Chapter 1

A Compiler for a Functional Quantum Programming Language

Jonathan Grattage and Thorsten Altenkirch

Abstract: We introduce a compiler for the functional quantum programming language QML [1], developed in Haskell. The compiler takes QML expressions as input and outputs a representation of quantum circuits (via the category **FQC** of finite quantum computations) which can be simulated by the simulator presented here, or by using a standard simulator for quantum gates. We discuss the structure of the compiler and how the semantic rules are compiled.

1.1 INTRODUCTION

Quantum programming is now a firmly established field; see recent text books as an example [2, 4] However, quantum programs are usually presented as low level quantum gates. There are a number of proposals as to how best to integrate quantum effects in a high level language, [5, 6, 7, 9], but none of them introduce high level structures for quantum control and quantum data. In our previous paper we introduced the language QML [1] which realizes quantum control structures acting on quantum data, and which can, in principle, be translated in quantum circuits. This paper explains in detail how a compiler, which translates QML programs into quantum circuits, is implemented. The compiler itself is written in Haskell and its source code is available from http://www.cs.nott.ac.uk/∼jjg/qml.

We begin by describing the structure of QML programs and the compiler. The compilation of QML terms is then discussed in detail, with emphasis on the differences between the strict and non-strict case constructs, and orthogonality judgements. This leads to the definition of a compiler following the denotational semantics and rules of QML.

1.2 SYNTAX OF QML

QML programs are typed expressions defined by the syntax and grammar of QML. The BNF grammar for QML types and expressions is:

$$\sigma = 1 \mid \sigma \otimes \tau \mid \sigma \oplus \tau$$

$$\begin{aligned} t \;&= x \qquad \mid \textbf{let}\ x = t\ \textbf{in}\ u \\ &\mid\ x \uparrow ys \mid () \\ &\mid\ (t,u) \mid \textbf{let}\ (x,y) = t\ \textbf{in}\ u \\ &\mid\ \mathrm{qinl}\ t \mid \mathrm{qinr}\ u \\ &\mid\ \textbf{case}\ \ t\ \textbf{of}\ \{\mathrm{qinl}\ x \Rightarrow u \mid \mathrm{qinr}\ y \Rightarrow u'\} \\ &\mid\ \textbf{case}^\circ\ t\ \textbf{of}\ \{\mathrm{qinl}\ x \Rightarrow u \mid \mathrm{qinr}\ y \Rightarrow u'\} \\ &\mid\ \{(\kappa)\ t \mid (\iota)\ u\} \end{aligned}$$

As an illustration of a program, a simple example is the Hadamard operation on a single qbit, given by:

$$\begin{aligned} &had : \mathbf{Q_2} \multimap \mathbf{Q_2} \\ &had\ x = \textbf{if}^\circ\ x \\ &\qquad \textbf{then}\ \{\mathrm{qfalse} \mid (-1)\ \mathrm{qtrue}\} \\ &\qquad \textbf{else}\ \ \{\mathrm{qfalse} \mid \mathrm{qtrue}\} \end{aligned}$$

This program can be read as an operation which, depending on the input qbit x, returns one of two possible superpositions of a qbit. Note that **if**$^\circ$ is a special case of **case**$^\circ$. We can also easily calculate that applying the program twice returns the qbit to the original state, cancelling the amplitudes. Note that qtrue and qfalse are syntactic-sugar for qinl () and qinr (), respectively. The type of a single qbit, denoted $\mathbf{Q_2}$, is $1 \oplus 1$

The compiler must have some way of interpreting these typed terms, and this is achieved by implementing the grammar in the compiler as Haskell datatypes: *Ty* representing the QML expression types and *Tm* representing the QML expressions themselves.

$$\textbf{data}\ Ty \;= T1 \mid Ty \oplus Ty \mid Ty \otimes Ty$$

$$\begin{aligned} \textbf{data}\ Tm = {} & Var\ String && \mid Let\ String\ Tm\ Tm\ (Maybe\ Ty) \\ \mid {} & Weak\ Tm\ [String] && \mid Void \\ \mid {} & Tm \otimes Tm && \mid LetP\ String\ String\ Tm\ Tm\ (Maybe\ Ty) \\ \mid {} & Inl\ Tm && \mid Inr\ Tm \\ \mid {} & Case\ \ Tm\ (String, Tm)\ (String, Tm) \\ \mid {} & Case^o\ Tm\ (String, Tm)\ (String, Tm)\ Orth \\ \mid {} & Sup\ (\mathbb{C}, Tm)\ (\mathbb{C}, Tm) \\ \mid {} & Embed\ Tm \end{aligned}$$

The final construct above in *Tm*, *Embed*, is not syntactic, but is a type level construct. It embeds a strict computation into a non-strict computation, see [1].

Returning to the example program *had*, passing it to the QML parser would rewrite it according to the internal representation, producing the term:

$$\begin{aligned} &Case^o\ (Var\ \texttt{"x"})\ (\texttt{"\_l"}, Sup\ (1, \mathrm{qfalse})\ (-1, \mathrm{qtrue})) \\ &\qquad\qquad (\texttt{"\_r"}, Sup\ (1, \mathrm{qfalse})\ (1, \mathrm{qtrue}))\ OSup \end{aligned}$$

where *OSup* is an orthogonality judgement, which is required whenever the strict morphism $Case^o$ is used; see section 1.5.4.

1.3 QUANTUM MACHINE CODE

The low level machine code that out compiler generates is an **FQC** morphism that is represented as a quantum circuit. This is included along with some other information in the computation type, *Comp*:

data $Comp = Comp\{\, uCon \in Snoc\ String, ty \in Ty, fqc \in FQC \,\}$

This datatype is constructed by three functions. The first, *uCon* (used context), returns a list which contains the names of all variables that are used by the computation. This is for type correctness: a computation is not correct unless every variable passed to it by the context has been used. The second contains the returned type of the function, and the last contains the **FQC** morphism, represented by the type *FQC*:

data $FQC = FQC\{\, a, h, b, g \in Int, \phi \in Unitary \,\}$

This datatype is made up from five destructor functions: *a*, *h*, *b*, and *g*, which all return integers, and ϕ, which returns an object of type *Unitary*. *a* represents the 'size' of the input to this **FQC** (i.e, the size of the input type; the number of qbits required to represent it), *b* represents the size of the output, while *h* and *g* represent the size of the heap and garbage required, respectively. There is the condition that $a + h = b + g$, but this does not need to be enforced as the compiler never builds *FQC* objects where this is not true. $\phi \in A \otimes H \multimap_{\text{unitary}} B \otimes G$ represents a reversible quantum computation as a unitary operator on Hilbert spaces — for details see [1]/

