Advanced Topics in Programming Languages (ATPL)

Quantum Circuits in Standard ML

Martin Elsman[1]

Department of Computer Science
University of Copenhagen
DIKU

November 25, 2024

[1]With som quantum material borrowed from slides by Michael Kirkedal Thomsen.

Why Writing Our Own Quantum Circuit Framework?

- Much understanding is obtained by focusing on the details of an implementation.
- Great for reproducing old ideas and for exploring new ideas!

Why Standard ML?

- A strongly-typed functional language with good abstraction mechanisms (higher-order functions, polymortphism, modules).
- Developed in the 90'es. Few (if any) modifications to the language since.

The Breakdown of Classical Reasoning

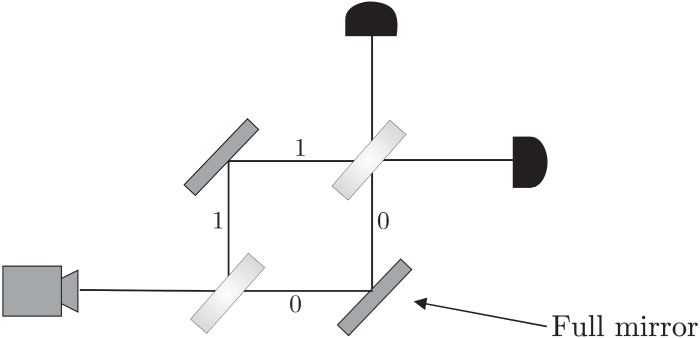


Fig. 1.9 Setup with two beam splitters.

(from *Phillip Kaye, Raymond Laflamme, and Michele Mosca. An Introduction to Quantum Computing. Oxford University Press. 2007.*)

Qubits and Single-Qubit Gates

- A *qubit* can be modelled as a two-dimensional complex vector $\begin{bmatrix}\alpha \\ \beta\end{bmatrix}$ specifying a linear combination $\alpha\,|0\rangle + \beta\,|1\rangle$ of the basis vectors $|0\rangle$ and $|1\rangle$ such that $|\alpha|^2 + |\beta|^2 = 1$.

- A *single-qubit gate* models a transformation of a qubit and can be represented as a 2×2 *unitary* complex matrix U, meaning $U^\dagger U = UU^\dagger = I$ (norm-preserving and reversible).

- Pauli-gates:

$$X = \begin{bmatrix}0 & 1 \\ 1 & 0\end{bmatrix} \qquad Y = \begin{bmatrix}0 & -i \\ i & 0\end{bmatrix} \qquad Z = \begin{bmatrix}1 & 0 \\ 0 & -1\end{bmatrix}$$

- Hadamard-gate and T-gate:

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1 \\ 1 & -1\end{bmatrix} \qquad T = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 0 \\ 0 & e^{i\pi/4}\end{bmatrix}$$

The Hadamard Gate Models a Beam Splitter

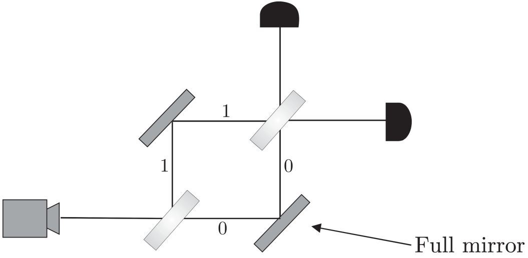


Fig. 1.9 Setup with two beam splitters.

One Hadamard gate puts a qubit in "superposition", but two Hadamard gates in sequence (modelled using matrix multiplication), acts as the identity:

$$HH = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = I$$

Multi-Qubit Circuits

A state of more qubits is modelled as a product of all standard bases.

- A two-qubit state is $\alpha\,|00\rangle + \beta\,|01\rangle + \gamma\,|10\rangle + \theta\,|11\rangle$, where $|\alpha|^2 + |\beta|^2 + |\gamma|^2 + |\theta|^2 = 1$.
- A three-qubit state is modelled by an eight-element complex vector.

$$|000\rangle = \begin{bmatrix}1\\0\\0\\0\\0\\0\\0\\0\end{bmatrix} \qquad |001\rangle = \begin{bmatrix}0\\1\\0\\0\\0\\0\\0\\0\end{bmatrix} \qquad |010\rangle = \begin{bmatrix}0\\0\\1\\0\\0\\0\\0\\0\end{bmatrix} \qquad \cdots \qquad |111\rangle = \begin{bmatrix}0\\0\\0\\0\\0\\0\\0\\1\end{bmatrix}$$

