
                                       %CallableImplTable*
@Microsoft.Quantum.Tutorial.Teleport-lambda1,
                                       %TuplePointer %26)

    ; Create the argument tuple for calling measureZ([source])
    %29 = alloca %10
    %30 = bitcast %10* %29 to %TuplePointer
    call void quantum.rt.tuple_init_stack(%TuplePointer %30)
    ; And initialize the qubit array, assuming a %Qubit takes 8 bytes
    %31 = extractvalue %10* %29, i32 1
    call void quantum.rt.array_create(%ArrayPointer %31, i32 8, %Destructor null,
i32 1, [ i64 1 ])
    %32 = call i8* quantum.rt.array_get_element(%ArrayPointer %31, [i64 0])
    ; We know there are no other handles to this array, so we can write directly
to the element
    %33 = bitcast i8* %32 to %Qubit*
    store %Qubit %source, %Qubit* %33

    ; And now the result tuple
    %34 = type { %TupleHeader, %Result }
    %35 = alloca %34
    %36 = bitcast %34* %35 to %TuplePointer
    call void quantum.rt.tuple_init_stack(%TuplePointer %36)

    ; Call measureZ
    call void quantum.rt.callable_invoke(%Callable* %measureZ, %TuplePointer %30,
%TuplePointer %36)

    ; And get the result out
    %37 = getelementptr %Result*, %34* %35, i64 0, i32 1
    %38 = load %Result, %Result* %37

    ; Is the result One?
    %39 = icmp eq %38, %Result.One
    br %39, label %true-1, label %continue-1
true-1:
    call quantum.z(%target)
    br label %continue-1
continue-1:

    ; We know that the result tuple from the previous call is not going to be
used, so we can reuse it here.
    ; Indeed, this is also true of the qubit array it uses, so all we need to do
is update the qubit in the array
    ; and call measureZ again. The key here is that all of this information is
available to the compiler.
    store %Qubit %msg, %Qubit* %33
    call void quantum.rt.callable_invoke(%Callable* %measureZ, %TuplePointer %30,
%TuplePointer %36)
    %40 = load %Result, %Result* %37
    %41 = icmp eq %40, %Result.One
    br %41, label %true-2, label %continue-2
true-2:
    call quantum.z(%target)
    br label %continue-2
continue-2:

    ; Create the adjoint of initialize.
    ; This could obviously be hoisted from the loop, but we omit that optimization
here to make the
    ; LLVM code easier to match with the Q# source.
    %40 = alloca %Callable
    call void quantum.rt.callable_copy(%Callable* %40, %Callable* %initialize)
    call void quantum.rt.callable_adjoint(%Callable* %40)

    ; And call it. Again, we re-use the tuple from earlier (the call to
initialize) because we know
    ; that it isn't used any more.
    store %Qubit %target, %Qubit* %20
    call void quantum.rt.invoke_callable(%Callable* %40, %TuplePointer %19,
%TuplePointer null)

    ; Measure the target and update the results array. The compiler can statically
determine that there
    ; are no other references to the results array and that the old value is
discarded immediately, so
    ; the array can safely be updated in place.
    %41 = call %Result quantum.mz(%target)
    %42 = call i8* quantum.rt.array_get_element(%results, [%Int %iter])
    store %Result %41, i8* %42

    ; Our partial application is going out of scope, so we need to unreference the
array we created for it.
    call void quamtum.rt.array_unreference(%ArrayPointer %27)

    ; Finish the using statement by releasing the allocated qubit
    call quantum.release i32 1, [0 x %Qubit]* %15

exiting-1:
    %nextindex = add %Int %iter, %Int 1
    br label %header-1

exit-1:

    ; The results array is already stored in the result tuple, so there's nothing
else to do before returning.
    ret void
}

@Microsoft.Quantum.Tutorial.Teleport = constant %CallableImplTable
  [
    %CallableImpl* @Microsoft.Quantum.Tutorial.Teleport-body,
    %CallableImpl* null,
    %CallableImpl* null,
    %CallableImpl* null
  ]