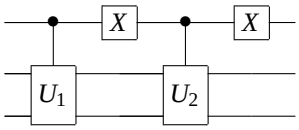
We represent unitary operators as circuits which can be constructed using the following combinators:

data $Unitary = \bigotimes [Unitary] \mid \bigodot [Unitary]$
$\mid Perm\ [Int] \quad \mid Cond\ Unitary\ Unitary$
$\mid Rotate\ (\mathbb{C},\mathbb{C})\ (\mathbb{C},\mathbb{C})$

$\bigotimes$ takes a list of *Unitary*, and represents these operations acting in parallel (hence the use of the tensor symbol). Similarly, $\bigodot$ represents the *Unitary* operations passed to it acting in series (the choice of symbol represents composition). *Perm xs* is simply a permutation operator. *Rotate* represents any 1-qbit unitary rotation to the state passed, which is used to generate any given superposition - we can represent negation as a special rotation $unot = Rotate\ (0,1)\ (1,0)$. It follows from the Kitaev-Solovay theorem, see [4], pp. 616-624, that we can represent all unitary operators this way.

Finally, *Cond* represents a conditional operation, and is slightly unusual. If the control qbit is true then the first operation is performed on the remaining qbits, while if it is false the second operation is performed. If the control qbit is in a superposition, then both operations are applied to generate the correct superposition of outputs. This is different to the standard $controlled - U$ operation, where the identity is applied if the control qbit is false, but *Cond* can be generated by simple application of two $controlled - U$ operations and two *Not* operations, as shown

in the following circuit, for which *Cond* is a useful shorthand:



Two further functions are often used when defining *Unitary* operators, to aid understanding and to simplify what could otherwise become cumbersome definitions. These are *idu x* $\in$ *Int* $\rightarrow$ *Unitary* and *swapN o l n* $\in$ *Int* $\rightarrow$ *Int* $\rightarrow$ *Int* $\rightarrow$ *Unitary*. The function *idu* simply creates an identity permutation of the required size. *swapN* creates another permutation operation, based on the three input integers, *o*, *l* and *n*. The permutation generated by this operation swaps a 'block' of qbits from anywhere in a circuit to the top, putting what was originally first after the moved block, followed by the remainder. *o* gives the size of the 'offset' to the block we wish to move, with *l* giving the size of this block. *n* is simply the size of the entire segment, for purposes of plumbing.

All objects of *Unitary* also have an associated arity; the number of qbits required to implement them. These are given by the following function:

$$
\begin{array}{lll}
\mathit{arity} \in \mathit{Unitary} & \rightarrow \mathit{Maybe\ Int} & \\
\mathit{arity}\ \ (\bigotimes\ [\,]) & = \mathit{Just}\ 0 & \\
\mathit{arity}\ \ (\bigotimes\ (x:xs)) & = \textbf{do}\ m \leftarrow \mathit{arity}\ x & \\
 & \qquad n \leftarrow \mathit{arity}\ (\bigotimes\ xs) & \\
 & \qquad \mathit{return}\ (m+n) &
\end{array}
$$

$$
\begin{array}{lll}
\ldots & & \\
\mathit{arity}\ \ (\mathit{Cond}\ \phi\ \psi) & = \textbf{do}\ m \leftarrow \mathit{arity}\ \phi & \\
 & \qquad n \leftarrow \mathit{arity}\ \psi & \\
 & \qquad \mathit{guard}\ (m \equiv n) & \\
 & \qquad \mathit{return}\ (1+m) &
\end{array}
$$

We define a function *eval* which calculates the matrix representation of a unitary circuit — here are some examples of how this mapping is achieved, with *matMult* being matrix multiplication, and *tensor* calculating the matrix tensor product:

$$
\begin{array}{lll}
\textbf{type}\ \mathit{Row} & = & [\mathbb{C}] \\
\textbf{type}\ \mathit{Matrix} & = & [\mathit{Row}]
\end{array}
$$

$$
\begin{array}{lll}
\mathit{eval} \in & \mathit{Unitary} & \rightarrow \mathit{Matrix} \\
\mathit{eval} & (\bigotimes\ xs) & = \mathit{mPar}\ (\mathit{map}\ \mathit{eval}\ xs) \\
\mathit{eval} & (\bigodot\ xs) & = \mathit{mSeq}\ (\mathit{map}\ \mathit{eval}\ xs)
\end{array}
$$

$$
\begin{array}{lll}
\ldots & & \\
\mathit{mPar} \in [\mathit{Matrix}] & \rightarrow \mathit{Matrix} & \\
\mathit{mPar} & [\,] & = [[1]] \\
\mathit{mPar} & (x:xs) & = \mathit{tensor}\ x\ (\mathit{mPar}\ xs)
\end{array}
$$

$$
\begin{array}{lll}
\mathit{mSeq} \in [\mathit{Matrix}] & \rightarrow \mathit{Matrix} & \\
\mathit{mSeq} & [x] & = x \\
\mathit{mSeq} & (x:xs) & = \mathit{matMult}\ x\ (\mathit{mSeq}\ xs)
\end{array}
$$

The compiler also allows a direct mapping from programs to matrices using

the *compute* rather than *compile* function. Additionally, the compiler also optimises *Unitary* types to give an optimised representation of the circuit, without changing the meaning.

1.4 STRUCTURE OF THE COMPILER

While the general structure of the compiler is quite standard, it is unual because our language doesn't allow implicit weakenenings. Hence we have to implement some book keeping of used variables.It takes as input parsed QML programs, and then performs a translation into its object language, in this case **FQC** morphisms. These morphisms, explained in [1], fully describe the computation, giving the size of input required, how much heap is required, how much garbage is produced, and a unitary operation that performs the computation. This unitary operation is essentially a description of a low-level quantum circuit (our 'machine' code). The **FQC** object can then be compiled further into the matrix that describes the computation (the lowest level 'machine' code).

The compiler starts by taking parsed QML programs, which we assume to be originally written as typed expressions, but have been converted by the parser into its internal representation. We have already stated that QML programs are typed expressions, and given the type for terms, *Tm*, and for QML types, *Ty*. These types are combined to give the final type for QML programs, implemented in the following way:

$$\begin{aligned}
&\textbf{type } \mathit{Prog} \;= \mathit{Env}\ \mathit{FDef}\\
&\textbf{data } \mathit{Env}\ a = \mathit{Env}\{\,\mathit{unEnv} \in \mathit{Snoc}\ (\mathit{String}, a)\,\}\\
&\textbf{data } \mathit{FDef} \;= \mathit{FDef}\ \mathit{FSig}\ \mathit{Tm}\\
&\textbf{data } \mathit{FSig} \;= \mathit{FSig}\ \mathit{Con}\ \mathit{Ty}\\
&\textbf{type } \mathit{Con} \;= \mathit{Env}\ \mathit{Ty}
\end{aligned}$$

These constructors tell us that a QML program (*Prog*) is an environment (*Env*) of function definitions (*FDef*). Function definitions are in turn defined as having a function signature, *FSig*, which contains the required context (*Con*) with the expected return type, and the QML term, *Tm*.

