Typechecking Linear Data: Quantum Computation in Haskell

Richard Eisenberg
University of Pennsylvania
eir@seas.upenn.edu

Benoît Valiron
University of Pennsylvania
valiron@seas.upenn.edu

Steve Zdancewic
University of Pennsylvania
stevez@cis.upenn.edu

Abstract

This paper demonstrates how to use type classes to get Haskell's type inference engine to infer the types of an embedded language for quantum computation. The embedded language, a variant of Selinger & Valiron's *quantum lambda calculus* (QLC), uses linear types to express the non-duplicability of quantum data, along with subtyping to allow convenient mixing of classical data with quantum data. The embedding makes sophisticated use of Haskell's type classes to do logic programming at the type level.

1. Introduction

This paper shows how to hijack Haskell's type inference algorithm to infer types for an embedded language. More specifically, this paper shows how to encode non-trivial linearity constraints, including a subtyping relation, using the Haskell typechecker. We demonstrate the technique by embedding a variant of Selinger & Valiron's quantum lambda calculus (QLC) [10]. The technique makes heavy use of Haskell's type classes—functional dependencies and equality constraints guide the unification algorithm to typecheck embedded terms.

Programming languages tailored to the situation of quantum computing enable researchers to concisely describe such algorithms, abstracting the complexities of the underlying physics [2]. In this regard, Haskell provides a compelling framework for experimenting with quantum computing language designs, partly because Haskell provides good support for embedded domain-specific languages, and partly because Haskell's monads provide a natural way of handling quantum side effects.

Indeed, Green and Altenkirch [3], and Vizzotto, da Rocha Costa, and Sabry [12] have previously investigated interfacing quantum computation with Haskell, using a quantum IO-monad and with arrows, respectively.

Physics imposes several unusual constraints on the kinds of programs that can be expressed in models of quantum computation [8]. In particular, quantum data (built from *qubits*) must be treated *linearly*—it cannot be duplicated—and observing it in a computation yields a probabilistic result. As a consequence, any programming language for describing quantum computations must take into account these rather nonstandard properties. In exchange for adhering to these physical constraints, such programs can potentially harness quantum properties in new algorithms that go beyond what can easily be expressed by classical computers [8].

This paper's goal, like that of the previous works, is to obtain a "natural" way of describing quantum features in Haskell. Unlike the earlier works, however, here we focus on the *linearity* aspects of quantum data. As we explain, one novel aspect of the encoding is that it fully piggy-backs on Haskell's type inference algorithm to infer the types of terms in the embedded language.

The Haskell folklore has it that type classes with functional dependencies are very versatile [5, 7, 9]. The use proposed here is another concrete example of their logic programming capabilities. The result is a direct embedding of QLC, a previously designed language that is consistent with one of the common models of quantum computation (i.e. the QRAM model). We believe that this approach to expressing type constraints for an embedded language is general enough to be applied in other domains.

Before describing the Haskell encoding (Sections 3 through 6), we first explain the details of the QLC, mainly by way of example. Section 7 shows the usability of the embedding and revisits the examples in their Haskell form. Section 8 sketches how to attach a monad to the embedding for concrete experimentation and Section 9 concludes with some discussion of related approaches.

Software and code. The Haskell code in this paper is available in full on the web [11]. We used GHC version 7.4.1 with some extensions listed in Appendix A.

2. QLC: A lambda-calculus for quantum computation

A quantum program can essentially be viewed as a classical program that interacts with a random access memory bank (the QRAM) with non-standard properties [6]. The memory holds *qubits*, the quantum equivalent of classical Booleans.

This section presents QLC, a language for interacting with a QRAM. A variant of the Selinger-Valiron quantum lambda-calculus [10], QLC is a typed, functional, call-by-value language. The salient properties of the QRAM model are presented along with the language.

2.1 QLC terms

QLC is an ML-style language, featuring term variables $x, y, z \ldots$ and terms $r, s, t, \ldots$. At their core, terms are made of the usual lambda-calculus constructors

$$x \mid \lambda x.s \mid (s)t.$$

As usual for a linearly typed language, there are no direct projections for pairs; instead, a *let*-notation is used to extract the components:

$$\langle s, t\rangle \mid let\ \langle x, y\rangle = s\ in\ t.$$

For convenience, we use the macros

$$\begin{array}{lcl} let\ x = s\ in\ t & := & ((\lambda x.t)s), \\ \lambda\langle x, y\rangle.s & := & \lambda t.(let\ \langle x, y\rangle = t\ in\ s). \end{array}$$

The language supports classical Booleans through tt, ff and the *if-then-else* construct:

$$tt \mid ff \mid if\ s\ then\ t\ else\ t'.$$

Support for access to the QRAM is given by the four term constructs:

$$new(s) \mid H(s) \mid CNOT(s) \mid meas(s).$$

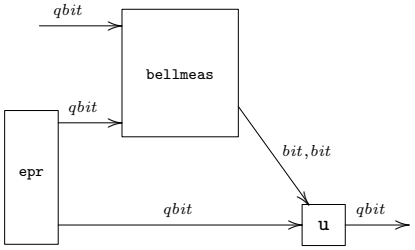
The construct $new(s)$ allocates a new qubit in the memory and sets its value to the qubit corresponding to s (assumed to be a Boolean). The output of the function is a pointer to the corresponding qubit. With $meas$ (standing for "measure"), one can retrieve a classical Boolean out of a qubit. Because of the physics behind the QRAM, this operation is *probabilistic*, and it *destroys* the quantum bit. Finally, H and $CNOT$ are two operations on qubits, returning their input. The operation H acts upon one qubit, whereas $CNOT$ acts upon a pair of (distinct) qubits.

The simple program

$$meas\ (H\ (new\ ff))$$

first creates a qubit initialized to (quantum) false, then applies H on it, then measures it. The output of the program is the classical Boolean, that is the output of *meas*. In quantum computing, this program tosses a coin: it returns either tt or ff with equal probability.

A more involved program encodes so-called "quantum teleportation". This algorithm captures the spirit of the strange nature of quantum computation. It consists of three parts, summarized by this diagram:



The input qubit on top of the diagram is in the same state as the one coming at the bottom right: it is "as if" one had encoded a qubit into the two classical bits flowing from `bellmeas` to `u`.

In QLC, the code is as follows.

$$\texttt{epr} = \quad \lambda x.CNOT\,\langle H\,(new ff), (new ff)\rangle$$

$$\texttt{bellmeas} = \quad \lambda x.\lambda y.let\ \langle z,t\rangle = CNOT\langle x,y\rangle \\ in\ \langle meas\,(Hz), meas\,t\rangle$$

$$\texttt{u} = \quad \lambda z.\lambda\langle x,y\rangle.if\ x \\ then\ (if\ y\ then\ U_1\ z\ else\ U_2\ z) \\ else\ (if\ y\ then\ U_3\ z\ else\ U_4\ z)$$

$$\texttt{telep} = \quad let\ \langle x,y\rangle = \texttt{epr}\,ff\ in \\ let\ z = \texttt{bellmeas}\,x\ in \\ let\ t = \texttt{u}\,y\ in \\ \lambda x.t(zx)\rangle$$

where the U_i's are one-qubit operations. The function `epr` does not use its input: it merely constructs two qubits with `new` and does some operation on them before returning them. The function `bellmeas` takes two qubits x and y, processes them and ultimately returns a pair of classical Booleans, created with `meas`. The function `u` takes a qubit z and a pair of classical bits $\langle x,y\rangle$. It applies an operation on z depending on the values of the bits, and returns the result.

