Introduction to Probabilistic and Quantum Programming

Part II

Ugo Dal Lago

BISS 2014, Bertinoro

Section 1

Probabilistic Programming Languages

Flipping a Coin

- Adding probabilistic behaviour to any programming language is relatively **easy**, at least from a purely **linguistic** point of view.
- The naïve solution is to endow your favourite language with a primitive that, when executed, "flips a fair coin" returning 0 or 1 with equal probability.
  - In **imperative** programming langauges, this can take the form of a keyword `rand`, to be used in expressions;
  - In **functional** programming languages, one could also use a form of binary, probabilistic sum, call it $\oplus$, as follows:

    ```
    letrec f x = x (+) (f (x+1))
    ```
- How do we get the necessary randomness, when executing programs?
  - By a source of *true* randomness, like physical randomness.
  - By *pseudorandomness* (we will come to that later).

Sampling

- If incepted into a universal programming language, binary uniform choice is enough to encode sampling from any **computable** distribution.
- As an example, if `f` is defined as follows

  ```
  letrec f x = x (+) (f (x+1))
  ```

  then `f(0)` produces the exponential distribution

$$\{0^{\frac{1}{2}}, 1^{\frac{1}{4}}, 2^{\frac{1}{8}}, \ldots\}.$$

- A **real number** $x \in \mathbb{R}$ is **computable** if there is an algorithm $\mathcal{A}_x$ outputting, on input $n \in \mathbb{N}$, a rational number $q_n \in \mathbb{Q}$ such that $|x - q_n| < \frac{1}{2^n}$.
- A **computable distribution** is one such that there is an algorithm $\mathcal{B}$ that, on input n, outputs the code of $\mathcal{A}_{p^n}$, where p^n is the probability the distribution assigns to n.

Theorem

PTMs are universal for computable distributions.

True Randomness vs. Pseudorandomness

- Having access to a source of true randomness is definitely not trivial.
- One could make use, as an example, of:
  - Keyboard and mouse actions;
  - External sources of randomness, like sound or movements;
  - TRNGs (True Random Number Generators).
- A **pseudorandom generator** is a deterministic algorithm $\mathcal{G}$ from strings in Σ^* to Σ^* such that:
  - $|\mathcal{G}(s)| > |s|$;
  - $\mathcal{G}(s)$ is somehow **indistinguishable** from a truly random string t *of the same length.*
- The point of pseudorandomness is **amplification**: a short truly random string is turned into a longer string which is not random, but looks so.

Programming vs. Modeling

- Probabilistic *models*, contrarily to probabilistic *languages*, are pervasive and very-well studied from decades.
  - Markov Chains
  - Markov Processes
  - Stochastic Processes
  - ...
- A probabilistic **program**, however, can indeed be seen as a way to concisely specify a model, for the purpose of doing, e.g., machine learning or inference.
  - A quite large research community is currently involved in this effort, mainly in the United States (for more details, see `http://probabilistic-programming.org`).
  - Roughly, you cannot only "flip a coin", but you can also incorporate **observations** about your dataset in your program.

Programming vs. Modeling

- Probabilistic *models*, contrarily to probabilistic *languages*, are pervasive and very-well studied from decades.
  - Markov Chains
  - Markov Processes
  - Stochastic Processes
  - ...
- A probabilistic **program**, however, can indeed be seen as a way to concisely specify a model, for the purpose of doing, e.g., machine learning or inference.
  - A quite large research community is currently involved in this effort, mainly in the United States (for more details, see `http://probabilistic-programming.org`).
  - Roughly, you cannot only "flip a coin", but you can also incorporate **observations** about your dataset in your program.

A Nice Example: FUN

```
let heads1 = random (Bernoulli(0.5)) in
let heads2 = random (Bernoulli(0.5)) in
let u = observe (heads1 || heads2) in
(heads1,heads2)
```