To illustrate the **FQC** morphism, the morphism generated for the quantum equivalent of the negation operation is described, which in QML syntax could be written:

$$\begin{aligned}
&\mathit{qnot} : \mathbf{Q_2} \multimap \mathbf{Q_2}\\
&\mathit{qnot}\ x = \textbf{if}^{\circ}\ x\\
&\qquad\qquad \textbf{then } \text{qfalse}\\
&\qquad\qquad \textbf{else } \text{qtrue}
\end{aligned}$$

From this the parser generates:

$$\begin{aligned}
&\mathit{qnot} \in \mathit{Prog}\\
&\mathit{qnot} = \mathit{Env}\ (\mathit{SNil} :< (\texttt{"qnot"}, \mathit{FDef}\ (\mathit{FSig}\ (\mathit{Env}\ (\mathit{SNil} :< (\texttt{"x"}, qb)))\ qb)\\
&\qquad\qquad (\mathit{Case}^{o}\ (\mathit{Var}\ \texttt{"x"})\ (\texttt{"_l"}, \text{qfalse})\ (\texttt{"_r"}, \text{qtrue})\ (\mathit{OInrl}))))
\end{aligned}$$

The compiler now applies the compilation function to produce an **FQC** morphism

that performs the action of the program:

$FQC\{a = 1, h = 0, b = 1, g = 0, \phi = Not\}$

This is packaged up inside the *Comp* type, which also contains information about the variables used to compute the program, and the output type of the program:

$Comp\{uCon = SNil :< \texttt{"x"}, ty = \mathbf{Q_2}, FQC = ...\}$

The morphism ϕ in the *FQC* object describes the simple circuit that negates a single qbit, just the Not gate, while using the *eval* function described in section 1.3, we could further compile this into the standard matrix representation.

The actual mechanics of how compilation is achieved are described in the next section, 1.5.

1.5 COMPILING QML

The compiler for QML is defined as a single Haskell function, *compile*. This function compiles each subterm of the expression that represents the QML program into *Comp* objects, which contain the **FQC** morphism, by calling the recursive function *compileTm*.

$compile \in Prog \rightarrow Error\ (Env\ Comp)$

As can be seen from the type, *compile* takes as input a QML program and returns, if there are no errors, a list (environment) of computations. At present we only consider programs consisting of a single function, and hence only return a single computation. The *compile* function is mainly concerned with error checking, using the Error monad, and passes the work of compilation onto the *compileTm* function. If there are no type, arity, or context errors then it returns the computation, via *cleanComp*, which simply 'cleans' the computation by optimising the underlying unitary morphism that *compileTm* produced.

The function that performs the majority of the compilation, *compileTm*, has the type

$compileTm \in (Env\ Comp) \rightarrow Con \rightarrow Tm \rightarrow Maybe\ Ty \rightarrow Error\ Comp$

The type $(Env\ Comp)$ simply contains the functions that have already been compiled, and are available for use by the compiler. *compileTm* also takes as input the current context, the term to be compiled, and the expected type (if known, hence the use of the *Maybe*) of the term. It returns either an error or a *Comp* object. The function is defined using pattern matching over the type of terms, *Tm*, so there is a separate function for each of the twelve term forms given by the grammar for QML, given by recursion over the term syntax. We begin by looking at how the variable rule is compiled, as it is the first rule – and is very simple.

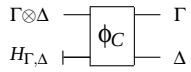
$$
\begin{aligned}
&compileTm\ \_\ \Gamma\ (Var\ x)\ mTy = \\
&\quad \textbf{do}\ \sigma \quad \leftarrow (elookup\ x\ \Gamma) \\
&\qquad \textbf{let}\ |\sigma| = size\ \sigma \\
&\qquad greturn\ mTy\ (Comp\{uCon = (SNil :< x), ty = \sigma, \\
&\qquad\qquad\qquad\qquad fqc \quad = FQC\{a = |\sigma|, h = 0, b = |\sigma|, g = 0, \\
&\qquad\qquad\qquad\qquad\qquad\qquad \phi = idu\ |\sigma|\}\})
\end{aligned}
$$

To compile a variable, we first look up the type of the variable x in the context Γ, and call it σ. The *Env* lookup function, *elookup*, would return an Error type

if the variable did not exist. The size of the type is then calculated, and this information is used to generate the **FQC** morphism. The context used in compiling this term is simply the variable, hence *uCon* (*u*sed-*Con*text) only contains *x*. The type of the computation is the type of the variable, σ, and the *FQC* is the identity morphism of the appropriate size. *greturn* is a small extension of the usual *return* function that confirms the type correctness of the returned computation, or else gives an informative error.

1.5.1 Context Sharing

Several of the more complicated circuit diagrams make use of a strict **FQC** morphism labelled ϕ_C.

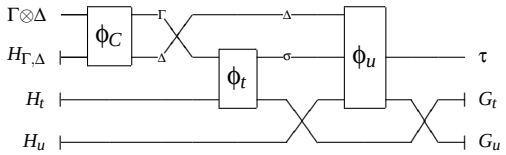


This morphism is used in the compilation of several terms, and allows the interpretation of the operator $\otimes$ on contexts. It makes use of δ_2, which shares a single qbit using a simple controlled-not operation [1]. We define the ϕ_C morphism as $C_{\Gamma,\Delta} \in \mathbf{FQC}^\circ [\![\Gamma \otimes \Delta]\!] ([\![\Gamma]\!] \otimes [\![\Delta]\!])$ by induction over the definition of $\Gamma \otimes \Delta$. If a variable $x : \sigma$ appears in both contexts we must use $\delta_\sigma \in \mathbf{FQC}^\circ [\![\sigma]\!] ([\![\sigma]\!] \otimes [\![\sigma]\!])$ which generalises δ_2 by applying it in parallel to all qbits. All the other cases can be dealt with by applying monoidal isomorphisms. The implementation works by taking the context, $\Gamma \otimes \Delta$, and the two lists of the variables required by the operations; one of which requiring Γ as its context, and the other Δ. From this we can calculate which variables need to be shared and the appropriate permutations to output a **FQC** with the appropriate behaviour. The first $|\Gamma|$ qbits contain Γ, the remaining $|\Delta|$ contain Δ, and the list of variables in these two contexts (for verification purposes). This is arguably the most complicated function in the compiler.