Finally, the program `telep` constructs the diagram above: it creates two qubits $\langle x,y\rangle$ with `epr` and then feeds x to `bellmeas` and y to `u`. That is, z inputs a qubit and outputs a pair of bits, whereas t inputs a pair of bits and returns a qubit. The final operation is the composition of these operations: it is a map from a qubit to a qubit.

Note that since the code builds two auxiliary quantum bits, the function `telep` is non-duplicable.

2.2 QLC's type system

As described so far, the language is completely untyped. As usual, this fact allows us to write non-sensical code, such as $meas\ (\lambda x.x)$. The usual type constructs are sufficient to prevent these run-time errors: basic types bit and $qbit$ and type constructors for functions $(\to)$ and for pairing $(\otimes)$.

However, due to the peculiarity of the QRAM, this is not quite enough: a qubit is a non-duplicable piece of data. This means that duplicating a pointer to a qubit can yield a run-time error:

$$let\ x = new\ tt\ in\ CNOT\langle\, x, x\,\rangle$$

will try to apply the operation $CNOT$ on two copies of the same qubit, a prohibited operation, as described in Section 2.1. Another source of error comes from measurement: if s constructs a qubit,

$$let\ x = s\ in\ \langle\, meas\ x, meas\ (Hx)\,\rangle$$

is not well-defined. Once we are done with one of the branches of the pair, the variable x does not point to any valid qubit anymore, as the measurement is destructive.

The solution proposed is to use a *linear* type system. A term is non-duplicable, unless explicitly typed as such, and, by default, functions are to use their argument only once. The type system is defined as follows:

$$\begin{array}{lll} A, B & ::= & !^m bit \mid !^m qbit \mid !^m(A \to B) \mid !^m(A \otimes B), \\ m & ::= & \texttt{True} \mid \texttt{False}. \end{array}$$

The flag "!" is used to indicate whether a term is duplicable or not: If the term s is of type $!^m A$, it is duplicable if $m = \texttt{True}$ and it is not if $m = \texttt{False}$. There is a canonical subtyping relation $A <: B$ driven by the remark that a duplicable term does not need to be duplicated. In particular, $!^{\texttt{True}} A <: !^{\texttt{False}} A$. The formal rules for the relation is as follows:

$$\frac{}{!^m A <: A} \qquad \frac{A <: A' \quad B <: B' \quad m <: n}{!^m(A' \to B) <: !^n(A \to B')}$$

$$\frac{A <: A' \quad B <: B' \quad m <: n}{!^m(A \otimes B) <: !^n(A' \otimes B')}$$

The notation $m <: n$ means "$m = \texttt{True}$ or $n = \texttt{False}$". Note that as expected, the type constructor $(\to)$ is covariant on the right and contravariant on the left.

The typing rules for the language are given in Table 1, and they define a variant of the linear lambda-calculus with subtyping. We use the following conventions:

- When multiple contexts Δ and Γ are used, it is assumed that they are independent, sharing no variables.
- $!\Delta$ denotes a context containing only duplicable types.
- When the context is clear, we write A and $!A$ respectively instead of $!^{\texttt{False}} A$ and $!^{\texttt{True}}$.

We briefly discuss the typing rules.

- The typing rule for a variable term makes use of the subtyping relation. With this rule, it is possible to show that if $\Delta \vdash s : A$ and $A <: B$, then $\Delta \vdash s : B$, as one would expect.
- There are two rules for the lambda-abstraction: one for typing a non-duplicable lambda-abstraction, having no particular constraints, and one for typing duplicable ones. Since the language

$$\frac{A <: B}{\Delta, x : A \vdash x : B} \quad \frac{\Delta, x : A \vdash B}{\Delta \vdash \lambda x.s : A \to B} \quad \frac{!\Delta, \Gamma, x : A \vdash s : B \quad \mathrm{dom}(\Delta) \cup \{x\} \supseteq \mathrm{FV}(s)}{!\Delta, \Gamma \vdash \lambda x.s : !(A \to B)} \quad \frac{!\Delta, \Gamma_1 \vdash s : A \to B \quad !\Delta, \Gamma_2 \vdash t : A}{!\Delta, \Gamma_1, \Gamma_2 \vdash (s)t : B}$$

$$\frac{!\Delta, \Gamma_1 \vdash s : !^{m \vee n} A \quad !\Delta, \Gamma_2 \vdash t : !^{m \vee o} B}{!\Delta, \Gamma_1, \Gamma_2 \vdash \langle s, t \rangle : !^m(!^n A \otimes !^o B)} \otimes_I \quad \frac{!\Delta, \Gamma_2 \vdash s : !^o(!^m A \otimes !^n B) \quad !\Delta, \Gamma_1, x : !^{m \vee o} A, y : !^{n \vee o} B \vdash t : C}{!\Delta, \Gamma_1, \Gamma_2 \vdash \mathit{let}\ \langle x, y \rangle = s\ \mathit{in}\ t : C} \otimes_E$$

$$\frac{}{\Delta \vdash \mathit{tt}, \mathit{ff} : !^m \mathit{bit}} \quad \frac{!\Delta, \Gamma_1 \vdash s : \mathit{bit} \quad !\Delta, \Gamma_2 \vdash t, t' : A}{!\Delta, \Gamma_1, \Gamma_2 \vdash \mathit{if}\ s\ \mathit{then}\ t\ \mathit{else}\ t' : A}$$

$$\frac{\Delta \vdash s : \mathit{bit}}{\Delta \vdash \mathit{new}(s) : \mathit{qbit}} \quad \frac{\Delta \vdash s : \mathit{qbit}}{\Delta \vdash \mathit{meas}(s) : !^m \mathit{bit}} \quad \frac{\Delta \vdash s : \mathit{qbit}}{\Delta \vdash \mathit{had}(s) : \mathit{qbit}} \quad \frac{\Delta \vdash s : \mathit{qbit} \otimes \mathit{qbit}}{\Delta \vdash \mathit{CNOT}(s) : \mathit{qbit} \otimes \mathit{qbit}}$$

Table 1. Typing rules for the quantum lambda-calculus

is call-by-value, a lambda-abstraction is only duplicable when all of its constituents are duplicable. A duplicable closure over non-duplicable terms is disallowed.

- The typing rule for the application is a good example of where duplicable and non-duplicable variables play a role. The only way of having the same free variable appearing both in s and in t is to have it in $!\Delta$, making sure it is duplicable.
- Concerning the tensor, the rules $\otimes_I$ and $\otimes_E$ use the same philosophy regarding the typing context. The flags on the "!" are set up so that a duplicable pair is made of duplicable elements, and a pair of duplicable elements is duplicable.
- In the *if-then-else* rule, the terms t and t' share the same typing context, as usual in a linear setting. In particular, if a non-duplicable variable appears in t, it also appears in t'. However, if a variable appears both in s and in t, then it has to be duplicable.
- Finally, the Booleans can be seen as duplicable or not, and they are of type bit. The term constructor new takes a Boolean and return a non-duplicable quantum bit, the constructor meas takes a quantum bit and returns a bit, the H operation takes a quantum bit and returns a quantum bit, and the CNOT operation acts on a pair of quantum bits.

