

Typechecking embedded languages with linearity constraints in Haskell

Benoit Valiron

University of Pennsylvania
valiron@seas.upenn.edu

Steve Zdancewic

University of Pennsylvania
stevez@cis.upenn.edu

Abstract

We aim at implementing the quantum lambda-calculus of Selinger-Valiron as an embedded language in Haskell. The main difficulty is to deal with the linearity constraints of the type system, which take the form of a special flag “!” that can be set on any type to indicate that one can duplicate an object of that type. The type system is equipped with a subtyping relation: one is never forced to duplicate a duplicable element. We show how the Haskell type checker can be programmed to do the type inference for us, using the logic programming feature of type classes. We describe the source language, how it is embedded in Haskell, and we discuss a few examples.

We believe that this approach is novel and could be usefully applied to other embedded languages where constraints on the type system need to be enforced.

1. Introduction

We aim at implementing a lambda-calculus with linearity constraints in Haskell. The objective is to be able to manipulate the quantum lambda-calculus of [6] as an embedded language. We first provide a short introduction to quantum computation, then discuss the programming paradigm we are interested in. Finally, we sketch the embedding in Haskell.

1.1 Quantum computation.

In quantum computation, we deal with classical data but also with quantum data, encoded on the state of particles governed by the laws of quantum mechanics. The state of a quantum particle is a normalized vector in a Hilbert space, that is, a complex vector space with a notion of norm and a notion of orthogonality. In this section, we provide a brief overview of what is quantum computation. For a full overview, see e.g. [5].

Quantum bits. A quantum bit, or qubit, is a normalized vector in the Hilbert space of dimension 2. If one chooses an orthonormal basis $\{|f\rangle, |t\rangle\}$, a qubit is $\alpha|f\rangle + \beta|t\rangle$, a linear combination of f and t : this superposition of data accounts for part of the power of quantum computation.

The state of two quantum bits is the tensor product of the original states. That is, a two-quantum bits system is described by

a normalized vector in the Hilbert space generated by the basis $\{|ff\rangle, |ft\rangle, |tf\rangle, |tt\rangle\}$. A two-quantum bits system is therefore a quantum superposition of pairs of classical bits.

Operations. Beside creating them, one can perform two operations on quantum bits. First, one can apply unitary operations (that is, linear maps sending a normalized basis to a normalized basis). For example, the Hadamard gate is the one-qubit gate sending $|f\rangle$ to $\frac{1}{\sqrt{2}}(|f\rangle + |t\rangle)$ and $|t\rangle$ to $\frac{1}{\sqrt{2}}(|f\rangle - |t\rangle)$. The CNOT gate is a two-qubits gate, inverting the value of the second qubit if the first qubit is t : it sends $|ff\rangle$ and $|ft\rangle$ to themselves, $|tf\rangle$ to $|tt\rangle$ and $|tt\rangle$ to $|tf\rangle$.

The other operation is used for retrieving classical data out of quantum one and is called *measurement*. It is a destructive and probabilistic operation. The measure of the quantum bit $\alpha|f\rangle + \beta|t\rangle$ answers f and projects the qubit on the basis element $|f\rangle$ with probability $|\alpha|^2$, answers t and project the qubit on the basis element $|t\rangle$ with probability $|\beta|^2$. If we were to measure the first quantum bit of the state $\alpha|ff\rangle + \beta|ft\rangle + \gamma|tf\rangle + \delta|tt\rangle$, we would get f with probability $|\alpha|^2 + |\beta|^2$ and the state would be changed to $\alpha|ff\rangle + \beta|ft\rangle$ (up to renormalization), and we would get t with probability $|\gamma|^2 + |\delta|^2$ and the state would be changed to $\gamma|tf\rangle + \delta|tt\rangle$ (again, modulo renormalization).

No-cloning. One constraint of quantum computation is the impossibility of cloning a qubit. There is no physical operation that sends any quantum bits $\alpha|f\rangle + \beta|t\rangle$ to the two-qubits states

$$\begin{aligned} (\alpha|f\rangle + \beta|t\rangle) \otimes (\alpha|f\rangle + \beta|t\rangle) \\ = \alpha^2|ff\rangle + \beta\alpha|tf\rangle + \alpha\beta|ft\rangle + \beta^2|tt\rangle. \end{aligned}$$

The “copying” operation sending linearly $|f\rangle$ to $|ff\rangle$ and $|t\rangle$ to $|tt\rangle$ is well defined but distinct. It sends $\alpha|f\rangle + \beta|t\rangle$ to $\alpha|ff\rangle + \beta|tt\rangle$.

1.2 Programming in a QRAM model

[4] describes a framework for quantum computation with classical control, the quantum random access machine (or QRAM). In this setting, quantum data is thought of as being stored in a quantum device and accessed from a classical computer through a library of functions for creating quantum bits, applying unitary gates or performing measurements. Control, that is, test and loops, is performed on the classical computer on which the quantum device is attached.

A state monad. A QRAM model can easily be interpreted with a combination of a probabilistic monad and a state-monad.

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

[Copyright notice will appear here once ‘preprint’ option is removed.]

```
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)
```

where we hard-coded the Hadamard gate and the CNOT gate for the purpose of the discussion. The class QC gives access to the low-level operation of the QRAM device. The class StrongMonad states that the monad m is strong.

Duplicable and non-duplicable data. Using just the state monad, nothing would prevent us from writing the following code:

```
q :: PQM (Qbit, Qbit)
q = do
  let q' = qc_had (qc_new True)
  strength q' q'
```

Listing 1. Cloning with state monad

What is the meaning of q in this setting? Since we manipulate quantum and classical bits, the natural interpretation is not “copying” but “cloning” (and this is the default behavior of the state monad). Indeed, cloning is the natural counterpart to classical duplication (in a categorical sense).