1.5.2 Compiling *Let*

The QML *Let* statement is the standard construction, and has the usual interpretation. *Let* is, however, interesting, as it demonstrates some of the subtleties of context handling in QML. The *Let* rule and diagram are given as:

$$\frac{\begin{array}{c}\Gamma \vdash^a t : \sigma \\ \Delta, x : \sigma \vdash^b u : \tau\end{array}}{\Gamma \otimes \Delta \vdash^{a \sqcap b} \texttt{let } x = t \texttt{ in } u : \tau}\ \text{let}$$



The function begins with a recursive call to compute the subterm *t*, and then extracts some data:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{compileTm}\ c\ \Gamma\ (\mathit{Let}\ x\ t\ u\ tTy)\ mTy =} \\
\quad \textbf{do}\ t^C & \leftarrow \mathit{compileTm}\ c\ \Gamma\ t\ tTy \\
\quad\quad \textbf{let}\ nt & = \mathit{uCon}\ t^C \\
\quad\quad\quad \sigma & = \mathit{ty}\ t^C \\
\quad\quad\quad t^F & = \mathit{fqc}\ t^C
\end{array}
$$

Next, we extend the context Γ with the typed variable $x : \sigma$. We call the extended context Γ', and this is the context that will be passed to functions within the scope of the *Let* statement. This is achieved by calling the auxiliary function *extEnv*, which simply adds the pair (x, σ) to the context environment Γ:

$$
\textbf{let}\ \Gamma' = \mathit{extEnv}\ \Gamma\ (x, \sigma)
$$

The compilation of the subterm over which our new variable has scope, *u*, can now be performed. This is done in the usual way; but passing Γ' as the context.

$$
\begin{array}{ll}
u^C & \leftarrow \mathit{compileTm}\ c\ \Gamma'\ u\ mTy \\
\textbf{let}\ nu & = \mathit{uCon}\ u^C \\
\quad \tau & = \mathit{ty}\ u^C \\
\quad u^F & = \mathit{fqc}\ u^C
\end{array}
$$

We must now verify the subterm *u* did indeed use the new variable *x*. This is done by checking that the name of *x* appears in the list of variables used by the subterm, $nu = \mathit{uCon}\ u^C$. If it is there then it is removed to give us nu', which gives us the list of variables in the context Γ that are used the subterm (giving us Γ from $\Gamma \otimes \Delta$, in the diagram). If *x* does not appear in *nu* then an *Error* is returned and propagated by the Error monad. This is done by the simple auxiliary function *findrm* (find and remove):

$$
nu' \quad \leftarrow \mathit{findrm}\ x\ \sigma\ nu
$$

Using the used context lists from the two subterms we can now calculate ϕ_C, using the function described before. From the diagram, the size of Γ is equal to $a\ t^F$, and the size of Δ in the diagram is given by $a\ c^F - a\ t^F$:

$$
\begin{array}{ll}
\textbf{let}\ (ngd, c^F) & = \phi_C\ \Gamma\ nt\ nu' \\
\quad |\Delta| & = a\ c^F - a\ t^F \\
\multicolumn{2}{l}{\mathit{greturn}\ mTy\ (\mathit{Comp}\{\mathit{uCon} = ngd, ty = \tau,}
\end{array}
$$

Finally, we compute the *FQC* morphism using this data, with ϕ following the construction in the diagram exactly, which is returned after *greturn* verifies the inferred and expected types match:

$$
\begin{array}{l}
\mathit{fqc} = \mathit{FQC}\{a = a\ c^F, h = h\ c^F + h\ t^F + h\ u^F, b = b\ u^F, g = g\ t^F + g\ u^F, \\
\qquad \phi = \bigodot\ [\bigotimes\ [\phi\ c^F, \mathit{idu}\ (h\ t^F + h\ u^F)], \\
\qquad\qquad \mathit{swapN}\ (a\ t^F)\ |\Delta|\ (a\ t^F + |\Delta| + h\ t^F + h\ u^F), \\
\qquad\qquad \bigotimes\ [\mathit{idu}\ |\Delta|, \phi\ t^F, \mathit{idu}\ (h\ u^F)], \\
\qquad\qquad \bigotimes\ [\mathit{idu}\ (a\ u^F), \mathit{swapN}\ (h\ t^F)\ (h\ u^F)\ (h\ t^F + h\ u^F)], \\
\qquad\qquad \bigotimes\ [\phi\ u^F, \mathit{idu}\ (g\ t^F)], \\
\qquad\qquad \bigotimes\ [\mathit{idu}\ (b\ u^F), \mathit{swapN}\ (g\ u^F)\ (g\ t^F)\ (g\ u^F + g\ t^F)]]\}\})
\end{array}
$$

Note how the construction of ϕ follows the construction of the circuit diagram, with each line representing a column of the diagram. The computation of *uCon* is simply the shared context, and the type follows from the computation. The *FQC* constructors *a*, *h*, *b*, and *g* are easily calculated.

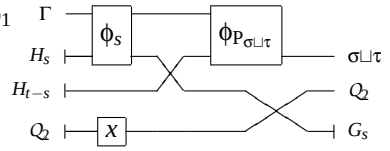
The implementation of the diagram for $\otimes$ introduction, also denoted $t \otimes u$ (and

akin to pairing), is very similar. We simply compile the two subterms, generate the appropriate context for each using ϕ_C, and then construct the *FQC* object following the circuit diagram given in [1].

1.5.3 Compiling *Inl* ($\oplus$)

A third interesting example is the compilation of injections, which shows some of the complications implementing the diagrammatic form of the compilation; specifically, care is required to ensure that the subcomputations are 'plumbed' correctly into the rest of the computation. The rule and compilation diagram for constructing *Inl s* are given as:

$$\frac{\Gamma \vdash^a s : \sigma}{\Gamma \vdash^a \texttt{inl}\ s : \sigma \oplus \tau}\ +\text{intro}_1$$



The rule tells us that from a term s of some type σ we can compute $Inl\ s : \sigma \oplus \tau$. The returned type cannot be inferred from the input as we know nothing about τ. For this reason, in order to compile *Inl*, the *compileTm* function must be passed the expected return type, *mTy* which should be $Just\ \sigma \oplus \tau$. *Inr*, *Case*, and $Case^o$ also need this information for the same reason.