  Note that all the term constructs that build quantum bits make them non-duplicable.

Teleportation algorithm, revisited. We can now type the teleportation algorithm and its components, as follows:

$$\begin{array}{lcl} \texttt{epr} & : & A \to \mathit{qbit} \otimes \mathit{qbit}, \\ \texttt{bellmeas} & : & \mathit{qbit} \to \mathit{qbit} \to \mathit{bit} \otimes \mathit{bit}, \\ \texttt{u} & : & \mathit{qbit} \to (\mathit{bit} \otimes \mathit{bit}) \to \mathit{qbit}, \\ \texttt{telep} & : & \mathit{qbit} \to \mathit{qbit}. \end{array}$$

The function `telep` is the composition of $\texttt{bellmeas}\, x$ of type $\mathit{qbit} \to \mathit{bit} \otimes \mathit{bit}$ and of $\texttt{u}\, y$ of type $(\mathit{bit} \otimes \mathit{bit}) \to \mathit{qbit}$. It *cannot* be typed with $!(\mathit{qbit} \to \mathit{qbit})$: it is a non-duplicable function.

3. Haskell Encoding: A simply-typed lambda-calculus with named variables

This section develops the methodology used for encoding typing judgments. We first present the general techniques for embedding a simply-typed lambda-calculus in Haskell. The later sections will be devoted to upgrading this techniques to QLC.

This simply-typed lambda-calculus has variables denoted with natural numbers: the natural numbers are the set of available variable *names*. The syntax is therefore

$$\begin{array}{lcl} s, t & ::= & n \mid \lambda n.s \mid (s)t \\ A, B & ::= & X \mid A \to B \end{array}$$

The encoding in Haskell is using the class `Judg` and its instance `J::* → * → *`. The type `J c a` denotes a typing judgment whose context is `c` and whose term is of type `a`. Formally, `Judg` and `J` are defined as follows:

```
class Judg j where
  var :: (GetVar x a c) ⇒ x → j c a
  appl :: (Zip c1 c2 c3) ⇒
    j c1 (a → b) → j c2 a → j c3 b
  lamb :: (AddVar x a c2 c1) ⇒
    x → j c1 b → j c2 (a → b)

data J c a = J (c → a)
instance Judg J where
  lamb x (J f) = J $ λc a → f (add x a c)
  appl (J f) (J g) =
    J $ λx → let (x1,x2) = split x in f x1 (g x2)
  var x = J $ λz → get_var z x
```

In order to type-check our variables, it is necessary to use a type-level encoding of the natural numbers, using `Z` and `S n`. These types are singleton types, so the name of a variable can also be known at runtime.

```
data Z = Z
data S a = S a
```

The typing context is a tower of pairs

$$(A_0, (A_1, \ldots, (A_n, ()) \ldots))$$

where the type A_i is the type of the variable named i. The type operators `Used` and `Free`, both of kind $\star \to \star$, record which variables are used and free by tagging the types.

```
data Used a = Used a
data Free a = Free a
```

For example, the typing judgment

$$x : \mathtt{a}, y : \mathtt{u}, z : \mathtt{a} \to \mathtt{b} \vdash z\, x : \mathtt{b}$$

is the term

```
let x = Z in
let y = S Z in
let z = S (S Z) in
appl (var z) (var x) ::
  J (Used a, (Free u, (Used (a → b), ()))) b
```

The following sections describe the typing rules and the constraints that are being used in the class `Judg` and its instance `J`.

3.1 Typing a variable

The `GetVar` class and its instances type a variable expression. The constraint `GetVar x a c` states that `c` contains only the used variable `x` of type `a` (all others are free). The constraint is defined as a relation in logic programming: each instance describes a property that has to be satisfied by the relation. If it were to be defined in prolog, we would write:

```
GetVar(Z, a, [Used a]).
GetVar(S n, a, (Free u):c) :- GetVar(n,a,c)
```

The corresponding type-class is

```
class GetVar x a c | x a → c where
  get_var :: c → x → a
instance GetVar Z a (Used a,()) where
  get_var (Used a,c) Z = a
instance GetVar n a c ⇒
         GetVar (S n) a (Free u,c) where
  get_var (u,c) (S n) = get_var c n
```

Typing a variable is both a base case in our formal typing rules and also the place where the Haskell type checker bottoms out. Because the Haskell type of a variable expression (`Z` or `S n`) is easy to infer, the type checker builds its proof of a term's type from its variables. The `GetVar` class instances direct this process, by forcing the type checker to build a context exactly big enough to hold a type for the variable under consideration. The `GetVar` class enforces a relationship among three types: the type `x` of the variable expression, the type `a` of the value of the variable, and the type `c` of the context. In general, the type-checker cannot immediately infer the value of `a`, but it does assert that `a` take its correct place within the context and gives `get_var`, the run-time value-lookup function, the correct return type.

Note that the functional dependency of `GetVar` indicates that the context is a function of the variable name and its type. This dependency follows directly from our understanding of how the type checker processes a variable expression, though it is not what we naïvely might guess.

3.2 Typing an application

The `Zip` class and its instances type an application.

```
class Zip c1 c2 c3 | c1 c2 → c3 where
   split :: c3 → (c1,c2)
```

The typing rule for an application has three contexts. The `Zip` class represents the relationship among these contexts: `c1` is the context for the function, `c2` is the context for its argument, and `c3` is the context of the completed application. In the regular simply-typed lambda-calculus, the context `c3` would be equal to `c1` and `c2`. However, in our formulation, the contexts are built first from the variables and may not necessarily be the same when the type checker addresses an application. We can also see at this point that the final context `c3` is a function of the two component contexts `c1` and `c2`. This relationship is expressed in the functional dependency.

To unify the two contexts, the `Zip` instances will assert that they match on the part they have in common, allowing the contexts `c1` and `c2` to be of differing size. Furthermore, any variable flagged as `Used` in `c1` or `c2` must be flagged as such in `c3`. To get the type checker to unify the types of the variables in the context, we could be tempted to write

```
-- broken instance
instance (Zip c1 c2 c3) ⇒
   Zip (Free a, c1) (Free a, c2) (Free a, c3)
   where ...
```

Unfortunately, the type checker will fail to solve a constraint of the form

```
Zip (Free a, ()) (Free b, ()) (Free c, ())
```

if the types `a`, `b` and `c` are distinct. We somehow need to add this as an instance and enforce the fact that these types should be equal. This is possible thanks to the constraint $\sim$.

```
instance Zip () c c where
   split c = ((),c)
instance Zip c () c where
   split c = (c,())
instance Zip () () () where
   split () = ((),())
instance (Zip c1 c2 c3, c ~ a, c ~ b) ⇒
   Zip (Free a, c1) (Free b, c2) (Free c, c3) where
     split (Free c,c3) =
       let (c1,c2) = split c3 in
       ((Free c,c1),(Free c, c2))
instance (Zip c1 c2 c3, c ~ a, c ~ b) ⇒
   Zip (Free a, c1) (Used b, c2) (Used c, c3) where
     split (Used c,c3) =
       let (c1,c2) = split c3 in
       ((Free c,c1),(Used c,c2))
instance (Zip c1 c2 c3, c ~ a, c ~ b) ⇒
   Zip (Used a, c1) (Free b, c2) (Used c, c3) where
     split (Used c,c3) =
       let (c1,c2) = split c3 in
       ((Used c,c1),(Free c,c2))
instance (Zip c1 c2 c3, c ~ a, c ~ b) ⇒
   Zip (Used a, c1) (Used b, c2) (Used c, c3) where
     split (Used c,c3) =
       let (c1,c2) = split c3 in
       ((Used c,c1),(Used c,c2))
```

When typing an application, the function must indeed be typed as an arrow and the argument must be of an appropriate type. These constraints are enforced by the type of the `appl` function, `j c1 (a → b) → j c2 a → j c3 b`. As the Haskell type checker is unifying the variable `a` in this definition, it assigns the correct arrow type to the function being applied.