Circuit Diagrams

- Circuits may be *drawn* using a simple diagram notation with each qubit represented by a horizontal line, single-qubit operations drawn as boxes, and swaps drawn by interchanging two lines.
- Tensors are implicit and represented as vertical composition whereas sequential composition is horizontal.
- Control-gates use a special “connect notation”.
- Example diagram and semantics:

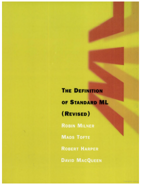
```
              .---.
----------*---| Z |---*-----------------*----
          |   '---'   |                 |
          |           |                 |
  .---. .-+-. .---. .-+-. .---.       .-+-.
--| H |-| X |-| Z |-| X |-| H |-.   .-| X |--
  '---' '---' '---' '---' '---'  \ /  '---'
                                  /
                                 / \  .---.
--------------------------------'   '-| Y |--
                                      '---'
```

$$\begin{bmatrix} i & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & i & 0 & 0 & 0 & 0 & 0 \\ 0 & i & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & i & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & i & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & i \\ 0 & 0 & 0 & 0 & i & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & i & 0 \end{bmatrix}$$

Terminology

- **Superposition**: A single qubit $\alpha\,|0\rangle + \beta\,|1\rangle$ is in a *superposition* state if it is in a non-trivial combination of the basis states (both α and β are non-zero).
- **Entanglement**: Two qubits are *entangled* if there is a dependency between their individual states. Entanglement may be created using control-gates.
- **Measurement**: Projects the state of a qubit onto one of the basis vectors. Measurement thus "destroys" any superposition of a qubit and affects also other qubits that a qubit is entangled with!

Standard ML – the Language



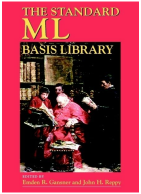
A strongly-typed functional language, specified (i.e., defined) separately from any implementation (much like C).

Compilers

- Quite a few compilers are avaiable, including MosML, Mlton, SML/NJ, MLKit, and PolyML.
- MLton and MLKit support the use of so-called mlb-files to organise the dependencies of source files.

Libraries

- A **large stadardised basis library** is specified and implemented by most compilers.
- A package system **smlpkg** is available for controling the use of non-standardised libraries.



A Module for Specifying Circuits

```
signature CIRCUIT =
  sig
    datatype t =
         I | X | Y | Z | H
       | SW       (* swap *)
       | C of t (* control *)
       | Tensor of t * t
       | Seq of t * t

    val oo   : t * t → t
    val **   : t * t → t
    val pp   : t → string
    val draw : t → string
    val dim  : t → int
  end
```

Notice:

- Read signatures, before reading implementations!
- Tensor composition (infix **) is binary only!
- Sequential composition (infix oo) takes circuits with the same dimension!
- The `dim` function returns the *dimensionality* of a circuit (i.e., the number of qubits it works on) and fails if a sequential composition operator is misused.

A Recursive Function for Identifying the Circuit Dimension

```
fun dim (t:t) : int =
    case t of
        Tensor(a,b) => dim a + dim b
      | Seq(a,b) =>
        let val d = dim a
        in if d <> dim b
           then raise Fail "Sequence error: \
                            \mismatching dimensions"
           else d
        end
      | SW => 2
      | C t => 1 + dim t
      | _ => 1
```

A Standard ML Library for Complex Numbers

Available at `http://github.com/diku-dk/sml-complex`

```
signature COMPLEX = sig
  type complex

  val mk       : real * real → complex
  val fromRe   : real → complex
  val fromIm   : real → complex
  val fromInt  : int → complex
  val conj     : complex → complex
  val re       : complex → real
  val im       : complex → real
  val mag      : complex → real
  val ~        : complex → complex
  val +        : complex * complex → complex
  val -        : complex * complex → complex
  val *        : complex * complex → complex
  ...
  val sqrt     : complex → complex
  val exp      : complex → complex
  val abs      : complex → complex
  val pow      : complex * complex → complex
  val fmt      : StringCvt.realfmt → complex → string
  val fmtBrief : StringCvt.realfmt → complex → string
  val toString : complex → string
end
```

A Standard ML Library for Matrices using "Pull-Arrays"

Available at `http://github.com/diku-dk/sml-matrix`

```
signature MATRIX = sig
  type α t

  val fromListList   : α list list → α t
  val dimensions     : α t → int * int
  val sub            : α t * int * int → α
  val map            : (α → β) → α t → β t
  val map2           : (α * β → γ) → α t → β t → γ t
  val listlist       : α t → α list list
  val transpose      : α t → α t
  val memoize        : α t → α t
  val tabulate       : int * int * (int*int → α) → α t
  val pp             : int → (α → string) → α t → string
  val ppv            : int → (α → string) → α vector → string

  val dot_gen        : (α*α→α) → (α*α→α) → α → α vector → α vector → α
  val matmul_gen     : (α*α→α) → (α*α→α) → α → α t → α t → α t
  val matvecmul_gen  : (α*α→α) → (α*α→α) → α → α t → α vector → α vector
  ...
end
```

Implementation:

```
type α t = {rows:int,cols:int,get: (int * int) → α}
```

The function `memoize` may be used to materialize a matrix in memory.