These first few lines of the definition for *Inl* take the type input and split it into its component types using the simple function *unPlusMaybe*. These are then extracted from the Maybe monad using *unMaybeTy* to give the expected return type of the computed left injection.

$$\begin{aligned}
&compileTm\ c\ \Gamma\ (Inl\ s)\ mTy = \\
&\quad \textbf{do}\ (\sigma^?, \tau^?) \leftarrow unPlusMaybe\ mTy \\
&\qquad \sigma' \quad \leftarrow unMaybeTy\ \sigma^? \\
&\qquad \tau' \quad \leftarrow unMaybeTy\ \tau^?
\end{aligned}$$

We next compile the subterm s by calling *compileTm* again, and passing $\sigma^?$ as the expected type of the subterm.

$$\begin{aligned}
&\qquad s^C \quad \leftarrow compileTm\ c\ \Gamma\ s\ \sigma^? \\
&\qquad \textbf{let}\ \sigma \quad = ty\ s^C \\
&\qquad\quad s^F \quad = fqc\ s^C
\end{aligned}$$

There is no restriction that the sizes of σ and τ must be equal, so if the types are of different sizes we have to pad the smaller one to make them the same size, using the padding operator ϕ_P.

$$\begin{aligned}
&\max^{(s,t)} = max\ (size\ \sigma')\ (size\ \tau') \\
&\delta \quad = \max^{(s,t)} - (size\ \sigma) \\
&p^F \quad = \phi_P\ (b\ s^F)\ \delta
\end{aligned}$$

Next we confirm that the inferred type for the subcomputation was indeed the type returned. If they match, then the computation is constructed, directly following the circuit diagram:

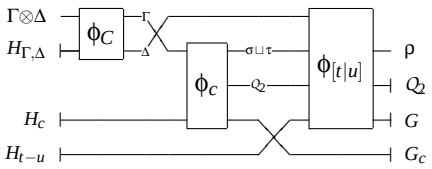
$$
\begin{array}{rl}
\multicolumn{2}{l}{\textbf{if } \sigma \not\equiv \sigma' \textbf{ then } \mathit{Error}\ (\texttt{"Inl type mismatch: "} +\!\!+ \ldots)} \\
\multicolumn{2}{l}{\qquad \textbf{else } \mathit{greturn}\ \mathit{mTy}\ (\mathit{Comp}\{\mathit{uCon} = \mathit{uCon}\ t^C, \mathit{ty} = \sigma \oplus \tau',} \\
\mathit{fqc} = \mathit{FQC}\{a & = a\ s^F, h = \delta + 1, b = \max^{(s,t)} + 1, g = g\ s^F, \\
\phi & = \bigodot\ [\bigotimes\ [\phi\ s^F, \mathit{idu}\ \delta, \mathit{unot}], \\
& \qquad \bigotimes\ [\mathit{idu}\ (b\ s^F), \mathit{swapN}\ (g\ t^F)\ \delta\ (g\ s^F + \delta + 1)], \\
& \qquad \bigotimes\ [\phi\ p^F, \mathit{swapN}\ (g\ s^F)\ 1\ (1 + g\ s^F)]]\}\})
\end{array}
$$

1.5.4 Compiling *Case* & *Case*^{o}

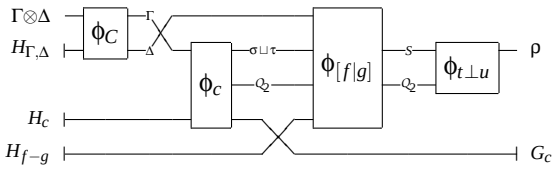
In QML there are two versions of the traditional case construct. The first, *Case*, measures a qbit of the data over which we branch to determine which branch has to be returned. By measuring, it collapses part of the quantum state, and causes entangled states to decohere. The second, *Case*^{o}, is an example of the *quantum control* available in QML. It does not measure the result of executing the conditional statement, but produces a superposition of results which corresponds to the superposition of data it analyses. It is the source of quantum parallelism; intuitively both branches are run in parallel. However, there is a price to pay: we have to provide evidence that both branches are observably different, i.e. orthogonal. Only then are we able to translate our quantum case analysis into quantum circuits.

The rules and circuits for compiling *Case* and *Case*^{o} are:

$$
\frac{\begin{array}{l}\Gamma \vdash c : \sigma \oplus \tau \\ \Delta, x : \sigma \vdash t : \rho \\ \Delta, y : \tau \vdash u : \rho\end{array}}{\Gamma \otimes \Delta \vdash \texttt{case}\ c\ \texttt{of}\ \{\texttt{inl}\ x \Rightarrow t \mid \texttt{inr}\ y \Rightarrow u\} : \rho}\ \oplus - \text{elim}
$$



$$
\frac{\begin{array}{l}\Gamma \vdash^a c : \sigma \oplus \tau \\ \Delta,\ x : \sigma \vdash^o t : \rho \\ \Delta,\ y : \tau \vdash^o u : \rho \quad t \perp u\end{array}}{\Gamma \otimes \Delta \vdash^a \texttt{case}^\circ\ c\ \texttt{of}\ \{\texttt{inl}\ x \Rightarrow t \mid \texttt{inr}\ y \Rightarrow u\} : \rho}\ \oplus - \text{elim}^\circ
$$



Note both of these circuits make use of the context sharing morphism ϕ_C, defined earlier.

Looking at compiling *Case* first, the function begins by computing the *Comp* object for the conditional term c^T, from which the subtypes (via *unPlus*), context required and **FQC** morphism are extracted:

$$
\begin{array}{l}
\mathit{compileTm}\ c\ \Gamma\ (\mathit{Case}\ c^T\ (x,t)\ (y,u))\ \mathit{mTy} = \\
\quad \textbf{do}\ c^C \qquad \leftarrow \mathit{compileTm}\ c\ \Gamma\ c^T\ \mathit{Nothing} \\
\qquad (\sigma,\tau) \qquad \leftarrow \mathit{unPlus}\ (\mathit{ty}\ c^C) \\
\qquad \textbf{let}\ \mathit{nc} \qquad = \mathit{uCon}\ c^C \\
\qquad\quad c^F \qquad = \mathit{fqc}\ c^C
\end{array}
$$

Note that the type passed when compiling c^T is *Nothing*. This is because we cannot infer the type from the information given, and therefore restricts what terms conditional statements can be composed of. Next, two extended contexts are created, $\Gamma^{x:\sigma}$ for the subterm t, and $\Gamma^{y:\tau}$ for u. These are then used in the compilation of the subterms t and u:

$$
\begin{array}{lll}
\Gamma^{x:\sigma} & = \mathit{extEnv}\ \Gamma\ (x,\sigma) \\
\Gamma^{y:\tau} & = \mathit{extEnv}\ \Gamma\ (y,\tau) \\
t^C & \leftarrow \mathit{compileTm}\ c\ \Gamma^{x:\sigma}\ t\ \mathit{mTy} \\
u^C & \leftarrow \mathit{compileTm}\ c\ \Gamma^{y:\tau}\ u\ \mathit{mTy}
\end{array}
$$