Lastly, we must consider how to perform the application at run-time. Recalling that Haskell evaluates each term into a continuation wrapped with the data constructor `J`, an application must force evaluation of its function and then apply that to the evaluated form of its argument. To do this, the application context `c3` splits into its initial contexts `c1` and `c2`, for that is what the sub-terms expect. The `split` function does exactly this.

3.3 Typing a lambda-abstraction

The `AddVar` class and its instances type a lambda-abstraction.

```
class AddVar x a c2 c1 | x a c1 → c2 where
   add :: x → a → c2 → c1

instance AddVar n a () () where
   add n a () = ()
instance a1 ~ a2 ⇒
   AddVar Z a1 (Free b,c) (Free a2,c) where
     add Z a (Free b,c) = (Free a, c)
instance a1 ~ a2 ⇒
   AddVar Z a1 (Free b,c) (Used a2,c) where
     add Z a (Free b,c) = (Used a, c)
```

```
instance (AddVar n a c d) ⇒
   AddVar (S n) a (b,c) (b,d) where
     add (S n) a (b,c) = (b,add n a c)
```

The constraints on a lambda-abstraction are encoded in the class `AddVar` and in the `lamb` function whose type is `x → j c1 b → j c2 (a → b)`. The `AddVar` class enforces a constraint on a variable type `x`, an argument type `a`, a result context `c2`, and an internal context `c1`. Note that, in a typical formal setting, `c1` = (`c2`, `x : a`). Since the contexts are built up from the variables, the type checker will have inferred the internal context `c1` before inferring the outer context. This leads to the functional dependency stating that `c2` is a function of `x`, `a`, and `c1`.

The contexts must remain the same length. This is necessary to keep the lower-valued variables matched with their correct types. Thus, in the second and third instances listed, the outer context starts with `Free b`, even though the type `b` is not needed to encode the lambda-abstraction. The variable is marked `Free` because it cannot have been used in this context: even if the variable is bound by an enclosing lambda-abstraction, it is shadowed by the inner lambda-abstraction and is unused, as of yet.

A further constraint on the types is made by the type of the `lamb` function requiring, in particular, that the return type of the lambda-abstraction matches the type of its term.

At runtime, the implementation of `lamb` produces a continuation that returns a function. When this function is given a value `a`, it inserts `a` at the appropriate place in its context (using the `add` function in the `AddVar` class). It then evaluates `f`, the continuation formed when evaluating the term inside the lambda-abstraction.

3.4 Examples

The identity written in this embedded language can be written as

```
*> :t lamb Z (var Z)
lamb Z (var Z) :: (Judg j) ⇒
                  j (Free b, ()) (a → a)
*> :t lamb (S Z) (var (S Z))
lamb (S Z) (var (S Z))
  :: (Judg j) ⇒
     j (Free u, (Free b, ())) (a → a)
```

In both cases, we would like to be able to get back the function `(a → a)`. This amounts to feeding the context with a dummy entry since none of the variables are being used. Since the context is of variable size (depending on which variables were used in the term), it is parametrized with a class `Eval`. The type in the context is forced to be `Dummy` using the constraint $\sim$.

```
data Dummy = Dummy

class Eval c a where
   eval :: (J c a) → a

instance Eval () a where
   eval (J f) = f ()
instance (Eval c a, (Free Dummy) ~ b) ⇒
   Eval (b,c) a where
     eval (J f) =
       eval (J $ λc → f (Free Dummy, c))
```

It behaves as expected.

```
*> :t eval (lamb Z (var Z))
eval (lamb Z (var Z)) :: a → a
*> eval (lamb Z (var Z)) 0
0
*> :t eval (lamb (S Z) (var (S Z)))
eval (lamb (S Z) (var (S Z))) :: a → a
*> eval (lamb (S Z) (var (S Z))) 42
42
```

4. Implementing the subtyping relation

We now turn to the question of encoding the subtyping relation of Table 1 in Haskell.

Since we still want to represent a type in our calculus with a Haskell type, the type system is augmented with an additional construct for the "!" operator. The type constructor `Flag` encapsulates flags on types. The flags are written `Dup` (for "duplicable") in place of "!$^{\texttt{True}}$" and `Sim` (for "simple") in place of "!$^{\texttt{False}}$". Since the flags do not carry any computational meaning, `Dup` and `Sim` are empty types.

```
data Flag f a = Flag a
data Dup
data Sim
```

The type for quantum bits has a dummy implementation.

```
data Qbit = Qbit Bool
```

The arrow type is still the Haskell arrow. For example, the type `!(Bool → Qbit)` is encoded with

```
Flag Dup ((Flag Sim Bool) → (Flag Sim Qbit))
```

4.1 Subtyping relation

Consider the typing derivation of term variables in Table 1: there is a subtyping constraint on types appearing on the left and on the right of the turnstile. A Haskell type class enforces this relation, so that the typing rule

$$\frac{\texttt{a} <: \texttt{b}}{x : \texttt{a} \vdash x : \texttt{b}}$$

yields the type

```
TestSubType a b ⇒ j (a, ()) b
```

instead of the first projection

```
j (a, ()) a
```

as it was for the judgment $x : \texttt{a} \vdash x : \texttt{a}$.

We then need to be able to state whether a type is a subtype of another one with the type class `TestSubType`. Due to the nature of the subtyping relation, if A is a subtype of B then both types have the same internal structure; the only parts that change are the flags. In particular, the proposition `TestSubType a b` implies that `a` and `b` can safely be extended using a unification algorithm. A bidirectional functional dependency enables the use of Haskell's unification in both directions.

```
class TestSubType a b | a → b, b → a where
   subType :: a → b

instance TestSubType Bool Bool where subType x = x
instance TestSubType Qbit Qbit where subType x = x

instance (TestSubType a b, FlagSubType f1 f2) ⇒
  TestSubType (Flag f1 a) (Flag f2 b) where
    subType (Flag x) = Flag (subType x)

instance (TestSubType c a, TestSubType b d) ⇒
  TestSubType (a → b) (c → d) where
    subType f x = subType (f (subType x))
```

Suppose the constraint `TestSubType a b` for some open types `a` and `b`. The double functional dependency calls the unification algorithm of Haskell so that `a` and `b` have the same skeleton. It also propagates the subtyping constraints to flags recorded in an auxiliary type class `FlagSubType`.