We therefore need to keep track of what is duplicable and what is not. The next paragraph proposes to do this using a type system.

A quantum lambda-calculus. [6] follows a functional approach to the QRAM model by mean of a typed, call-by-value lambda-calculus. Terms of the language are designed to live in the Kleisli category of the state monad.

We describe a slightly modified version of the language in [6].

$$s, t ::= x \mid \lambda x. s \mid (s)t \mid \langle s, t \rangle \mid \text{let } \langle x, y \rangle = s \text{ in } t \mid tt \mid \text{ff} \mid \text{if } s \text{ then } t \text{ else } t' \mid \text{new}(s) \mid H(s) \mid \text{CNOT}(s) \mid \text{meas}(s)$$

Being linear, there is no native projections for the pair; instead, a *let*-notation is used to extract the components. The language can deal with classical booleans through *tt*, *ff* and the *if-then-else* constructor and with quantum booleans, through the term constructors *new* (qubit creation), *H* (Hadamard gate), *CNOT* (CNOT-gate) and *meas* (measurement).

The type system is

$$A, B ::= \text{bit} \mid \text{qbit} \mid A \rightarrow B \mid A \otimes B \mid !A.$$

It contains a special flag “!” to indicate whether a term is duplicable or not. If the term *s* is of type *!A*, it means that it is of type *A* and that it is duplicable. In particular, there is a canonical subtyping relation $A <: B$ driven by this remark: $!A <: A$. As expected, the type constructor (\rightarrow) is covariant on the right and contravariant on the left. For simplicity we assume that there is either no flag or only one flag: $!!A$ is the same thing as $!A$, making the language slightly different from [6]. The other difference lies in the way quantum operations are treated: here we are using term constructors whereas in [6], term constants were used. However, these are minor modifications that do not change the behavior of the language.

The typing rules for the language are given in Table 1, and they define a variant of the linear lambda calculus with subtyping. In

particular, an exponent *m* on $!^m$ is either True or False, meaning that the type constructor ! is correspondingly either present or elided. There are two typing rules for the lambda-abstraction. There are no constraints for building a non-duplicable lambda-abstraction, but for it to be duplicable we need to make sure that all the parts of the body are themselves duplicable.

1.3 Embedding in Haskell.

[6] provides an operational semantics in the form of an abstract machine. A state of the machine is of the form $[Q, L, s]$, where *Q* is an *n*-qubits state, *s* is a term and *L* is a table of pointers from free variables of type *qbit* to the qubits in the state *Q*. The language is shown to satisfy subject reduction and the type system prevents the quantum bits from being duplicated: the only way to create a quantum bit is by using the term constructor *new*, and the corresponding typing rule enforces the non-duplicability of the resulting qubit.

In this paper, we want to embed this language in Haskell using the state monad PQM so that, in particular, a term $s : \text{bit}$ can be written as a Haskell term $\text{prog} :: \text{PQM Bool}$. We want to make sure that the linearity constraints are enforced, and we shall use the logic programming capabilities of functional dependencies in type classes for that matter.

1.4 Contribution and plan of the paper.

The plan of the paper is as follows. In Section 2, we discuss the encoding of a simply-typed lambda-calculus. For that purpose, we propose a new utilization of type classes to enforce equality of types. In Section 3, we show how to deal with the flag !. This is a novel approach that make the Haskell type unification algorithm decide on subtyping constraints. In Section 4, we extend the language to the full quantum lambda-calculus of Table 1, sketching how generalizing this method to a richer language. Finally, in Section 5, we discuss some examples, an alternative approach using de Bruijn indices, how to use the designed language for quantum computation, and we talk about related works in the literature.

2. A simply-typed lambda-calculus

In this section, we discuss the embedding of the language in Haskell.¹ For simplicity, we first describe a bare-bone simply-typed lambda-calculus. In Section 3, we shall extend the type system with the “!” flag and in Section 4 the missing constructs to get the quantum lambda-calculus.

We consider a lambda-calculus whose variables range over the natural numbers and whose types can be open.

$$s, t ::= n \mid \lambda n. s \mid (s)t, \\ A, B ::= X \mid A \rightarrow B.$$

The Haskell representation of a type is simply a Haskell type. A typing judgment will be typed with $j :: * \rightarrow * \rightarrow *$ taking as first argument a context and as second argument the final type of the term.

Because we write variables as natural numbers, a typing context can be encoded as a tower of pairs $(A_n, (A_{n-1}, \dots, (A_0, ()) \dots))$ where the type A_i is the type of the free variable named *i*. To record which variable is actually used in the list, we define two types Used and Free of sort $* \rightarrow *$.