; This is a generated operation that implements the partial application of Rx
inside RunTests.
; Note that, in principal, because the captured value is a compile-time constant,
the language-specific
; phase could simply insert the constant value and ignore the capture tuple.
; We don't perform this optimization here in order to provide an example of
capture tuple usage.
define void Microsoft.Quantum.Tutorial.RunTests-lambda1-body(%TuplePointer
%capture, %TuplePointer %args, %TuplePointer %result)
```

```
{
entry:
    ; Get the rotation angle value from the capture tuple
    %0 = type { %TupleHeader, %Double }
    %1 = bitcast %TuplePointer %capture to %0*
    %2 = getelementptr %Double*, %0* %1, i64 0, i32 1
    %3 = load %Double, %Double* %2

    ; Get the qubit from the args tuple
    %4 = type { %TupleHeader, %Qubit }
    %5 = bitcast %TuplePointer %capture to %4*
    %6 = getelementptr %Qubit*, %0* %5, i64 0, i32 1
    %7 = load %Qubit, %Qubit* %5

    call void quantum.rz(%3, %7)

    ret void
}

define void Microsoft.Quantum.Tutorial.RunTests-lambda1-adj(%TuplePointer
%capture, %TuplePointer %args, %TuplePointer %result)
{
    ; Get the rotation angle value from the capture tuple
    %0 = type { %TupleHeader, %Double }
    %1 = bitcast %TuplePointer %capture to %0*
    %2 = getelementptr %Double*, %0* %1, i64 0, i32 1
    %3 = load %Double, %Double* %2

    ; Get the qubit from the args tuple
    %4 = type { %TupleHeader, %Qubit }
    %5 = bitcast %TuplePointer %capture to %4*
    %6 = getelementptr %Qubit*, %0* %5, i64 0, i32 1
    %7 = load %Qubit, %Qubit* %5

    ; The adjoint of a rotation is a rotation through the negative of the angle
    %8 = fneg %3

    call void quantum.rz(%8, %7)

    ret void
}

define void Microsoft.Quantum.Tutorial.RunTests-lambda1-ctl(%TuplePointer
%capture, %TuplePointer %args, %TuplePointer %result)
{
    ; ... elided ...
}

define void Microsoft.Quantum.Tutorial.RunTests-lambda1-ctladj(%TuplePointer
%capture, %TuplePointer %args, %TuplePointer %result)
{
    ; ... elided ...
}
```

```
@Microsoft.Quantum.Tutorial.RunTests-lambda1 = constant %CallableImplTable
  [
    %CallableImpl* @Microsoft.Quantum.Tutorial.RunTests-lambda1-body,
    %CallableImpl* @Microsoft.Quantum.Tutorial.RunTests-lambda1-adj,
    %CallableImpl* @Microsoft.Quantum.Tutorial.RunTests-lambda1-ctl,
    %CallableImpl* @Microsoft.Quantum.Tutorial.RunTests-lambda1-ctladj
  ]


define void Microsoft.Quantum.Tutorial.RunTests-body(%TuplePointer %capture,
%TuplePointer %args, %TuplePointer %result)
{
entry:
    ; Get nrReps from the argument tuple
    %0 = type { %TupleHeader, %Int }
    %1 = bitcast %args to %0*
    %2 = getelementptr %Int*, %0* %1, i64 0, i32 1
    %nrReps = load %Int, %Int* %2

    %angle = double 0.5

    ; Generate the callable value for the partial application pr Rx, starting with
the capture tuple
    %3 = type { %TupleHeader, %Double }
    %4 = alloca %3
    %5 = bitcast %3* %4 to %TuplePointer
    call void quantum.rt.tuple_init_stack(%TuplePointer %5)
    %6 = getelementptr %Double*, %3* %4, i64 0, i32 1
    store %Double %angle, %Double* %6
    %initialize = alloca %Callable
    call void quantum.rt.callable_init(%Callable* %initialize,
                                       %CallableImplTable*
@Microsoft.Quantum.Tutorial.RunTests-lambda1,
                                       %TuplePointer %5)

    ; Allocate space for the mutable variable. A compiler would normally nore that
this value is actually
    ; part of the result tuple, and so just use that space directly. We don't do
that here so we can give
    ; an example of how mutable values are dealt with.
    %success = alloca %Bool
    store %Bool true, %Bool* %success