Here is an example of how the Haskell type checker builds the set of constraints when asked to solve the relation

```
TestSubType (Flag f1 (a → b)) (Flag g1 d)
```

The list of constraints, unwound one by one, grows as follows:

```
FlagSubType f1 g1,
TestSubType (a → b) d.
```

Then, unification on `d` is done to yield

```
d = (a1 → b1)
FlagSubType f1 g1,
TestSubType (a → b) (a1 → b1),
```

then

```
d = (a1 → b1)
FlagSubType f1 g1,
TestSubType a1 a,
TestSubType b b1.
```

which is what we could have inferred by hand from the subtyping rules.

4.2 Constraints on flags

Let us now define the type class `FlagSubType`. `Dup` is a subtype of anything and `Sim` is above any other flag.

```
class FlagSubType a b

instance FlagSubType Dup a
instance FlagSubType a Sim
```

Then `FlagSubtype a Dup` should imply that `a` is precisely `Dup`. Similarly, `FlagSubType Sim a` implies that `a` is equal to `Sim`. Following the identification of relations on types and type classes, we use a type constraint $\sim$ and write

```
instance (Dup ~ a) ⇒ FlagSubType a Dup
instance (Sim ~ a) ⇒ FlagSubType Sim a
```

The constraint `a ~ b` forces the Haskell type checker into identifying the structure of `b` with the structure of `a`. In particular, in the case `Sim ~ a` it will replace `a` with `Sim`, the expected behavior.

These various instances allow the type checker to correctly infer the flags "!" appearing in types. In all non-ambiguous cases, the type checker can, if necessary, set the corresponding flag variables to the values they should have. For example, if $A <: !B$, for sure A is duplicable. This will be enforced by the third instance of `TestSubType` and by the instance of `FlagSubType` corresponding to the case `a <: Dup`.

Although it seems that we are now done, we are not quite there yet as there are overlapping instances for the type class `FlagSubType`. Note that although this is a potential problem in general as it might yield inconsistent operational behavior of Haskell programs, in our case, this is not an issue since the computational content of the type class is completely empty. It is possible to close the overlapping cases as follows.

```
-- close overlapping cases
instance FlagSubType Dup Sim
instance FlagSubType Dup Dup
instance FlagSubType Sim Sim
instance (Dup ~ Sim) ⇒ FlagSubType Sim Dup
```

The last instance has a blatantly false constraint; this is on purpose: the flag `Sim` should not be a subtype of `Dup`. This trick is already known [5] and makes use of Haskell's type error to get meaningful information on the cause of the error. For example, the following code

```
*Main> :t undefined :: FlagSubType Sim Dup ⇒ a
```

produces the error

```
<interactive>:1:1:
 Couldn't match type 'Dup' with 'Sim'
 In the expression:
          undefined :: FlagSubType Sim Dup ⇒ a
```

This can be used to raise an error when some linear data is being duplicated, and provide a meaningful error message. Sections 5 and 7 show uses of this trick.

Note that compiling requires two potentially dangerous extensions of GHC:

```
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE IncoherentInstances #-}
```

These extensions can cause computational inconsistencies. In our case, they are needed because of the overlapping instances of `FlagSubType`. Since this class has no computational content, these extensions are not problematic here.

The classes `TestSubType`, `FlagSubType` and their instances deserve particular attention, as they are at the core of the construction presented in this paper: they are really the one fully piggy-backing on Haskell's type inference algorithm, and they can be said to be among this paper's main contributions.

5. Adding linearity constraints and subtyping

This section incorporates the type system and subtyping relation described in Section 4 to the lambda-calculus of Section 3.

The class of judgments. The class `Judg` becomes

```
class Judg j where
  var :: (GetVar x a c) ⇒ x → j () c a
  appl :: Zip c1 c2 c3 ⇒
    j w1 c1 (Flag f (a → b)) → j w2 c2 a →
    j (w1,w2,a,f) c3 b
  lamb :: (AddVar x a c2 c1, TestCont c2 f1,
           FlagSubType f1 f) ⇒
    x → j w c1 b → j (w,f,f1) c2 (Flag f (a → b))
```

The type constructor `j` now features a new variable as its first argument. This change deserves some explanation.

The problem with the new type system is the presence of open types for flags, together with constraints on them. In many cases, these open flags do not cause any problem. But in some situations, they do, as depicted with the term

```
let x = Z in
let y = S Z in
let z = S (S Z) in
  (lamb z (appl (lamb x (var z)) (lamb y (var y))))
```

of type `TestSubType u b ⇒Flag f (u → b)`.

The type inference algorithm creates a type with open flags for `lamb y (var y)`, together with a subtyping constraint on a type that will then be completely dropped from the final type. However, the constraints on these forgotten types will remain, creating the type-checking error

```
<interactive>:1:46:
  Ambiguous type variables 'a', 'b0'
```

```
    in the constraint:
    (TestSubType a b0) arising from a use of 'var'
    ...
```

An easy solution is to keep these types around, in order for Haskell's type checker not to complain. However, we only need to keep a *trace* of them: they can be stored as phantom types; this is the purpose of this new first argument of the judgment `j`: store the constrained types that might get erased. A canonical instance of the typeclass `Judg` is

```
data J w c a = J (c → a)
```

The other type classes. Let's go through the other type classes. First, `GetVar` is almost the same as the one used in Section 3.1. The difference lies in the first instance that now reads

```
instance TestSubType a b ⇒ GetVar Z b (Used a,())
  where get_var (Used a,c) Z = subType a
```

The type in the context can be a subtype of the type of the variable. This is enforced by the constraint `TestSubType a b`.

The type class `Zip` now has one more constraint in the last instance, reading as follows:

```
instance (Zip c1 c2 c3, (Flag Dup d) ~ c,
          c ~ a, c ~ b) ⇒
  Zip (Used a, c1) (Used b, c2) (Used c, c3)
  where ...
```

This constraint says that a variable can be used in both branches only when it is duplicable.

For constraints used for the creation of the lambda-abstraction, the class `AddVar` is unchanged. However, there is a new type class `TestCont`. It asserts that the free variables of the body (not including the argument) of a duplicable lambda-abstraction are themselves duplicable. In other words, there cannot be a duplicable closure over non-duplicable variables. This assertion can be decomposed into two propositions:

1. If any of the variables of the context used in the term is non-duplicable, the lambda-abstraction is not duplicable either. This makes a function from the context to the type of the lambda-abstraction.
2. If the lambda-abstraction is duplicable, then all the used variables in the context are also duplicable. That makes a function from the type of the lambda-abstraction to its context.