Typing judgments can be made into a class

```
class MyJudg 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
```

¹ In this paper we use GHC version 7.0.3.

$$\begin{array}{c}
\frac{A < : B}{\Delta, x : A \vdash x : B} \quad \frac{\Delta, x : A \vdash B}{\Delta \vdash \lambda x.s : A \rightarrow B} \quad \frac{! \Delta, \Gamma, x : A \vdash s : B \quad |\Delta| \cup \{x\} = \text{FV}(s)}{! \Delta, \Gamma \vdash \lambda x.s : !(A \rightarrow B)} \quad \frac{! \Delta, \Gamma_1 \vdash s : A \rightarrow B \quad ! \Delta, \Gamma_2 \vdash t : A}{! \Delta, \Gamma_1, \Gamma_2 \vdash (s)t : B} \\
\\
\frac{! \Delta, \Gamma_1 \vdash s : !^{m \vee n} A \quad ! \Delta, \Gamma_2 \vdash t : !^{m \vee o} B}{! \Delta, \Gamma_1, \Gamma_2 \vdash \langle s, t \rangle : !^m (!^n A \otimes !^o B)} \quad \frac{! \Delta, \Gamma_2 \vdash s : !^o (!^m A \otimes !^n B) \quad ! \Delta, \Gamma_1, x : !^{m \vee o} A, y : !^{n \vee o} B \vdash t : C}{! \Delta, \Gamma_1, \Gamma_2 \vdash \text{let } \langle x, y \rangle = s \text{ in } t : C} \\
\\
\frac{}{\Delta \vdash tt, ff : !^m \text{bit}} \quad \frac{! \Delta, \Gamma_1 \vdash s : \text{bit} \quad ! \Delta, \Gamma_2 \vdash t, t' : A}{! \Delta, \Gamma_1, \Gamma_2 \vdash \text{if } s \text{ then } t \text{ else } t' : A} \\
\\
\frac{\Delta \vdash s : \text{bit}}{\Delta \vdash \text{new}(s) : \text{qbit}} \quad \frac{\Delta \vdash s : \text{qbit}}{\Delta \vdash \text{meas}(s) : !^m \text{bit}} \quad \frac{\Delta \vdash s : \text{qbit}}{\Delta \vdash \text{had}(s) : \text{qbit}} \quad \frac{\Delta \vdash s : \text{qbit} \otimes \text{qbit}}{\Delta \vdash \text{CNOT}(s) : \text{qbit} \otimes \text{qbit}}
\end{array}$$

Table 1. Typing rules for the quantum lambda-calculus

```

lamb :: (AddVar x a c2 c1) =>
  x -> j c1 b -> j c2 (a -> b)

```

The variable `x` is made out of a type-level encoding of the natural numbers

```

data Z = Z
data S a = S a

```

A typical instance of `MyJudg` is

```

data J c a = J (c -> a)

```

In the following paragraphs, we describe the typing rules and the constraints that are being used in the class `MyJudg`. The spirit of the typing rules is that they are top-to-bottom: the context and the type of a typing judgment is function of what is above.

Typing a free variable. The type of a free variable is exactly the type at the corresponding place in the context. Moreover, the size of the context needs to be enough for the context to contain the variable but it does not need to be larger. We can summarize this with the rule

$$\frac{}{[\text{Used}A_n, \text{Free}A_{n-1}, \dots, \text{Free}A_0] \vdash n : A_n}$$

The class `GetVar` is twofold: first, it characterizes the relationship between the name `x` of the variable, its type `a` and the context `c` and it also produces a projection to get back the variable out of the context.

```

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

```

The functional dependency enforces that the context `c` is function of the variable and its type.

Note that since the type `x` is always uniquely determined by the term, given an open `c` and an open `a`, the type checker will always manage to satisfy the constraint `GetVar x a c`.

Typing an application. In the regular simply-typed lambda-calculus, the context `c3` would be equal to `c1` and `c2`. Since the rules are top-to-bottom, we can authorize the context to be of distinct size, provided that they match on their common part, and that `c3` is equal to the largest of `c1` and `c2`. In particular, `c3` is function of `c1` and `c2`: we shall have a functional dependency. We also need to be able to get back the contexts `c1` and `c2` out of `c3`: for this

reason we have the function `split :: c3 -> (c1, c2)`. The class `Zip` then starts as follows.

```

class Zip c1 c2 c3 | c1 c2 -> c3 where
  split :: c3 -> (c1, c2)
instance Zip () c c where
  split c = ((), c)
instance Zip c () c where
  split c = (c, ())
instance Zip () () () where
  split () = ((), ())

```

Regarding the flags `Used` and `Free`, a variable in `c3` is used if and only if it is used at least in one of `c1` or `c2`. In the case where it is free in both contexts, 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 need to add this as an instance and enforce the fact that these types should be equal. This is possible thanks to the following pair of classes

```

class IsEqAux a b | a -> b where
  iseqaux :: a -> b
instance IsEqAux a a where
  iseqaux x = x

class (IsEqAux a b, IsEqAux b a) => IsEq a b where
  iseqid :: a -> b
instance IsEq a a where
  iseqid x = iseqaux x

```

The class `IsEqAux` with its unique instance will ask the Haskell type checker to unify `b` with `a`. The second class will ask the Haskell type to unify `a` with `b` and `b` with `a`, as well as hiding the functional dependency. The class has an operation `iseqid :: a -> b` that behaves as an identity. Using the Haskell interpreter `ghci`, we can check that it behaves as expected:

```

*> let u = undefined
*> :t u :: IsEq a b => b
u :: IsEq a b => b :: a
*> :t u :: IsEq (a1 -> (a2 -> a3)) ((b1 -> b2) -> b3) => a1

```

```

u :: IsEq (a1 → (a2 → a3)) ((b1 → b2) → b3) ⇒ a1 ::
  b1 → b2
*> :t u :: IsEq (a1 → (a2 → a3)) ((b1 → b2) → b3) ⇒ b3
u :: IsEq (a1 → (a2 → a3)) ((b1 → b2) → b3) ⇒ b3 ::
  a2 → a3

```

We can now use this class to enforce the equality of types, and rewrite the broken instance of Zip as follows.

```

instance Zip c1 c2 c3, IsEq c a, IsEq c b) ⇒
  Zip (Free a, c1) (Free b, c2) (Free c, c3)
  where
  split (Free c, c3) =
    let (c1, c2) = split c3 in
      (Free (iseqid c), c1), (Free (iseqid c), c2)

```

Note that we use the fake identity `iseqid` for going from a term of type `c` to a term of type `a` and from a term of type `c` to a term of type `b`.