Similar to the implementation of the *Let* construct, we now ensure the added variables were used and remove them from the appropriate list (or else fail). The types and *FQC* objects are then extracted:

$$
\begin{array}{lll}
\mathit{nt} & \leftarrow \mathit{findrm}\ x\ \sigma\ (\mathit{uCon}\ t^C) \\
\mathit{nu} & \leftarrow \mathit{findrm}\ y\ \tau\ (\mathit{uCon}\ u^C) \\
\textbf{let}\ \rho^t & = \mathit{ty}\ t^C \\
\rho^u & = \mathit{ty}\ u^C \\
t^F & = \mathit{fqc}\ t^C \\
u^F & = \mathit{fqc}\ u^C
\end{array}
$$

We can now compute the context sharing using ϕ_C, to give us Γ and Δ *from the diagram*, and then compute the $[t|u]^F$ morphism. This is again done by an auxiliary function, *condOp*, which creates an *FQC* object using the *Cond* operator on t^F and u^F, appropriately padded so they are the same size, and plumbed correctly. Some useful mnemonics are then created to help in the definition of ϕ:

$$
\begin{array}{lll}
(\_, C^F) & = \phi_C\ \Gamma\ \mathit{nc}\ \mathit{nt} \\
[t|u]^F & = \mathit{condOp}\ u^F\ t^F\ \mathit{True} \\
|\Gamma| & = a\ c^F \\
|\Delta| & = a\ C^F + h\ C^F - |\Gamma| \\
\mathit{arity} & = |\Gamma| + |\Delta| + h\ c^F + h\ [t|u]^F
\end{array}
$$

We now begin to prepare the *Comp*, but first must ensure that that t and u make use of the same variables from the original context Γ; otherwise there is a context error, and that ρ^t and ρ^u also agree:

$$
\begin{array}{l}
\quad \textbf{if}\ \mathit{nt} \not\equiv \mathit{nu}\ \textbf{then}\ \mathit{Error}\ \texttt{"Context error"} \\
\qquad \textbf{else if}\ \rho^t \not\equiv \rho^u\ \textbf{then}\ \mathit{Error}\ \texttt{"Type error"} \\
\qquad \textbf{else}\ \mathit{greturn}\ \mathit{mTy}\ (\mathit{Comp}\{\mathit{uCon} = \mathit{nc} +\!\!+\!\!< \mathit{nt}, \mathit{ty} = \rho^t,
\end{array}
$$

Finally, the construction of the *FQC* can begin. The values of a, h, b, and g can

be read from the diagram, and again the morphism is constructed using permutations and the subcomputations generated above, to match the construction of the diagram:

$$\begin{aligned}
fqc = FQC\{\, & a = a\ C^F, \quad h = && h\ C^F + h\ c^F + h\ [t|u]^F,\\
& b = b\ [t|u]^F, g = && g\ [t|u]^F + g\ c^F,\\
& \phi = \bigodot [\, \bigotimes [\phi\ C^F, idu\ (h\ c^F + h\ [t|u]^F)],\\
& \qquad swapN\ |\Gamma|\ |\Delta|\ arity,\\
& \qquad \bigotimes [idu\ |\Delta|, \phi\ c^F, idu\ (h\ [t|u]^F)],\\
& \qquad \bigotimes [\ idu\ (|\Delta| + b\ c^F),\\
& \qquad\quad swapN\ (g\ c^F)\ (h\ [t|u]^F)\ (g\ c^F + h\ [t|u]^F)],\\
& \qquad \bigotimes [\phi\ [t|u]^F, idu\ (g\ c^F)]]\,\})
\end{aligned}$$

As the diagram for $Case^o$ is very similar to that for *Case*, so is the compilation function. Only the differences are discussed:

$$\begin{aligned}
&compileTm\ c\ \Gamma\ (Case^o\ c^T\ (x,t)\ (y,u)\ o)\ mTy =\\
&\quad \textbf{do}\ c^C && \leftarrow compileTm\ c\ \Gamma\ c^T\ Nothing\\
&\qquad (\sigma,\tau) && \leftarrow unPlus\ (ty\ c^C)\\
&\qquad \textbf{let}\ nc && = uCon\ c^C\\
&\qquad\quad c^F && = fqc\ c^C\\
&\qquad\quad \Gamma^{x:\sigma} && = extEnv\ \Gamma\ (x,\sigma)\\
&\qquad\quad \Gamma^{y:\tau} && = extEnv\ \Gamma\ (y,\tau)
\end{aligned}$$

The next step in the implementation of *Case* is to compile the two subterms, t and u, which are passed to the *condOp* to give $[t|u]^F$. However, note that in this case, from the diagram, we want $[f|g]^F$, where f and g are computations based on t and u, but modified using the orthogonality judgement. This is carried out by the *orthcomp* function, see section 1.5.5. If this function fails, then we know that the terms t and u were not orthogonal. The compilation then proceeds as in *Case*:

$$\begin{aligned}
&(f^C, g^C, \rho) && \leftarrow orthcomp\ c\ (t,\sigma,\Gamma^{x:\sigma})\ (u,\tau,\Gamma^{y:\tau})\ mTy\ o\\
&nf && \leftarrow findrm\ x\ \sigma\ (uCon\ f^C)\\
&ng && \leftarrow findrm\ y\ \tau\ (uCon\ g^C)\\
&\textbf{let}\ f^S && = ty\ f^C\\
&\quad g^S && = ty\ g^C\\
&\quad f^F && = fqc\ f^C\\
&\quad g^F && = fqc\ g^C\\
&\quad (_, C^F) && = \phi_C\ \Gamma\ nc\ nf\\
&\quad [f|g]^F && = condOp\ g^F\ f^F\ False\\
&\quad |\Gamma| && = a\ c^F\\
&\quad |\Delta| && = a\ C^F + h\ C^F - |\Gamma|\\
&\quad arity && = |\Gamma| + |\Delta| + h\ c^F + h\ [f|g]^F\\
&\quad orth && = orth2unitary\ f^S\ g^S\ o
\end{aligned}$$

Now we must compile the orthogonality judgment morphism. This is done using the auxiliary function *orth2unitary*, which takes the type of f and g, and the judgment o. This function simply computes the circuit (*Unitary*) for the orthogonality judgment, and is explained in section 1.5.5.