A Nice Example: FUN

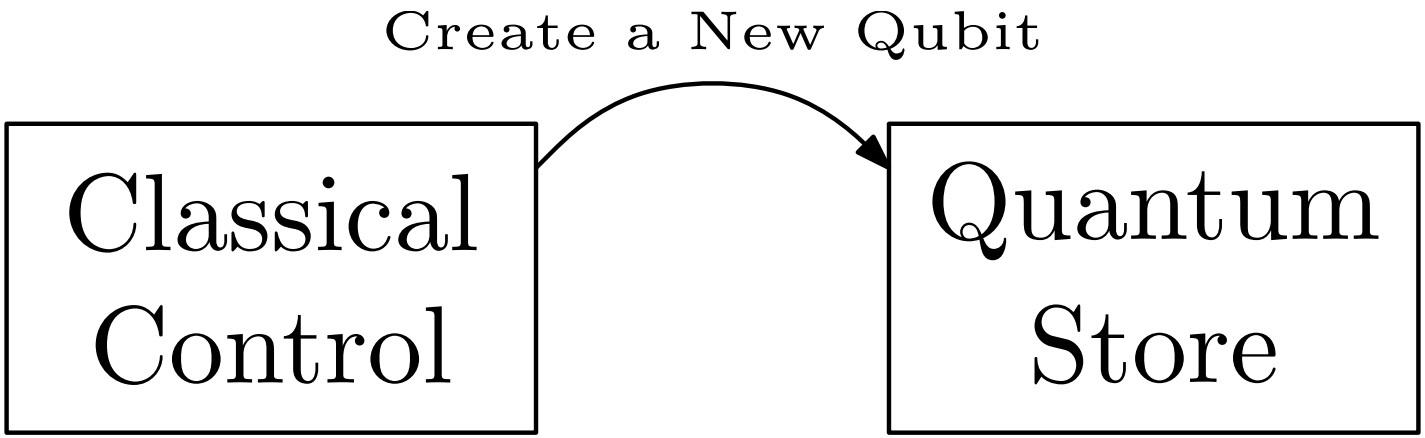
```
// prior distributions, the hypothesis
let skill() = random (Gaussian(10.0,20.0))
let Alice,Bob,Cyd = skill(),skill(),skill()
// observe the evidence
let performance player = random (Gaussian(player,1.0))
observe (performance Alice > performance Bob) //Alice beats Bob
observe (performance Bob > performance Cyd) //Bob beats Cyd
observe (performance Alice > performance Cyd) //Alice beats Cyd
// return the skills
Alice,Bob,Cyd
```

Section 2

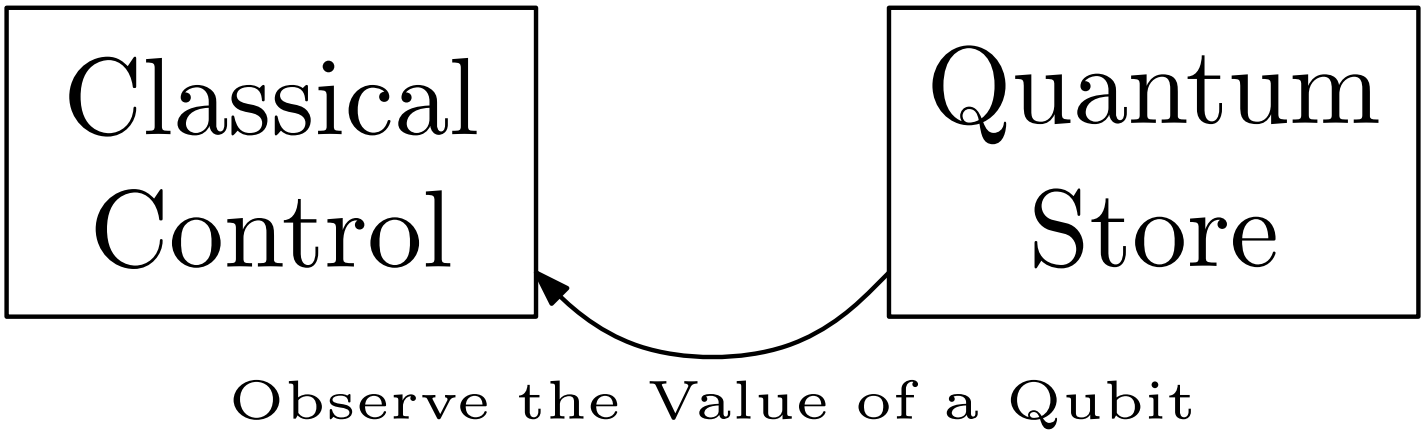
Quantum Programming Languages

Quantum Data and Classical Control

Quantum Data and Classical Control



Quantum Data and Classical Control



Quantum Data and Classical Control

Imperative Quantum Programming Languages

- QCL, which has been introduced by Ömer
- Example:

```
qufunct set(int n,qureg q)
  {
  int i;
  for i=0 to #q-1
    {
      if bit(n,i) {Not(q[i]);}
    }
  }
```

- Classical and quantum variables.
- The syntax is very reminiscent of the one of C.

Quantum Imperative Programming Languages

- qGCL, which has been introduced by Sanders and Zuliani.
  - It is based on Dijkstra's predicate transformers and guarded-command language, called GCL.
  - Features quantum, probabilistic, and nondeterministic evolution.
  - It can be seen as a generalization of pGCL, itself a *probabilistic* variation on GCL.
- Tafliovich, Hehner adapted predicative programming to quantum computation.
  - Predicative programming is not a proper programming language, but rather a methodology for specification and verification.