The remaining instances are written similarly.

```

instance Zip c1 c2 c3, IsEq c a, IsEq c b) ⇒
  Zip (Free a, c1) (Used b, c2) (Used c, c3)
  where ...
instance Zip c1 c2 c3, IsEq c a, IsEq c b) ⇒
  Zip (Used a, c1) (Free b, c2) (Used c, c3)
  where ...
instance Zip c1 c2 c3, IsEq c a, IsEq c b) ⇒
  Zip (Used a, c1) (Used b, c2) (Used c, c3)
  where ...

```

Typing lambda-abstractions. Given a judgment $j \text{ c1 } b$ and a variable x , one should be able to write a judgment of final type $a \rightarrow b$. Since the typing judgments are constructed top-to-bottom, the context $c1$ is given. There are two possibilities for the resulting lambda-abstraction.

1. Either x is larger than the context. Then it is a free variable, certainly unused, that can be of any type without any restriction. The typing judgment is

$$\frac{[F_n A_n, \dots, F_0 A_0] \vdash s : B}{[F_n A_n, \dots, F_0 A_0] \vdash \lambda x. s : A \rightarrow B}$$

2. Or x is already present in $c2$. Then its type in the context can be set to anything, and its flag is now `Free`.

$$\frac{[F_n A_n, \dots, F_x A_x, \dots, F_0 A_0] \vdash s : B}{[F_n A_n, \dots, \text{Free } C, \dots, F_0 A_0] \vdash \lambda x. s : A_x \rightarrow B}$$

The purpose of the class `AddVar` is to state this typing rule. Again, the typing context $c2$ is function of the other three arguments, and to stay as general as possible we have to use the class `IsEq`. We also need an operation to retrieve the context $c1$ out of the other informations.

```

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 IsEq a1 a2 ⇒
  AddVar Z a1 (Free b, c) (Free a2, c)
  where
  add Z a (Free b, c) = (Free (iseqid a), c)
instance IsEq a1 a2 ⇒
  AddVar Z a1 (Free b, c) (Used a2, c)
  where
  add Z a (Free b, c) = (Used (iseqid 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)

```

Realizing the judgment. We said that the data type `J` is an instance of `MyJudg`. This can be implemented as follows.

```

instance MyJudg J where
  lamb x (J f) = J $ \c a → f (add x a c)
  appl (J f) (J g) =
    J $ \x → let (x1, x2) = split x in f x1 (g x2)
  var x = J $ \z → get_var z x

```

The identity written in this embedded language can be written as

```

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

```

In both cases, we would like to be able to get back the function $(a \rightarrow a)$. This amounts to feed 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), we parametrize it with a class `Eval`. We force the type in the context to be `Dummy` using the class `IsEq`.

```

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, IsEq (Free Dummy) b) ⇒
  Eval (b, c) a
  where
  eval (J f) =
    eval (J $ \c → f (iseqid (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))) 0
0

```

3. Adding linearity constraints and subtyping

We now turn to the question of adding to the simply-typed lambda-calculus the linearity flags described in Section 1. Technically, we use the same terms as in Section 2 and the typing rules of the first row of Table 1.

If we still want the type system to match Haskell's type system, we have to work with type constructs a bit more evolved to handle the flag. We define a type constructor `Flag` to be able to add flags to types. The flags will be written `Dup` (for "duplicable") when there is a "!" and `Sim` (for "simple") when there is none.

```

data Flag f a = Flag a
data Dup
data Sim

```

The type $!(\text{Bool} \rightarrow \text{Qbit})$ is encoded with

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

Following what was done in Section 2, we use variables named with type integers and typing contexts made of regular lists of types with flags *Used* and *Free* (now, though, the types in the context are the new types). As previously, propositions stating constraints on the contexts, the flags and the types are stated using type classes.

Subtyping relation. We need to be able to state whether a type is a subtype of another one. We design a new type class `TestSubType` for that purpose. Due to the nature of the subtyping we use, if A is a subtype of B then both types have the same skeleton; the only parts that really change are the flags. In particular, if we have to solve `TestSubType a b`, we can safely extend a and b using a unification algorithm. To use the one of the Haskell type checker, we set a bidirectional functional dependency.

```
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 that we have 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, thanks to the type class `FlagSubType`.

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

The instances of this type class are crafted so that for all non-ambiguous cases where the constraint holds the type checker can remove it and possibly set the corresponding flag variable to the value it should have. For example, if $A < ! B$, for sure A is duplicable. The third instance takes care of this case.

These non-ambiguous cases form the four first instances. Since they overlap, we need to close the overlapping instances. The problem we have is with the case `FlagSubType Sim Dup`. It is not supposed to hold, but it is an overlapping case of the third and fourth instances. It turns out that this is all right, since the constraint we end up with, that is `IsEq Dup Sim`, is bound to fail.

Note that we do not use the type class `IsEqAux` (as we could have done). The reason is that there is a sanity check from the type checker: if we use `IsEqAux Dup Sim` as constraint, the compiler complains that no such instance exists and fails to load the type class `FlagSubType`... By setting one level of indirection, it goes through and only complains when the type class is actually used.

Typing judgments. The class `MyJudg` now becomes

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

Let's go through the type classes we use. First, `GetVar` is almost the same one as the one used in Section 2. The difference lies in the first instance that now read

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

Indeed, 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` has now one more constraint in the last instance, reading as follows.

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

Indeed, a variable can be used in both branches as long as it is duplicable.

For constraints used for the creation of the lambda-abstraction, the class `GetVar` is unchanged. However, there is a new type class `TestCont`. It is meant to account for the fact that the free variables of the body that are not the argument of the lambda-abstraction are supposed to be duplicable if the lambda-abstraction is. It 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.

Of course, they say the same thing. But from a logic programming point of view, they do not behave the same way: the first one allow to generate the flag of the type from the context, whereas the second one allow to generate the flags of the types in the context. These two propositions can be made into 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, IsEq (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, if we try on Haskell's interpreter