The Semantics of Circuits — the SEMANTICS signature

```
signature SEMANTICS =
  sig
    type mat = Complex.complex Matrix.t
    val pp_mat : mat → string
    val sem    : Circuit.t → mat

    eqtype ket
    val ket    : int list → ket
    val pp_ket : ket → string

    type state
    val pp_state : state → string
    val init     : ket → state
    val eval     : Circuit.t → state → state

    type dist = (ket*real) vector
    val pp_dist      : dist → string
    val measure_dist : state → dist
  end
```

The Semantics of Circuits — Auxiliary Math

Kronecker Product:[2]

$$(A \otimes B)_{ij} \;=\; A_{(i/p,j/q)} B_{(i\%p,j\%q)}$$

where

$$(p, q) = Dim(B)$$

Generalised Control:

$$(Cntrl\ A)_{ij} \;=\; \begin{cases} A_{(i-p,j-q)} & i \geq p \wedge j \geq q \\ I_{ij} & i < p \wedge j < q \\ 0 & \text{otherwise} \end{cases}$$

where

$$(p, q) = Dim(A)$$

[2] `https://en.wikipedia.org/wiki/Kronecker_product`

The Semantics of Circuits — Auxiliary Functions

```
(* See https://en.wikipedia.org/wiki/Kronecker_product *)
fun tensor (a: mat,b:mat) : mat =
    let val (m,n) = M.dimensions a
        val (p,q) = M.dimensions b
    in M.tabulate(m * p, n * q,
                  fn (i,j) ⇒
                     C.* (M.sub(a,i div p, j div q),
                          M.sub(b,i mod p, j mod q)))
    end

(* Generalised control - see section 2.5.7 in
   https://iontrap.umd.edu/wp-content/uploads/2016/01/Quantum-Gates-c2.pdf
 *)
fun control (m: mat) : mat =
    let val (n,_) = M.dimensions m
    in M.tabulate(2*n,2*n,
                  fn (r,c) ⇒                             (* 1 0 0 0 *)
                     if r >= n andalso c >= n            (* 0 1 0 0 *)
                     then M.sub(m,r-n,c-n)               (* 0 0 a b *)
                     else if r = c then C.fromInt 1      (* 0 0 c d *)
                     else C.fromInt 0)
    end

fun fromIntM iss : mat =
    M.fromListList (map (map C.fromInt) iss)

fun matmul (t1:mat,t2:mat) : mat =
    M.matmul_gen C.* C.+ (C.fromInt 0) t1 t2
```

The Semantics of Circuits — the `sem` Function

```
fun sem (t:Circuit.t) : mat =
    let open Circuit
        val c0 = C.fromInt 0
        val c1 = C.fromInt 1
        val cn1 = C.˜ c1
        val ci = C.fromIm 1.0
        val cni = C.˜ ci
    in case t of
           I ⇒ fromIntM [[1,0],                                (* dim 1 *)
                          [0,1]]
         | X ⇒ fromIntM [[0,1],                                (* dim 1 *)
                          [1,0]]
         | Y ⇒ M.fromListList [[c0,cni],                       (* dim 1 *)
                                [ci,c0]]
         | Z ⇒ fromIntM [[1,0],[0,˜1]]                         (* dim 1 *)
         | H ⇒ let val rsqrt2 = C.fromRe (1.0 / Math.sqrt 2.0) (* dim 1 *)
               in M.fromListList [[rsqrt2,rsqrt2],
                                  [rsqrt2,C.˜ rsqrt2]]
               end
         | SW ⇒ fromIntM [[1,0,0,0],                           (* dim 2 *)
                           [0,0,1,0],
                           [0,1,0,0],
                           [0,0,0,1]]
         | Seq(t1,t2) ⇒ matmul(sem t2,sem t1)          (* dim t1 = dim t2 *)
         | Tensor(t1,t2) ⇒ tensor(sem t1,sem t2)       (* dim t1 * dim t2 *)
         | C t ⇒ control (sem t)                             (* dim t + 1 *)
    end
```