$$\textbf{if}\ nf \not\equiv ng\ \textbf{then}\ Error\ \texttt{"Context error"}$$

else if $f^S \not\equiv g^S$ **then** *Error* `"Type error"`
else if *orthcheck t u o* $\equiv$ *False*
then *Error* `"Orthogonality Failure"`
else *greturn mTy* (*Comp*{ *uCon* $=$ *nc* $+\!\!+\!<$ *nf*, *ty* $=\rho$,

The computation of the *FQC* morphism continues in the exact same way as before, but for the addition of the extra *orth* construct, as in the final column of the diagram:

$$
\begin{aligned}
fqc = FQC\{\, & a = a\,C^F, \quad h = h\,C^F + h\,c^F + h\,[f|g]^F, \\
& b = b\,[f|g]^F, g = g\,c^F, \\
& \phi = \odot\,[\otimes\,[\phi\,C^F, idu\,(h\,c^F + h\,[f|g]^F)], \\
& \qquad swapN\,|\Gamma|\,|\Delta|\,arity, \\
& \qquad \otimes\,[idu\,|\Delta|, \phi\,c^F, idu\,(h\,[f|g]^F)], \\
& \qquad \otimes\,[\quad\quad idu\,(|\Delta| + b\,c^F), \\
& \qquad\quad\quad\quad swapN\,(g\,c^F)\,(h\,[f|g]^F)\,(g\,c^F + h\,[f|g]^F)], \\
& \qquad \otimes\,[\phi\,[f|g]^F, idu\,(g\,c^F)], \\
& \qquad \otimes\,[orth, idu\,(g\,c^F)]]\,\}\,\})
\end{aligned}
$$

1.5.5 Compiling *Orth* ($\perp$)

Here we briefly introduce the type of orthogonality judgments, *Orth*, and discuss how the *orth2unitary* function and the *orthcomp* function, used in $Case^o$, behave for some examples of the *Orth* type. The *Orth* type is defined as:

data *Orth* $=$ *OInlr* | *OInrl*
| *OInl Orth* | *OInr Orth*
| *OPairl* | *OPairr*
| *OSup Orth Orth* $(\mathbb{C},\mathbb{C})\ (\mathbb{C},\mathbb{C})$

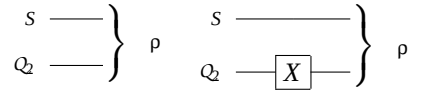
which matches the seven rules given in [1].

The idea of $t \perp u$ is that there exists a boolean observation which tells the two terms apart in every environment. Given $\Gamma \vdash t, u : \rho$, the interpretation $[\![t \perp u]\!] = (S_{t\perp u}, \phi_{t\perp u})$, where $\phi_{t\perp u} \in S_{t\perp u} \otimes \mathcal{Q}_2 \multimap_{\text{unitary}} [\![\rho]\!]$ is defined by induction over the derivation.

OInlr **&** *OInrl*

$$\frac{}{\texttt{inl}\ t \perp \texttt{inr}\ u}\ OInlr \qquad \frac{}{\texttt{inr}\ t \perp \texttt{inl}\ u}\ OInrl$$

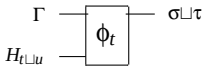
Given $\Gamma \vdash^\circ t : \sigma$ and $\Gamma \vdash^\circ u : \tau$, we define $S = \sigma \sqcup \tau$ and $\phi_{t\perp u}$ is simply the identity in the first case but negates the qbit in the second.



This can be defined by the function *orth2unitary* as:
orth2unitary $\in$ *Ty* $\to$ *Ty* $\to$ *Orth* $\to$ *Unitary*

$$\mathit{orth2unitary}\quad t\quad u\quad (\mathit{OInlr}) = \bigotimes [\mathit{idu}\ (\mathit{max}\ (\mathit{size}\ t)\ (\mathit{size}\ u)), \mathit{idu}\ 1]$$
$$\mathit{orth2unitary}\quad t\quad u\quad (\mathit{OInrl}) = \bigotimes [\mathit{idu}\ (\mathit{max}\ (\mathit{size}\ t)\ (\mathit{size}\ u)), \mathit{Not}]$$

For both of these rules, ϕ_f is simply given by ϕ_t:



while ϕ_g is given by ϕ_u, in the same way. This is also straightforward:

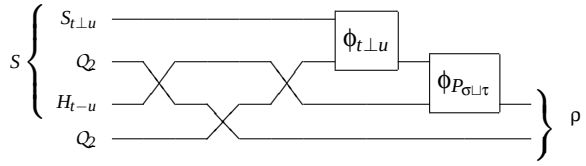
$$\begin{aligned}&\mathit{orthcomp} \in \mathit{Code} \to (\mathit{Tm}, \mathit{Ty}, \mathit{Con}) \to (\mathit{Tm}, \mathit{Ty}, \mathit{Con})\\ &\qquad\qquad \to \mathit{Maybe}\ \mathit{Ty} \to \mathit{Orth} \to \mathit{Error}\ (\mathit{Comp}, \mathit{Comp}, \mathit{Ty})\\ &\mathit{orthcomp}\ c\ (\mathit{Inl}\ t, \sigma, \Gamma^{x:\sigma})\ (\mathit{Inr}\ u, \tau, \Gamma^{y:\tau})\ \mathit{mTy}\ (\mathit{OInlr}) =\\ &\quad \mathbf{do}\ f^C \leftarrow \mathit{compileTm}\ c\ \Gamma^{x:\sigma}\ t\ (\mathit{Just}\ \sigma)\\ &\qquad g^C \leftarrow \mathit{compileTm}\ c\ \Gamma^{y:\tau}\ u\ (\mathit{Just}\ \tau)\\ &\qquad t^C \leftarrow \mathit{compileTm}\ c\ \Gamma^{x:\sigma}\ (\mathit{Inl}\ t)\ \mathit{mTy}\\ &\qquad \mathit{return}\ (f^C, g^C, \mathit{ty}\ t^C)\end{aligned}$$

OInr is identical. Note that the function also returns the type of the full subterm, for use by Case^o. The simplest way of doing this (though least efficient) is to compute the full subterm as well, and just extract the type.