```
:t eval (
  let x = (lamb Z (lamb (S Z)
    (appl (var Z)
      (appl (var Z) (var (S Z)))))) in
  (lamb Z (appl x
    (lamb (S Z) ((var Z)::
      J (Used (Flag Sim Bool), ()) (Flag Sim Bool))))))
```

it answers

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

since we try to duplicate the variable `(var Z) :: Flag Sim Qbit`, that is, an non-duplicable object. If we try instead

```
:t eval (
  let x = (lamb Z (lamb (S Z)
    (appl (var Z)
      (appl (var Z) (var (S Z)))))) in
  (lamb Z (appl x
    (lamb (S Z) ((var Z)::
      J (Used (Flag Dup Bool), ()) (Flag Dup Bool))))))
```

It gives the type of the term with a subtyping constraint.

```
... :: FlagSubType f2 f1 =>
  Flag f3 (Flag Dup Bool ->
  Flag f (Flag Dup Bool -> Flag f2 Bool))
```

4. Extending to the full quantum lambda-calculus

We have now passed through most of the difficulties for encoding the full quantum lambda-calculus. What is missing from the class `MyJudg` is the three last lines of Table 1.

Classical and quantum booleans. We first concentrate on the two last ones, concerned with classical boolean and quantum bits. The class is extended as follows.

```
class MyJudg 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`.

```
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, IsEq c a, IsEq c b) =>
  Match (Free a, c1) (Free b, c2) (Free c, c3)
  where
  match (Free c, c3) =
```

```
  let (c1, c2) = match c3 in
    ((Free (iseqid c), c1), (Free (iseqid c), c2))
instance (Match c1 c2 c3, IsEq c a, IsEq c b) =>
  Match (Used a, c1) (Used b, c2) (Used c, c3)
  where
  match (Used c, c3) =
    let (c1, c2) = match c3 in
      ((Used (iseqid c), c1), (Used (iseqid c), c2))
```

it makes sure that the context `c1` and `c2` match on their common subset and that the context `c3` is the largest of the two.

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

Tensor. The typing rules for the tensor are the most complicated. They involve a special operation \vee on flags. We encode it using the functional dependency `Or`. As for the class `TestCont`, there are two versions of the map. `Or1` sets the result depending on the input, and `Or2` sets the inputs depending on the value of the result: if the result is `Sim`, we have no choice for the inputs `e` and `a`. If it is `Dup`, one cannot say anything.

```
class Or1 e a x | e a -> x
instance Or1 e Dup Dup
instance Or1 Dup e Dup
instance Or1 Sim Sim Sim
instance (IsEq a b) => Or1 Sim a b
instance (IsEq 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, we can now state the creation of a pair in the class `MyJudg`.

```
class MyJudg 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))
```

For the last term construct, we have to create a new class to expose the types `x`, `a`, `y` and `b`: the typechecker needs these clues when making of an instance of this class.

```
class MyJudgTens 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))
```

Two last classes are not yet described. The first one, called `Add2Var`, is used within. As its name implies, it behaves in a manner similar to `AddVar`, except that it works with two variables. The other typeclass is needed in the definition of `Add2Var`, and it asserts the non-equality of two names.

```
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 (IsEq a1 a2, AddVar m b c2 c1) ⇒
  Add2Var Z a1 (S m) b (Free y, c2) (Free a2, c1)
instance (IsEq a1 a2, AddVar m b c2 c1) ⇒
  Add2Var Z a1 (S m) b (Free y, c2) (Used a2, c1)
instance (IsEq a1 a2, AddVar m b c2 c1) ⇒
  Add2Var (S m) b Z a1 (Free y, c2) (Free a2, c1)
instance (IsEq 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 happen in the creation of a lambda-abstraction. The second instance always fails since the names are both equal to `Z`. The third and fourth one add the first variable to the context, and call `AddVar` to add the second one to its tail (again, following the same idea as for the creation of a lambda-abstraction). The fifth and sixth ones do the opposite. The last one performs the induction step.

5. Discussion

5.1 Some examples of terms

Non-duplicability Using quantum computation, one build a perfect coin toss using the code

```

b :: MyJudg j m ⇒ j m () (m (Flag f' Bool))
b = meas (had (new ff))

```

Although it should create a quantum bit, this term is duplicable: the following code type check.

```

bb :: (StrongMonad m, QC m) ⇒
  m (Flag f (Flag Dup Bool, Flag Dup Bool))
bb = appl (lamb Z (tens (var Z) (var Z)))
  (meas (had (new ff)))

```

Catching errors If instead of passing the result of the measurement, we pass the quantum bit and measure inside, the term does not type check.

```

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

```

```

<interactive>:1:7:
  No instance for (IsEq (Flag Sim Qbit)
    (Flag Dup Qbit))
    arising from a use of `lamb`
Possible fix:
  add an instance declaration for
    (IsEq (Flag Sim Qbit) (Flag Dup Qbit))
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 we get a meaningful message and the place were the type checker stopped. This is one of the interest of having the Haskell type checker do the job for us.

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
... :: MyJudg j m ⇒
  j m (Free b, ())
  (m (Flag Sim (a → m (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`
  ...