Clearly, these two propositions say the same thing. However, from a logic programming point of view, they do not behave the same way: the first one generates the flag for the lambda-abstraction's type from the context, whereas the second one generates the flags of the types in the context from the lambda-abstraction. These two propositions are encoded in two type-classes `TestCont1` and `TestCont2`; the type class `TestCont` is their conjunction.

```
class TestCont1 cont d | cont → d
instance TestCont1 () Dup
instance (TestCont1 c f) ⇒
    TestCont1 (Used (Flag Dup a), c) f
instance TestCont1 (Used (Flag Sim a), c) Sim
instance (TestCont1 c f) ⇒
    TestCont1 (Free a, c) f

class TestCont2 c f
instance TestCont2 () Dup
instance (TestCont2 c Dup, (Flag Dup a) ~ b) ⇒
    TestCont2 (Used b, c) Dup
instance (TestCont2 c Dup) ⇒
    TestCont2 (Free b, c) Dup
instance TestCont2 c Sim

class TestCont c d
instance (TestCont1 c d, TestCont2 c d) ⇒
    TestCont c d
```

Now, we can try to type some expressions with Haskell's interpreter. Let us define the following terms.

```
*Main> let x = Z
*Main> let y = S Z
*Main> let non_dup_x = (var x) ::
    J () (Used (Flag Sim Bool), ()) (Flag Sim Bool)
*Main> let const_fun = lamb y non_dup_x
*Main> let two = lamb y (lamb x (appl (var y)
                          (appl (var y) (var x))))
```

The term `two` is the Church numeral $\bar{2}$. The term `non_dup_x` is `x` made non-duplicable, and `const_fun` is a constant function returning this non-duplicable variable `x`.

If we furthermore write in the Haskell interpreter the command

```
*Main> :t lamb x (appl two const_fun)
```

the interpreter answers

```
 Couldn't match type 'Sim' with 'Dup'
```

since this code tries to duplicate the variable `var x` of type `Flag Sim Bool` hidden in `const_fun`, that is, an non-duplicable object. If we define instead

```
*Main> let dup_x = (var x) ::
   J () (Used (Flag Dup Bool), ()) (Flag Dup Bool)
*Main> let const_fun = lamb y dup_x
```

with `Dup` instead of `Sim` and try again

```
*Main> :t lamb x (appl two const_fun)
```

we get the type of the term, with a dangling subtyping constraint:

```
(lamb x (appl two const_fun))
  :: FlagSubType f2 f1 ⇒
     J (...)                      -- phantom type
       (Free b, (Free b1, ()))    -- typing context
       (Flag f (Flag Dup Bool →
                Flag f3 (Flag Dup Bool →
                         Flag f2 Bool)))
```

Modifying the class `Eval`**.** The class `Eval` also needs to be changed. Its goal is to take a closed term and strip the empty context. In the new situation, it is still desirable to be able to do this. However, the trace of the derivation in the first parameter of the judgment type cannot be removed: the class is written with an auxiliary type `E` as follows.

```
data E w a = E a

class Eval c a where
   eval :: (J w c a) → E w a

instance Eval () a where
   eval (J f) = E (f ())
```

The last instance is not modified:

```
instance (Eval c a, (Free Dummy) ~ b) ⇒
       Eval (b,c) a where
   eval (J f) = eval (J (λc → f (Free Dummy, c)))
```

Instead of returning a bare type, the function `eval` now returns a type enclosed in a construct that keeps track of the dangling constraints.

One slight inconvenience of this new version is that it is not easy to directly enforce an embedded term to have a particular type by using ascription. Before, the following worked:

```
:t (eval (lamb Z (var Z))) ::
        Flag Dup (Flag Dup Bool → Flag Dup Bool)
```

Now, the trace has to be added:

```
:t eval (lamb Z (var Z)) ::
     E ((), Dup, Dup, Flag Dup Bool)
       (Flag Dup (Flag Dup Bool → Flag Dup Bool))
```

There is a trick to avoid the need to produce the trace: the function `setType`:

```
setType :: E w a → a → E w a
setType x y = x
```

We can now write

```
*Main> let u = undefined
*Main> :t setType (eval (lamb Z (var Z)))
    (u :: Flag Dup (Flag Dup Bool → Flag Dup Bool))
```

if we wish to enforce a type. And of course, it still fails in case of non-existing type derivation:

```
*Main> let u = undefined
*Main> :t setType (eval (lamb Z (var Z)))
    (u :: Flag Dup (Flag Sim Bool → Flag Dup Bool))

<interactive>:1:16:
    Couldn't match type 'Sim' with 'Dup'
    ...
```

A similar trick can be applied to be able to easily enforce a particular typing derivation, without to have to come up with the trace:

```
setJudg :: J w c a → c → a → J w c a
setJudg x y z = x
```

6. Extending to the full quantum lambda-calculus

Section 5 includes most of the difficulties for encoding the full quantum lambda-calculus. What is missing from the class `Judg` is the three last lines of Table 1.

Classical and quantum Booleans. Let us first concentrate on the two last ones, concerned with classical Boolean and quantum Booleans. The class is extended as follows. For legibility, we omit the trace of existing types: as in Section 5, its only purpose is to please Haskell's checking of constraint ambiguity.

```
class Judg j where
 ...
 tt :: j _ () (Flag f Bool)
 ff :: j _ () (Flag f Bool)
 ifx :: (Match c21 c22 c2, Zip c1 c2 c3) ⇒
        j _ c1 (Flag d Bool) →
        j _ c21 a → j _ c22 a → j _ c3 a
 new :: j _ c (Flag f Bool) → j _ c (Flag Sim Qbit)
 meas :: j _ c (Flag f Qbit) → j _ c (Flag f' Bool)
 had :: j _ c (Flag f Qbit) →
        j _ c (Flag Sim Qbit)
 cnot :: j _ c (Flag f1 (Flag f2 Qbit,
                Flag f3 Qbit)) →
         j _ c (Flag Sim (Flag Sim Qbit,
                          Flag Sim Qbit))
```

The class `Match` is a variant of the class `Zip`. It encodes the assertion that t and t' in the typing judgment for *if* are evaluated in the *same* context Γ_2.

```
class Match c1 c2 c3 | c1 c2 → c3 where
   match :: c3 → (c1,c2)
instance Match () c c where
   match c = ((),c)
instance Match c () c where
   match c = (c,())
instance Match () () () where
   match () = ((),())

instance (Match c1 c2 c3, c ~ a, c ~ b) ⇒
   Match (Free a, c1) (Free b, c2) (Free c, c3)
   where
   match (Free c,c3) =
     let (c1,c2) = match c3 in
     ((Free c,c1),(Free c,c2))
instance (Match c1 c2 c3, c ~ a, c ~ b) ⇒
   Match (Used a, c1) (Used b, c2) (Used c, c3)
   where
   match (Used c,c3) =
     let (c1,c2) = match c3 in
     ((Used c,c1),(Used c,c2))
```

This class ensures that the context `c1` and `c2` match on their common subset and that the context `c3` is larger than the others.

Note that in the class `Judg`, a quantum bit can only be created with a flag set to `Sim`. This enforces the non-duplication of quantum bits.