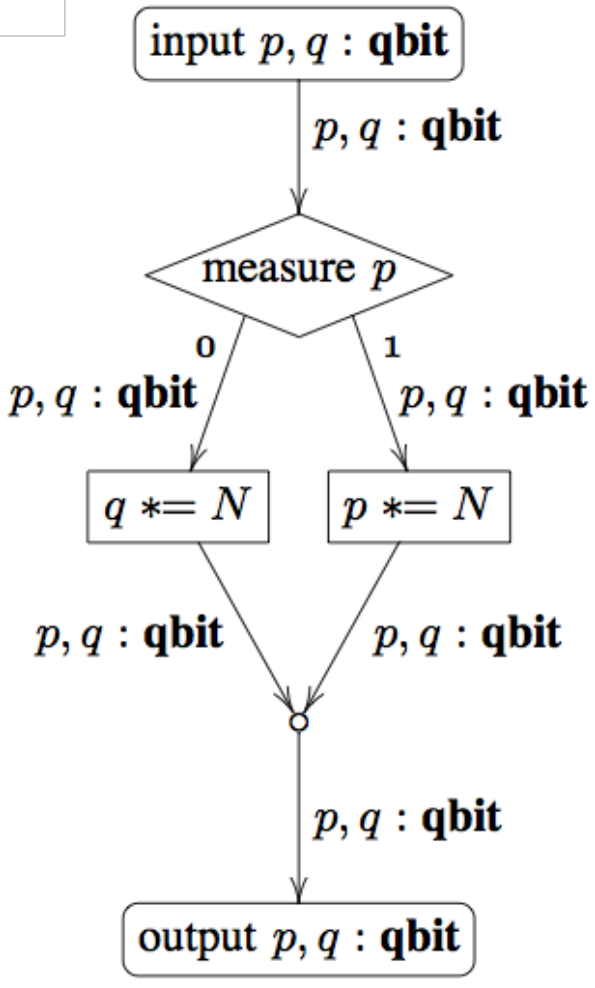
Quantum Functional Programming Languages

- QPL, introduced by Selinger.
  - A very simple, first-order, functional programming language.
  - The first one with a proper denotational semantics, given in terms of superoperators, but also handling divergence by way of domain theory.
  - A *superoperator* is a mathematical object by which we can describe the evolution of a quantum system **in presence of measurements**.
- Many papers investigated the possibility of embedding quantum programming into Haskell, arguably the most successful real-world functional programming language.
  - In a way or another, they are all based on the concept of a **monad**.

Quantum Functional Programming Languages

- QPL, introduced by Selinger.
  - A very simple, first-order, functional programming language.
  - The first one with a proper denotational semantics, given in terms of superoperators, but also handling divergence by way of domain theory.
  - A *superoperator* is a mathematical object by which we can describe the evolution of a quantum system **in presence of measurements**.
- Many papers investigated the possibility of embedding quantum programming into Haskell, arguably the most successful real-world functional programming language.
  - In a way or another, they are all based on the concept of a **monad**.

QPL: an Example



Quantum Functional Programming Languages

- In a first-order fragment of Haskell, one can also model a form of *quantum control*, i.e., programs whose interal **state** is in superposition.
  - This is Altenkirch and Grattage's QML.
- Operations are programmed at a very low level: unitary transforms become programs themselves, e.g.

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

- Whenever you program by way of the *if* construct, you should be careful and check that the two branches are **orthogonal** in a certain sense.

Quantum Functional Programming Languages

- Most work on functional programming languages has focused on **λ-calculi**, which are *minimalist*, paradigmatic languages only including the essential features.
- Programs are seen as terms from a simple grammar

$$M, N ::= x \mid MN \mid \lambda x.M \mid \dots$$

- Computation is captured by way of **rewriting**
- Quantum features can be added in many different ways.
  - By adding **quantum variables**, which are meant to model the interaction with the quantum store.
  - By allowing terms to be in **superposition**, somehow diverging from the quantum-data-and-classical-control paradigm:

$$M ::= \dots \mid \sum_{i \in I} \alpha_i M_i$$

- The third part of this course will be entirely devoted to (probabilistic and) quantum λ-calculi.

Quantum Process Algebras

- Process algebras are calculi meant to model **concurrency** and **interaction** rather than mere computation.
- Terms of process algebras are usually of the following form;

$$P, Q ::= \mathbf{0} \mid a.P \mid \overline{a}.P \mid P||Q \mid \ldots$$

- Again, computation is modeled by a form of rewriting, e.g.,

$$a.P||\overline{a}.Q \to P||Q$$

- How could we incept quantum computation? Usually:
  - Each process has its own set of classical and quantum (local) variables.
  - Processes do not only synchronize, but can also **send** classical and quantum data along channels.
  - Unitary transformations and measurements are done locally.

Other Programming Paradigms

- **Concurrent Constraint Programming**
- **Measurement-Based Quantum Computation**
- **Hardware Description Languages**
- ...

Thank You!

Questions?