```

A entangled pair of function. We give a large example to show the robustness of the implementation. In this example, we construct a pair of functions. The thing is, the functions share a pair of quantum bits, making them non-duplicable.

```

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

pair_fun = (
  lettens x y (cnot (tens (new tt) (had (new tt))))
  (tens (lamb z (ifx (var z)
    (meas (var x))
    (meas (had (var x))))))
  (lamb z (ifx (var z)
    (meas (var y))
    (meas (had (var y))))))

```

We first build a pair of quantum bits $\langle x, y \rangle$ and use them in each function of the pair. As in the previous example, the functions are not duplicable.

```

*> :t eval pair_fun
pair_fun ::
  (FlagSubType f1 f1, FlagSubType f2 f2,
  StrongMonad m, QC m) ⇒
  m (Flag
    Sim
    (Flag Sim (Flag f2 Bool → m (Flag f' Bool)),
    Flag Sim (Flag f1 Bool → m (Flag f'1 Bool))))

```

5.2 Alternative approach: de Bruijn indices

It is interesting to note that this type class programming is reasonably general. For example, one can perform the same reasoning using de Bruijn indices instead of named variables. There are no term variables anymore, and the typing context is now simply a list of types corresponding to the “free variable”.

5.2.1 Simply-typed lambda-calculus with de Bruijn indices

We propose an encoding of the simply-typed lambda-calculus with de Bruijn indices. The typing rules are sketched in Table 2.

The Haskell representation of a type is again simply a Haskell type. A typing judgment will be typed with $j :: * \rightarrow * \rightarrow *$ taking as first argument a context and as second argument the final type of the term. The context is a again a list of types represented as a tower of pairs $(A, (B, \dots ()))$. The typing rules can now be straightforwardly made into a type class, as follows.

$$\frac{}{A, \Delta \vdash 0 : A} \quad \frac{\Delta \vdash n : A}{B, \Delta \vdash n + 1 : A} \quad \frac{A, \Delta \vdash B}{\Delta \vdash \lambda s : A \rightarrow B} \quad \frac{\Delta \vdash s : A \rightarrow B \quad \Delta \vdash t : A}{\Delta \vdash (s)t : B}$$

Table 2. Typing rules for the simply-typed lambda-calculus

class MyJudg j **where**

```
lamb :: j (a, cont) b → j cont (a → b)
appl :: j cont (a → b) → j cont a → j cont b
z :: j (a, b) a
s :: j c a → j (b, c) a
```

The data type `J` can still be used as an instance of `MyJudg`.

instance MyJudg J **where**

```
lamb (J f) = J (\c a → f (a, c))
appl (J f) (J g) = J (\x → f x (g x))
z = J (\(y, z) → y)
s (J f) = J (\(y, z) → f z)
```

If we built a closed lambda-term, we can evaluate it by feeding it with the empty context. There is no need for a class `Eval`.

```
eval (J f) = f ()
```

The Church numeral 2 is evaluated using

```
f = eval (lamb (lamb (appl (s z) (appl (s z) z))))
```

5.2.2 Strict linearity

We now turn to the question of the addition of linearity constraints to this language. To keep the length of the paper reasonable, we concentrate on strict linearity, but the same calculus as presented in Section 3 can be done with de Bruijn indices.

The terms and types are the same as for the simply-typed lambda-calculus; the difference lies in the typing rules. If they are relatively simple to express in the lambda-calculus with named variables (top row of Table 3), one needs to include auxiliary informations if we still want to use de Bruijn indices. Indeed, consider the rule for application: the contexts of the hypotheses have to be mangled; this is a difficult things to do with de Bruijn notation. We therefore modify the context to include not one but two lists (bottom row of Table 3). The second one is the one already encountered in the simply-typed lambda-calculus. The first one is a list of values tt/ff , saying whether the variable in the corresponding place is actually supposed to be there or not.

The typing rules are adapted accordingly, using two propositions `MakeFalse` and `Zip`. The first one states that Δ and X have the same size and that X is only made of ff . The second proposition is similar to the previous ones already encountered. It states that a variable can only occur once in the two branches. That is, each of the three lists have the same size and for each index the value in X_1 and X_2 cannot be both tt and the value in X_3 is the disjunction of the two others.

For the Haskell encoding of the rules of the bottom row of Table 3, we need to update the type for typing judgments. Instead of two arguments, a judgment takes three of them: the first one is the list of constraints, the second one is the list of types and the last one the type of the term. The rules are again encoded rather directly.

class MyJudg j **where**

```
lamb :: j (True, x) (a,c) b → j x c (a → b)
appl :: (Zip x1 x2 x3) ⇒
  j x1 c (a → b) → j x2 c a → j x3 c b
z :: (MakeFalse c x) ⇒ j (True, x) (a,c) a
s :: j x c a → j (False, x) (b,c) a
```

We use type classes with functional dependencies to state the propositions `MakeFalse` and `Zip`. We first need to define types for tt and ff

```
data True
data False
```

The type class `MakeFalse` is a simple induction on the structure of the context.

```
class MakeFalse a b | a → b
instance MakeFalse () ()
instance (MakeFalse b x) ⇒
  MakeFalse (a,b) (False, x)
```