    ; Implement the using statement.
    %7 = alloca [2 x %Qubit]
    %8 = bitcast [2 x %Qubit]* %7 to [0 x %Qubit]*
    call void @quantum.alloc(i64 2, [0 x %Qubit]* %8)
    %9 = getelementptr %Qubit*, [2 x %Qubit]* %7, i64 0
    %target = load %Qubit, %Qubit* %9
    %10 = getelementptr %Qubit*, [2 x %Qubit]* %7, i64 1
    %msg = load %Qubit, %Qubit* %10

    ; Build the argument tuple for the call to Teleport
    %11 = type { %TupleHeader, %Int, %TuplePointer, %Callable* }
    %12 = alloca %11
    %13 = bitcast %11* %12 to %TuplePointer
    call void quantum.rt.tuple_init_stack(%TuplePointer %13)
    %14 = getelementptr %Int*, %11* %12, i64 0, i32 1
    store %Int %nrReps, %Int* %14
    %15 = getelementptr %TuplePointer*, %11* %12, i64 0, i32 2
    %16 = type { %TupleHeader,  %Qubit, %Qubit }
    %17 = alloca %16
    %18 = bitcast %16* %17 to %TuplePointer
    call void quantum.rt.tuple_init_stack(%TuplePointer %18)
    %19 = getelementptr %Qubit*, %16* %17, i64 0, i32 1
    store %Qubit %target, %Qubit* %19
    %20 = getelementptr %Qubit*, %16* %17, i64 0, i32 2
    store %Qubit %msg, %Qubit* %20
    store %TuplePointer %18, %TuplePointer* %15
    %21 = getelementptr %Callable**, %0* %1, i64 0, i32 3
    store %Callable* %initialize, %Callable** %21

    ; And the results tuple
    %22 = type { %TupleHeader, %ArrayPointer }
    %23 = alloca %22
    %24 = bitcast %22* %23 to %TuplePointer
    call void quantum.rt.tuple_init_stack(%TuplePointer %24)
    ; We don't have to initialize the array -- that will happen in the called
operation

    call void Microsoft.Quantum.Tutorial.Teleport-body(%TuplePointer null,
%TuplePointer %13, %TuplePointer %24)

    ; For this loop, we use the template without any constant folding or other
optimizations,
    ; so we have an example that's easy to compare to the spec.
preheader-1:
    %25 = getelementptr %ArrayPointer*, %22* %23, i64 0, i32 1
    %26 = load %ArrayPointer, %ArrayPointer* %25
    %init-1 = %Int 0
    %step-1 = %Int 1
    %end-1 = call quantum.rt.array_get_length(%ArrayPointer %26, i32 0)
    %dir-1 = icmp sgt %step-1, 0
    br label header-1
header-1:
    %index-1 = phi %Int [ %init-1, %preheader-1], [ %nextindex-1, %exiting-1 ]
    ; %2 tells us if we continue; we need to check <= or >= depending on whether
the step is > or < 0
    %27 = icmp sle %index-1, %end-1
    %28 = icmp sge %index-1, %end-1
    %29 = select %dir-1, %27, %28
    br %2, label %body-1, label %exit-1
body-1:
    %30 = call i8* quantum.rt.array_get_element(%ArrayPointer %26, [ %Int %index-1
])
    %31 = load i8, i8* %30
    %32 = trunc i8 %31 to %Result
    %33 = icmp ne %32, %Result.Zero
    br %33, label %true-1, label %false-1
true-1:
    store %Bool 0, %Bool* %success
false-1:
exiting-1:
    %nextindex-1 = add i64 %index-1, %step-1
    br label %header-1
exit-1:

    ; Unreference the results array to finish the using statement
    call void Quantum.rt.array_unreference(%26)

    ; Fill in our result tuple
    %34 = type { %TupleHeader, %Bool }
    %35 = bitcast %TuplePointer %result to %34*
    %36 = getelementptr %Bool*, %34* %35, i64 0, i32 1
    store %Bool %success, %Bool* %36

    ; Everything else is stack-allocated, so no need to unreference anything

    ret void;
}

@Microsoft.Quantum.Tutorial.RunTests = constant %CallableImplTable
  [
    %CallableImpl* @Microsoft.Quantum.Tutorial.RunTests-body,
    %CallableImpl* null,
    %CallableImpl* null,
    %CallableImpl* null
  ]
```