Tensors. The typing rules for tensors are more complicated. They involve a special operation `∨` on flags, encoded in the class `Or`. Just as for the class `TestCont`, there are two components. `Or1` declares that the result flag depends on the input flags, and `Or2` declares that the input flags depend on the result flag. If the result is `Sim`, there is no choice for the inputs `e` and `a`. If it is `Dup`, nothing further can be deduced.

```
class Or1 e a x | e a → x
instance Or1 e Dup Dup
instance Or1 Dup e Dup
instance Or1 Sim Sim Sim
instance (a ~ b) ⇒ Or1 Sim a b
instance (a ~ b) ⇒ Or1 a Sim b

class Or2 e a x | x → e a
instance Or2 Sim Sim Sim
instance Or2 e a Dup

class Or e a x
instance (Or1 e a x, Or2 e a x) ⇒ Or e a x
```

With this class, the creation of a pair in the class `Judg` is represented by

```
class Judg j where
  ...
  tens :: (Zip c1 c2 c3, Or e xa x, Or e yb y) ⇒
    j _ c1 (Flag x a) → j c2 (Flag y b) →
    j _ c3 (Flag e (Flag xa a, Flag yb b))
```

Again, we omit the trace of existing types in the first parameter of `j`. For the last term construct, a new class exposes the types `x`, `a`, `y` and `b`: the type checker needs these clues when making of an instance of this class.

```
class JudgTens j x a y b where
  lambtens ::
      (TestCont c3 f1, FlagSubType f1 f,
       Add2Var n (Flag x a) m (Flag y b) c3 c1,
       Or e xa x, Or e yb y) ⇒
    n → m → j _ c1 d →
    j _ c3 (Flag f ((Flag e (Flag xa a,
                             Flag yb b)) → d))
```

The construct `lettens` and `letx` are defined as follows.

```
lettens x y m n = appl (lambtens x y n) m
letx x m n = appl (lamb x n) m
```

The class `Add2Var` behaves in a manner similar to `AddVar`, except that it works with two variables. An auxiliary class `IsNotEq` is needed in the definition of `Add2Var`, failing when two variable names that should be distinct are not.

```
class IsNotEq a b
instance IsNotEq Z (S n)
instance IsNotEq (S n) Z
instance IsNotEq m n ⇒ IsNotEq (S m) (S n)

class Add2Var x a y b c2 c1 | x a y b c1 → c2 where
   add2 :: x → a → y → b → c2 → c1
instance (IsNotEq m n) ⇒ Add2Var m a n b () ()
instance (IsNotEq Z Z) ⇒ Add2Var Z a Z b c c
instance (a1 ~ a2, AddVar m b c2 c1) ⇒
   Add2Var Z a1 (S m) b (Free y, c2) (Free a2, c1)
instance (a1 ~ a2, AddVar m b c2 c1) ⇒
   Add2Var Z a1 (S m) b (Free y, c2) (Used a2, c1)
instance (a1 ~ a2, AddVar m b c2 c1) ⇒
   Add2Var (S m) b Z a1 (Free y, c2) (Free a2, c1)
instance (a1 ~ a2, AddVar m b c2 c1) ⇒
   Add2Var (S m) b Z a1 (Free y, c2) (Used a2, c1)
instance (Add2Var n a m b c d) ⇒
   Add2Var (S n) a (S m) b (y,c) (y,d)
```

The first instance of `Add2Var` always succeeds, as long as the names are distinct. This is similar to what happens in the creation of a lambda-abstraction. The second instance always fails since the names are both equal to `Z`, and the same name cannot be used twice for the two variables in the *let* construct. The other instances encode induction steps.

7. Examples of terms

We now examine some examples of terms. Due to the fact that QLC has a native non-duplicable object, it is possible to write examples of computations where the non-duplicability might be an issue.

This section does not require particular knowledge of quantum computation apart from the understanding that a quantum bit is non-duplicable.

7.1 Non-duplicability

Using quantum computation, one can build a perfect coin toss using the following code.

```
*Main> :t eval (meas (had (new ff)))
eval (meas (had (new ff)))
  :: E ... (Flag f' Bool)
```

Although it internally creates a qubit, this term is duplicable since it is ultimately a Boolean. And indeed:

```
*Main> :t setType (eval (meas (had (new ff))))
                  (undefined :: Flag Dup Bool)
```

succeeds. This is correct behavior, because the code is call-by-value: it consumes the qubit as well in the *meas* construct. The following code also type checks: duplicating the result of the coin toss is valid.

```
setType (eval (appl (lamb Z (tens (var Z) (var Z)))
                    (meas (had (new ff)))))
        (undefined ::
          Flag Dup (Flag Dup Bool, Flag Dup Bool))
```

7.2 Reporting errors

If instead of passing the result of the coin toss, we try to duplicate the the qubit before measuring it, the term does not type check.

```
*> :t appl (lamb Z (tens (meas (var Z))
                         (meas (var Z))))
           (had (new ff))

<interactive>:1:7:
   Couldn't match type 'Sim' with 'Dup'
   In the first argument of 'appl', namely
     '(lamb Z (tens (meas (var Z)) (meas (var Z))))'
   In the expression:
     appl (lamb Z (tens (meas (var Z))
                  (meas (var Z)))) (had (new ff))
```

Note that this is a meaningful message indicating the place were the type checker stopped. The error message says that a `Sim` type and a `Dup` type did not unify. This is one of the benefits of having the Haskell type checker do the job for us.

7.3 Non-duplicable functions

Quantum bits are not the only terms to be non-duplicable. In the following example, the function is non-duplicable as it contains the quantum bit that was fed to the outer lambda-abstraction.

```
*> let f = appl (lamb Z (lamb (S Z)
             (meas (var Z)))) (new ff)
*> :t f
... :: Judg j ⇒
     j ... (Free b, ())
         (Flag Sim (a → Flag f' Bool))
```

Trying to duplicate this term fails, as before.

```
*> :t appl (lamb Z (tens (var Z) (var Z))) f

<interactive>:1:7:
    Couldn't match type 'Dup' with 'Sim'
    ...
```

7.4 A non-duplicable pair of functions

As a larger example to show the robustness of the implementation, we encode the teleportation algorithm presented in Section 2. It shows that the Haskell compiler can infer the type of a non-trivial term.

For a complete correlation with the code Selinger and Valiron presented, assume that we add the maps

```
u1 :: j _ c (Flag f Qbit) → j _ c (Flag Sim Qbit)
u2 :: j _ c (Flag f Qbit) → j _ c (Flag Sim Qbit)
u3 :: j _ c (Flag f Qbit) → j _ c (Flag Sim Qbit)
u4 :: j _ c (Flag f Qbit) → j _ c (Flag Sim Qbit)
```

to the class `Judg`. These extra operations do not affect any substantive quality of the language. They are included here merely to match up with the example.

The term `telep` is constructed as follows.

```
x = Z
y = S Z
z = S (S Z)

epr = lamb x (cnot (tens (had (new ff)) (new ff)))

bellmeas =
    lamb x (
    lamb y (
    lettens z t
      (cnot (tens (var x)
                  (var y)))
      (tens (meas (had (var z)))
            (meas (var t)))))

u =
    lamb z (
    lambtens x y (
      ifx (var x)
          (ifx (var y) (u1 (var z)) (u2 (var z)))
          (ifx (var y) (u3 (var z)) (u4 (var z)))))

telep =
  lettens x y (appl epr ff) (
  letx z (appl bellmeas (var x)) (
  letx t (appl u (var y)) (
  (lamb x (appl (var t) (appl (var z) (var x)))))))
```

The type of this term is supposed to be $(qbit \to qbit)$, and cannot be made duplicable. The Haskell interpreter types the term with the type described in Section 2.2.

```
*> :t eval telep
eval telep :: (...) ⇒ E ...
  (Flag Sim
   (Flag Sim (Flag f7 Qbit → Flag Sim Qbit)))
```

It is non-duplicable, as the following attempt fails:

```
*> :t appl (lamb x (tens (var x) (var x))) telep

<interactive>:1:7:
    Couldn't match type 'Sim' with 'Dup'
    ...
```

The type of the other terms are as described in Section 2.2:

```
*> :t eval u
eval u
:: ... ⇒ E ...
   (Flag f3 (Flag f Qbit →
     Flag f5 (Flag e
       (Flag xa Bool, Flag yb Bool) →
        Flag Sim Qbit)))
*> :t eval bellmeas
eval bellmeas
:: ... ⇒ E ...
   (Flag f2 (Flag f1 Qbit →
              Flag f4 (Flag f Qbit →
      Flag e (Flag xa Bool, Flag yb Bool))))
*> :t eval epr
eval epr
:: E ...
     (Flag f (a →
      Flag Sim (Flag Sim Qbit, Flag Sim Qbit)))
```

8. Operational description

The proposed techniques for embedding linearity constraints are compatible with a monadic approach. This approach is given below for QLC as evidence that it can be used in a practical context for experimenting with the language.