The type class `Zip` is defined for all values but the pair (tt, tt) . This is on purpose: in the case of a duplicated variable, the type checker will look for such an instance. It will not find one and complain. That is precisely the expected behavior since we are purely linear.

```
class Zip c1 c2 c3 | c1 c2 → c3
instance Zip () () ()
instance Zip c1 c2 c3 ⇒
  Zip (True, c1) (False, c2) (True, c3)
instance Zip c1 c2 c3 ⇒
  Zip (False, c1) (True, c2) (True, c3)
instance Zip c1 c2 c3 ⇒
  Zip (False, c1) (False, c2) (False, c3)
```

Finally, we need a concrete implementation of a judgment. A slight modification of the type `J` do the job. The same instance as in the simply-typed lambda-calculus does the job.

```
data J x c a = J (c → a)

instance MyJudg J where
  lamb (J f) = J (\c a → f (a, c))
  appl (J f) (J g) = J (\x → f x (g x))
  z = J (\(y, z) → y)
  s (J f) = J (\(y, z) → f z)
```

```
eval (J f) = f ()
```

If we now try out the implementation on the Church numeral 1, the type checker does not complain.

```
*Main> :t eval (lamb (lamb (appl (s z) z)))
eval (lamb (lamb (appl (s z) z)))
  :: (a → b) → a → b
```

If we try out on the Church numeral 2, it fails as expected, with a missing instance for `Zip`.

```
*Main> :t eval (lamb (lamb (appl (s z)
                                (appl (s z) z))))
```

```
<interactive>:1:18:
No instance for (Zip (True, ())
                    (True, ()) (True, t))
  arising from a use of 'appl' at
  <interactive>:1:18-42
```

```
Possible fix:
add an instance declaration for
  (Zip (True, ()) (True, ()) (True, t))
```

$\frac{}{x : A \vdash x : A}$	$\frac{x : A, \Delta \vdash B}{\Delta \vdash \lambda x. s : A \rightarrow B}$	$\frac{\Delta \vdash s : A \rightarrow B \quad \Gamma \vdash t : A}{\Delta, \Gamma \vdash (s)t : B}$
$\frac{\text{MakeFalse } \Delta \ X}{(tt, X) (A, \Delta) \vdash 0 : A}$	$\frac{X \ \Delta \vdash n : A}{(ff, X) (B, \Delta) \vdash (n+1) : A}$	$\frac{(tt, X) (A, \Delta) \vdash B \quad X \ \Delta \vdash \lambda s : A \rightarrow B \quad \text{Zip } X_1 \ X_2 \ X_3 \quad X_1 \ \Delta \vdash s : A \rightarrow B \quad X_2 \ \Delta \vdash t : A}{X_3 \ \Delta \vdash (s)t : B}$

Table 3. Strictly linear lambda-calculus: top row with named variables, bottom row with de Bruijn indices.

In the first argument of `'lamb'`, namely `'(appl (s z) (appl (s z) z))'`
 In the first argument of `'lamb'`, namely `'(lamb (appl (s z) (appl (s z) z)))'`
 In the first argument of `'eval'`, namely `'(lamb (lamb (appl (s z) (appl (s z) z))))'`

Note that the error message is quite readable and points to the place in the code where the problem arose.

5.2.3 GADT's

We used type classes and functional dependencies. For the purpose of these two particular encodings, we could as well have used GADT's and type families. The strict linear encoding can be done with the following code².

```

data True
data False

type family Zip c1 c2 :: *
type instance Zip () () = ()
type instance Zip (True, c1) (False, c2) =
  (True, (Zip c1 c2))
type instance Zip (False, c1) (True, c2) =
  (True, (Zip c1 c2))
type instance Zip (False, c1) (False, c2) =
  (False, (Zip c1 c2))

type family MakeFalse c :: *
type instance MakeFalse () = ()
type instance MakeFalse (a,b) =
  (False, (MakeFalse b))

data E x c a where
  Lamb :: E (True, x) (a, cont) b →
    E x cont (a → b)
  Appl ::
    E x1 c (a → b) → E x2 c a →
    E (Zip x1 x2) c b
  Z :: E (True, (MakeFalse b)) (a,b) a
  S :: E x c a → E (False, x) (b,c) a

eval' :: E x c a → c → a
eval' (Lamb x) c = λa → eval' x (a, c)
eval' (Appl x y) c = (eval' x c) (eval' y c)
eval' Z (y, c') = y
eval' (S x) (y, c') = eval' x c'

eval e = eval' e ()

```

This is possible because the type checking is direct and the only possibility of failure is the absence of a class instance in the first case, and the non-totality of a type family in the other case (`Zip` is not defined when both arguments are `True`).

²many thanks to Stephanie Weirich for this.

In the case of subtyping, this is not possible anymore since the type class `FlagSubType` uses features that are not directly related to functional dependencies.

In the case of named variables, a similar problem occurs: as far as we tried, the forcing of the value of a type with the class `IsEq` cannot be done with type families.

5.3 Back to quantum computation

Let us go back to the embedded language in Section 4 We said in the introduction that the goal of the language was to be used together with a QRAM model, for example using the quantum state monad of Section 1.2. In particular, we want to work in the Kleisli category corresponding to the monad.