Example Semantics – `quantum_ex1.sml`

```
open Circuit Semantics
infix 3 oo
infix 4 **
val c = (I ** H oo C X oo Z ** Z oo C X oo I ** H) ** I oo I ** SW oo C X ** Y
val () = (print ("Circuit for c = " ^ pp c ^ ":\n" ^ draw c ^ "\n");
          print ("Semantics of c:\n" ^ pp_mat(sem c) ^ "\n"))
```

The result of running the code:

```
Circuit for c = (I ** H oo CX oo Z ** Z oo CX oo I ** H) ** I oo I ** SW oo CX ** Y:
              .---.
----------*---| Z |---*-----------------*----
          |   '---'   |                 |
          |           |                 |
  .---. .-+-. .---. .-+-. .---.         .-+-.
--| H |-| X |-| Z |-| X |-| H |-.   .-| X |--
  '---' '---' '---' '---' '---'  \ /  '---'
                                  /
                                 / \  .---.
--------------------------------'   '-| Y |--
                                      '---'

Semantics of c:
~i  0  0  0  0  0  0  0
 0  0  i  0  0  0  0  0
 0 ~i  0  0  0  0  0  0
 0  0  0  i  0  0  0  0
 0  0  0  0  0 ~i  0  0
 0  0  0  0  0  0  0  i
 0  0  0  0 ~i  0  0  0
 0  0  0  0  0  0  i  0
```

Implementation of Kets and States

A state of n qubits is modelled as a 2^n-dimensional complex vector.

```
type ket = int list (* list of 0's and 1's *)

fun ket xs = xs

fun pp_ket (v:ket) : string =
    "|" ^ implode (map (fn i => if i > 0 then #"1"
                                else #"0") v) ^ ">"

type state = C.complex vector

fun init (is: ket) : state =
    let val i = foldl (fn (x,a) => 2 * a + x) 0 is
    in Vector.tabulate(pow2 (length is),
                       fn j => if i = j then C.fromInt 1
                               else C.fromInt 0)
    end
```

Ket-Distributions

When evaluating a circuit in an initial state, the output is a state for which we can present the “output-ket probability distribution”:

```
type dist = (ket*real) vector

fun pp_dist (d:dist) : string =
    Vector.foldr (fn ((k,r),a) ⇒
                     (pp_ket k ^ " : " ^ pp_r r) :: a) nil d
                 |> String.concatWith "\n"

fun toKet (n:int, i:int) : ket =
    (* state i \in [0;2^n-1] among total states 2^n, in binary *)
    let val s = StringCvt.padLeft #"0" n (Int.fmt StringCvt.BIN i)
    in CharVector.foldr (fn (#"1",a) ⇒ 1::a | (_,a) ⇒ 0::a) nil s
    end

fun dist (s:state) : real vector =
    let fun square r = r*r
    in Vector.map (square o Complex.mag) s
    end

fun measure_dist (s:state) : dist =
    let val v = dist s
        val n = log2 (Vector.length s)
    in Vector.mapi (fn (i,p) ⇒ (toKet(n,i), p)) v
    end
```

Evaluation

```
fun eval (x:Circuit.t) (v:state) : state =
    M.matvecmul_gen C.* C.+ (C.fromInt 0) (sem x) v
```

Example Evaluation Code

```
val k = ket [101]
val () = (print ("Result distribution when evaluating c on " ^ pp_ket k ^ " :\n");
          print (pp_dist(measure_dist(eval c (init k))) ^ "\n\n"))
```

Output

```
Result distribution when evaluating c on |101> :
|000> : 0
|001> : 0
|010> : 0
|011> : 0
|100> : 1
|101> : 0
|110> : 0
|111> : 0
```

Exercises

See `http://github.com/diku-dk/atpl-sml-quantum`

- Clone the repository and read the `README.md` file.

Project Ideas

- The `README.md` file also mentions a series of project ideas.

Bonus Slide on Drawing Diagrams

The source code contains an auxiliary library for drawing diagrams, which is used by the `Circuit.draw` function:

```
signature DIAGRAM =
  sig
    type t
    val box      : string → t       (* dim 1 *)
    val line     : t                (* dim 1 *)
    val cntrl    : string → t       (* dim 2 *)
    val swap     : t                (* dim 2 *)
    val par      : t * t → t        (* dim(seq(a,b)) = max(dim a, dim b) *)
    val seq      : t * t → t        (* dim(par(a,b)) = dim a + dim b *)
    val toString : t → string
  end
```

Notice:

- The type `t` is a `string list`, but kept abstract! Each list entry represents a text line.
- Only `par` and `seq` adds spacing, other combinators draw to the boundary box (ensures proper associative drawing behavior).