OInl Orth

$$\frac{t \perp u}{\mathtt{inl}\ t \perp \mathtt{inl}\ u \quad \mathtt{inr}\ t \perp \mathtt{inr}\ u}\ \mathit{OInl}$$

Consider the first rule, *OInl*, only: We set $S = S_{\mathtt{inl}\ t \perp \mathtt{inl}\ u} = S_{t\perp u} \otimes Q_2$, to derive $\phi_{\mathtt{inl}\ t\perp\mathtt{inl}\ u} \in (S_{t\perp u} \otimes Q_2) \otimes Q_2 \multimap_{\text{unitary}} [\![\sigma \sqcup \tau]\!] \otimes Q_2$ from $\phi_{t\perp u} \in S_{t\perp u} \otimes Q_2 \multimap_{\text{unitary}} [\![\sigma]\!]$ we just swap two qbits. The other case proceeds analogously.

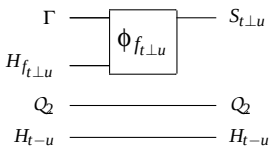


The code for creating this using the *orth2unitary* function is simply:

$$\begin{aligned}\mathit{orth2unitary}\ t\ u\ (\mathit{OInl}\ o) = \bigodot [&\bigotimes [\mathit{idu}\ \max^{(t,u)}, \mathit{swapN}\ 1\ \delta^{(t,u)}\ (\delta^{(t,u)}+2)],\\ &\bigotimes [\mathit{idu}\ (\max^{(t,u)}+\delta^{(t,u)}), \mathit{Perm}\ [1,0]],\\ &\bigotimes [\mathit{idu}\ \max^{(t,u)}, \mathit{swapN}\ 1\ \delta^{(t,u)}\ (\delta^{(t,u)}+2)],\\ &\bigotimes [\mathit{orth2unitary}\ t\ u\ o, \mathit{idu}\ (\delta^{(t,u)}+1)],\\ &\bigotimes [\phi\ (\phi_P\ (\max^{(t,u)}+1)\ \delta^{(t,u)}), \mathit{idu}\ 1]]\end{aligned}$$
$$\begin{aligned}\textbf{where}\ \max^{(t,u)} &= \mathit{max}\ (\mathit{size}\ t)\ (\mathit{size}\ u)\\ \delta^{(t,u)} &= \mathit{abs}\ ((\mathit{size}\ t) - (\mathit{size}\ u))\end{aligned}$$

$$\mathit{orth2unitary}\ t\ u\ (\mathit{OInr}\ o) = \mathit{orth2unitary}\ t\ u\ (\mathit{OInl}\ o)$$

Calculating ϕ_f and ϕ_g is more involved, as $\phi_{f_{t\perp u}}$ and $\phi_{g_{t\perp u}}$. must also be calculated. In the case where we have $\mathtt{inl} \perp \mathtt{inl}$, ϕ_f is given by:

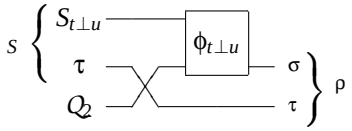


Note the output, $S_{t\perp u} \otimes Q_2 \otimes H_{t-u}$, includes S. ϕ_g can be constructed in the same way, substituting $\phi_{g_{t\perp u}}$ for $\phi_{f_{t\perp u}}$. In the case where we have `inr` $\perp$ `inr` , the construction of ϕ_f (and ϕ_g) is identical, except for a negation on the final wire – the control qbit. The code for *orthcomp*, which calculates these trivial circuits, is not shown here.

OPairl

$$\frac{t \perp u}{(t,v) \perp (u,w) \quad (v,t) \perp (w,u)}\ \mathit{OPairl}$$

Considering only the first rule above, given $\Gamma \vdash^\circ t,u : \sigma$ and $\Gamma \vdash^\circ v,w : \tau$. We set $S = S_{(t,v)\perp(u,w)} = S_{t\perp u} \otimes \tau$, to derive $\phi_{(t,v)\perp(u,w)} \in S_{t\perp u} \otimes \tau \otimes Q_2 \multimap_{\text{unitary}} [\![\sigma \otimes \tau]\!]$ from $\phi_{t\perp u} \in S_{t\perp u} \otimes Q_2 \multimap_{\text{unitary}} [\![\sigma]\!]$ requires simple rewiring.



This is computed by:

$$\begin{aligned}
\mathit{orth2unitary}\ \sigma\ \tau\ \mathit{OPairl} = \bigodot\ [\ \bigotimes\ [&\mathit{idu}\ |\sigma'|,\\
&\mathit{swapN}\ |\tau|\ 1\ (|\tau'|+1)],\\
\bigotimes\ [&\mathit{orth2unitary}\ \sigma'\ \tau'\ (\mathit{OInrl}),\\
&\mathit{idu}\ |\tau'|\ ]\ ]
\end{aligned}$$

$$\begin{aligned}
\textbf{where}\ |\sigma'| &= \mathit{size}\ s'\\
|\tau'| &= \mathit{size}\ t'\\
\mathit{OK}\ (\sigma',\tau') &= \mathit{unPair}\ \tau
\end{aligned}$$

To calculate ϕ_f in this instance, we make use of ϕ_t and ϕ_v, to construct $\phi_t \otimes \phi_v$, which is an application of the $\otimes$ - introduction rule, and can therefore be calculated simply by a call to *compileTm* $(t \otimes v)$. ϕ_g is constructed analogously, replacing ϕ_t with ϕ_u and ϕ_v with ϕ_w. For the case where the second rule is applied, the same constructions are used, but with t and v (u and w for ϕ_g) swapped appropriately.

1.6 CONCLUSION

Our present work exploits the functional paradigm in two ways: first we have designed a high level quantum programming language — QML — using con-

cepts from conventional functional programming [1]; and then in the present paper we have used Haskell for a prototypical, high level implementation of QML. In related work [8] use Haskell to explore the semantic foundation of QML by implementing Super operators using the arrows [3].

ACKNOWLEDGEMENTS

Thanks are extended by the authors to Conor McBride, Peter Selinger, Amr Sabry, Juliana Vizzotto, Graham Hutton, and Janine Swarbrick.

REFERENCES

[1] T. Altenkirch and J. Grattage. A functional quantum programming language. quant-ph/0409065, November 2004.

[2] M. Hirvensalo. *Quantum Computating*. Springer-Verlag NewYork, Inc., 2001.

[3] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[4] M. Nielsen and I. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, Cambridge, 2000.

[5] P. Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 2004.

[6] P. Selinger and B. Valiron. A lambda calculus for quantum computation with classical control. accepted for TLCA05, 2005.

[7] A. van Tonder. A lambda calculus for quantum computation. quant-ph/0307150, 2003. to appear in SIAM Journal of Computing.

[8] J. K. Vizzotto, T.Altenkirch, and A. Sabry. Structuring quantum effects: Superoperators as arrows. submitted for publication, 2004.

[9] P. Zuliani. *Quantum Programming*. PhD thesis, Oxford University, 2001.