A slight modification of the classes `MyJudg`, `MyJudgTens` and `Eval` needs to be done for that goal.

```

class MyJudg 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)
  tt :: j m () (m (Flag f Bool))
  ff :: j m () (m (Flag f Bool))
  ifx :: (Match c21 c22 c2, Zip c1 c2 c3) ⇒
    j m c1 (m (Flag d Bool)) → j m c21 (m a) →
    j m c22 (m a) → j m c3 (m a)
  new :: j m c (m (Flag f Bool)) →
    j m c (m (Flag Sim Qbit))
  meas :: j m c (m (Flag f Qbit)) →
    j m c (m (Flag f' Bool))
  had :: j m c (m (Flag f Qbit)) →
    j m c (m (Flag Sim Qbit))
  cnot :: j m c (m (Flag f1 (Flag f2 Qbit,
    Flag f3 Qbit))) →
    j m c (m (Flag Sim (Flag Sim Qbit,
    Flag Sim Qbit)))
  tens :: (Zip c1 c2 c3, Or e xa x, Or e yb y) ⇒
    j m c1 (m (Flag x a)) →
    j m c2 (m (Flag y b)) →
    j m c3 (m (Flag e (Flag xa a, Flag yb b)))

class MyJudgTens j m x a y b | j → m where
  lambtens :: (TestCont c3 f1, FlagSubType f1 f,
    Add2Var n1 (Flag x a) n2 (Flag y b) c3 c1,
    Or e xa x, Or e yb y) ⇒
    n1 → n2 → j m c1 (m d) →
    j m c3 (m (Flag f ((Flag e (Flag xa a,
    Flag yb b)) → (m d))))

```

```

class Eval (m :: * -> *) c a where
  eval :: (J m c a) -> a

instance Eval m () a where
  eval (J f) = f ()

instance (Eval m c a, IsEq (Free Dummy) b) =>
  Eval m (b,c) a
  where
  eval (J f) =
  eval ((J (\c ->
  f (iseqid (Free Dummy), c))))::(J m c a)

```

The classes `GetVar` and `TestSubType` have also to be adjusted to take into account the new type construct:

```

class TestSubType (m :: * -> *) a b | a -> b, b ->
a, a b -> m where
  subType :: a -> b

instance TestSubType m Bool Bool where ...
instance TestSubType m Qbit Qbit where ...
instance (Monad m, TestSubType m a b) =>
  TestSubType m (m a) (m b) where ...
instance (TestSubType m a b, FlagSubType f f2) =>
  TestSubType m (Flag f a) (Flag f2 b) where ...
instance (TestSubType m c a, TestSubType m b d) =>
  TestSubType m (a -> b) (c -> d) where ...
instance (TestSubType m a c, TestSubType m b d) =>
  TestSubType m (a, b) (c, d) where ...

class GetVar (m :: * -> *) x a c |
  x a -> c, a -> m where
  get_var :: c -> x -> a

instance TestSubType m a b =>
  GetVar m Z b (Used a, ()) where ...
instance GetVar m n a c =>
  GetVar m (S n) a (Free u, c) where ...

```

We can now adjust the data type `J`

```
data J m c a = J (c -> a)
```

and make it an instance of `MyJudg`, parametrized by `m`

```

instance (Monad m, StrongMonad m, QC m) =>
  MyJudg J m where ...

instance (StrongMonad m) =>
  MyJudgTens J m x a y b where ...

```

On the web³ you will find an implementation of a quantum state monad with destructive measurement and another one with non-destructive measurement. The first one have `Qram` defined as

```
type Qram = [Complex Double]
```

whereas the second one has `Qram` defined a

```
type Qram = (Int, [Complex Double],
  (IntMap.IntMap Int))
```

Both state monads are instances of the classes `Monad`, `QC` and `StrongMonad`, making them compatible with the embedded lambda-calculus.

³<http://www.monoidal.net/paper/qhaskell/>

Remark. Note that simulating quantum computation is not the primary purpose of the embedding in the Kleisli category. We want to stay as close as possible to a physical QRAM device, and the embedded language should be ready for it. We claim that we succeeded in this respect.

5.4 Relation to other works

Quantum computation. There are already quite a few papers discussing quantum computation in a Haskell framework. Altenkirch and Grattage [1] design QML, a language specific for a quantum computation with quantum control. On the side of this work⁴, Haskell is only used as the language for writing the QML compiler. [2] and [7] work on interfacing quantum computation with Haskell: with arrows [7], a quantum IO-state monad [2]. The goal is to obtain a “natural” way of describing quantum feature with Haskell type classes. The main drawback of this method is that it does not address the problem raised in Listing 1. However, note that the quantum IO monad can probably be used as a candidate for the quantum state monad.

We provide a novel solution in the sense that (1) we propose an embedded language, consistent with one of the common model of quantum computation (i.e. the QRAM model) ; (2) we address the issue raised in Listing 1 by using type classes to describe linear constraints.

We believe that this approach to constraints in an embedding language is novel and could usefully be applied to other field with similar type systems.

Linearity constraints and type classes. Kiselyov [3] provides an alternative way of enforcing linearity constraints 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.

6. Conclusion

In this paper we provide 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, we also get meaningful errors from the compiler.

It is a concrete use of functional dependencies for logic programming where type functions cannot be used.

Acknowledgments

We’d like to thank the UPenn PL-Club and Stephanie Weirich for their useful feedback.

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] 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.
- [3] O. Kiselyov. Linear and affine lambda-calculi. Lecture notes, accessible on [http://okmij.org/ftp/tagless-final/course/course.html#sharp\\$linear](http://okmij.org/ftp/tagless-final/course/course.html#sharp$linear), 2010.

⁴<http://sneezy.cs.nott.ac.uk/qml/compiler/>

- [4] E. H. Knill. Conventions for quantum pseudocode. Technical report, Los Alamos National Laboratory, 1996.
- [5] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2002.
- [6] P. Selinger and B. Valiron. A lambda calculus for quantum computation with classical control. *Mathematical Structures in Computer Science*, 16:527–552, 2006.
- [7] 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.