For the language QLC, this section formalizes the QRAM model: This can easily be interpreted as a state monad.

```
type Qram :: *
data Prob a = ...
data PQM a = PQM (Qram → Prob (Qram, a))
instance Monad Prob where ...
instance Monad PQM where ...
```

The monad `PQM` is also an instance of the classes

```
data Qbit = ...
class QC m where
   qc_new :: Bool → m Qbit
   qc_had :: m Qbit → m Qbit
   qc_CNOT :: m (Qbit, Qbit) → m (Qbit, Qbit)
   qc_meas :: m Qbit → m Bool

class (Monad m) ⇒ StrongMonad m where
    strength :: m a → m b → m (a,b)
```

The class `QC` gives access to the low-level operation of the QRAM device. The class `StrongMonad` states that the monad `m` is strong.

A slight modification of the classes `Judg`, `JudgTens` and `Eval` needs to be done for that goal. Here is a fragment of this work; the complete development is available on the web [11].

```
class Judg j m where
  lamb :: (AddVar x a c2 c1, TestCont c2 f1,
           FlagSubType f1 f) ⇒
      x → j m _ c1 (m (Flag f' b)) →
      j m _ c2 (m (Flag f (a →  (m (Flag f' b)))))
  appl :: Zip c1 c2 c3 ⇒
      j m _ c1 (m (Flag f ((Flag f'' a) →
                           m (Flag f' b)))) →
      j m _ c2 (m (Flag f'' a)) →
      j m _ c3 (m (Flag f' b))
  var :: (GetVar m x a c) ⇒ x → j m _ c (m a)
  ...
```

The data type `J` changes as follows.

```
data J m w c a = J (c → m a)
```

It is an instance of `Judg`, parametrized by `m`.

```
instance (Monad m, StrongMonad m, QC m) ⇒
    Judg J m where ...
```

Two implementations of this quantum state monad are on the web [11].

9. Discussion and related work

9.1 Scalability of the approach

We believe that the technique devised here can be applied to programming languages other than QLC. We found that the encoding of the type system in a logic programming manner was relatively fast, and the benefit—having Haskell at hand to develop an operational semantics—noticeable.

The main drawback of this approach is the need for keeping a trace of the typing derivation of the term. Since it is kept within the type, it potentially makes a large overhead for the type checker. The `telep` algorithm, for instance, has 114 elements in its trace, making the actual size of the type gigantic. This hampers the scalability

of the approach for uses other than small-scale experimentations with the embedded language.

However, it should be noted that the reason the trace is needed—the ambiguity check—could be eliminated with an extension that either instantiates "orphan" constraints to defaults values, or forgets them altogether. For QLC's type system, the orphan constraints deal only with flags that happen to be type variables: setting them all to `Sim` would do the trick.

9.2 Linearity constraints and type classes

Kiselyov [4] provides an alternative way of enforcing linearity constraints on a lambda-calculus with de Bruijn indices, using Haskell's type classes. His approach is based on a more algorithmic presentation of the typing rules in which the interpretation of a term is a function from input contexts to output contexts. He considers adding duplicable variables, but does not implement a subtyping relation as we do here. We have found our embedding to be a more natural representation of the typing rules that is more convenient to work with.

9.3 GADTS and type families

We believe that our approach cannot easily be transposed to GADT's and type families: the implementation we propose uses Haskell's unification algorithm. Using type families would require reimplementing this unification algorithm, at the type level.

9.4 Quantum computation

Altenkirch and Grattage design QML, a language specific for a quantum computation with quantum control [1]. In that paper,[1] Haskell is only used as the language for writing the QML compiler. However, the quantum IO monad can probably be used as a candidate for the quantum state monad described in this paper.

10. Conclusion

This paper provides a concrete case of use for the logic programming features of the Haskell type system. We make the type system check non-trivial linearity constraints while keeping a natural interpretation of terms as regular Haskell code. Because the constraints on types are checked on the fly, the compiler also provides meaningful error messages.

We have found a concrete use of functional dependencies for logic programming where we believe type families cannot be used.

Acknowledgments

We'd like to thank the UPenn PL-Club and Stephanie Weirich for their useful feedback. Supported by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior National Business Center contract number D11PC20168. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoI/NBC, or the U.S. Government.

References

[1] T. Altenkirch and J. Grattage. A functional quantum programming language. In *Proceedings of the 20th Symposium on Logic in Computer Science, LICS'05*, pages 249–258, 2005.

[2] S. J. Gay. Quantum programming languages: survey and bibliography. *Mathematical Structures in Computer Science*, 16(4):581–600, 2006.

[3] A. S. Green and T. Altenkirch. Shor in haskell: The quantum io monad. In S. Gay and I. Mackie, editors, *Semantic Techniques in Quantum Computation*, chapter 1. CUP, 2009.

[4] O. Kiselyov. Linear and affine lambda-calculi. Lecture notes, accessible on `http://okmij.org/ftp/tagless-final/course/course.html#linear`, 2010.

[5] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell 2004: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004. ISBN 1-58113-850-4. doi: http://doi.acm.org/10.1145/1017472.1017488.

[6] E. H. Knill. Conventions for quantum pseudocode. Technical report, Los Alamos National Laboratory, 1996.

[7] C. McBride. Faking it: Simulating dependent types in haskell. *J. Funct. Program.*, 12(5):375–392, 2002. ISSN 0956-7968.

[8] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2002.

[9] C. Parker. Type-level instant insanity. *The Monad.Reader*, 8, September 2007.

[10] P. Selinger and B. Valiron. A lambda calculus for quantum computation with classical control. *Mathematical Structures in Computer Science*, 16:527–552, 2006.

[11] B. Valiron. `http://www.monoidal.net/papers/qhaskell`.

[12] J. K. Vizzotto, A. C. da Rocha Costa, and A. Sabry. Quantum arrows in haskell. In P. Selinger, editor, *Proceedings of the Fourth International Workshop on Quantum Programming Languages*, volume 210 of *Electronic Notes in Theoretical Computer Science*, Oxford, UK., 2006.

A. Extensions of GHC

The code requires a few extensions to compile with GHC 7.4.1.

- The first ones are there to allow more expressive type classes, but are not known to cause major operational issues.

```
{-# LANGUAGE GADTs #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE NoMonomorphismRestriction #-}
```

- Finally, we use these extensions already discussed in Section 4.2.

```
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE IncoherentInstances #-}
```

[1] see also `http://sneezy.cs.nott.ac.uk/qml/compiler/`