Higher-Order Probabilistic Programming

A Tutorial at POPL 2019

Part I

Ugo Dal Lago

(Based on joint work with *Flavien Breuvart*, *Raphaëlle Crubillé*, *Charles Grellois*, *Davide Sangiorgi*,. . .)

POPL 2019, Lisbon, January 14th

Probabilistic Models

- The **environment** is supposed not to behave *deterministically*, but *probabilistically*.

Probabilistic Models

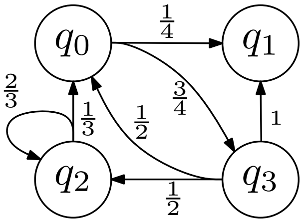
- The **environment** is supposed not to behave *deterministically*, but *probabilistically*.
- Crucial when modeling **uncertainty**.

Probabilistic Models

- The **environment** is supposed not to behave *deterministically*, but *probabilistically*.
- Crucial when modeling **uncertainty**.
- Useful to handle **complex** domains.

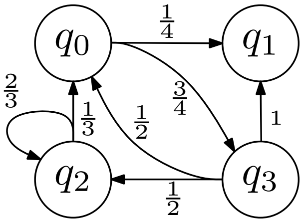
Probabilistic Models

- The **environment** is supposed not to behave *deterministically*, but *probabilistically*.
- Crucial when modeling **uncertainty**.
- Useful to handle **complex** domains.
- **Example:**



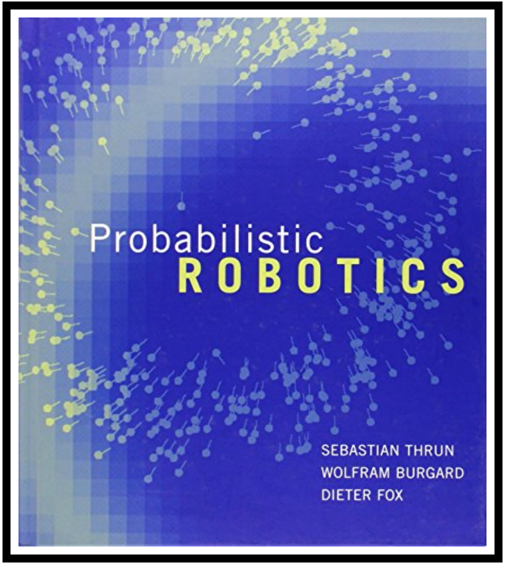
Probabilistic Models

- The **environment** is supposed not to behave *deterministically*, but *probabilistically*.
- Crucial when modeling **uncertainty**.
- Useful to handle **complex** domains.
- **Example:**



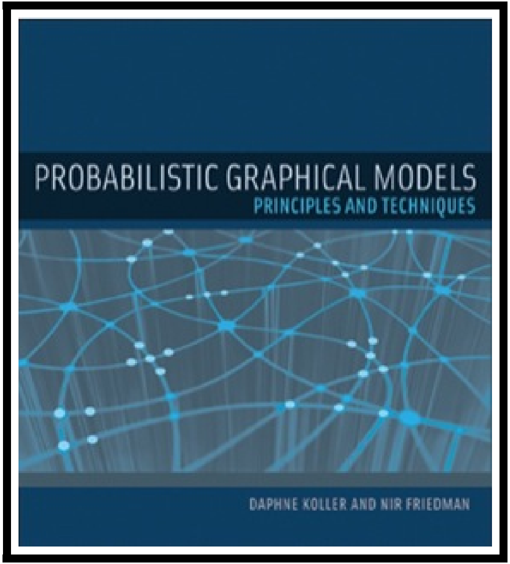
- **Abstractions:**
  - (Labelled) Markov Chains.

Probabilistic Models



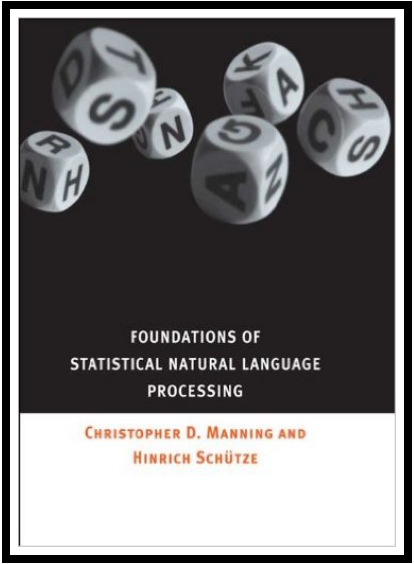
ROBOTICS

Probabilistic Models



ARTIFICIAL INTELLIGENCE

Probabilistic Models



NATURAL LANGUAGE PROCESSING

Randomized Computation

- Algorithms and automata are assumed to have the ability to **sample** from a distribution [dLMSS1956,R1963].

Randomized Computation

- Algorithms and automata are assumed to have the ability to **sample** from a distribution [dLMSS1956,R1963].
- This is a **powerful tool** when solving computational problems.

Randomized Computation

- Algorithms and automata are assumed to have the ability to **sample** from a distribution [dLMSS1956,R1963].
- This is a **powerful tool** when solving computational problems.
- **Example:**

```
Input: n > 3, an odd integer to be tested for primality;
Input: k, a parameter that determines the accuracy of the test
Output: composite if n is composite, otherwise probably prime
write n − 1 as 2^s·d with d odd by factoring powers of 2 from n − 1
WitnessLoop: repeat k times:
   pick a random integer a in the range [2, n − 2]
   x ← a^d mod n
   if x = 1 or x = n − 1 then do next WitnessLoop
   repeat s − 1 times:
      x ← x^2 mod n
      if x = 1 then return composite
      if x = n − 1 then do next WitnessLoop
   return composite
return probably prime
```

Randomized Computation

- Algorithms and automata are assumed to have the ability to **sample** from a distribution [dLMSS1956,R1963].
- This is a **powerful tool** when solving computational problems.
- **Example:**

```
Input: n > 3, an odd integer to be tested for primality;
Input: k, a parameter that determines the accuracy of the test
Output: composite if n is composite, otherwise probably prime
write n − 1 as 2^s·d with d odd by factoring powers of 2 from n − 1
WitnessLoop: repeat k times:
   pick a random integer a in the range [2, n − 2]
   x ← a^d mod n
   if x = 1 or x = n − 1 then do next WitnessLoop
   repeat s − 1 times:
      x ← x^2 mod n
      if x = 1 then return composite
      if x = n − 1 then do next WitnessLoop
   return composite
return probably prime
```

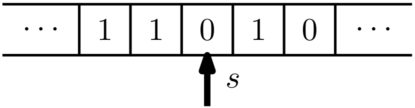
Randomized Computation

- Algorithms and automata are assumed to have the ability to **sample** from a distribution [dLMSS1956,R1963].
- This is a **powerful tool** when solving computational problems.
- **Example:**

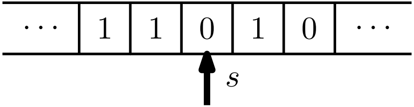
```
Input: n > 3, an odd integer to be tested for primality;
Input: k, a parameter that determines the accuracy of the test
Output: composite if n is composite, otherwise probably prime
write n − 1 as 2^s·d with d odd by factoring powers of 2 from n − 1
WitnessLoop: repeat k times:
   pick a random integer a in the range [2, n − 2]
   x ← a^d mod n
   if x = 1 or x = n − 1 then do next WitnessLoop
   repeat s − 1 times:
      x ← x^2 mod n
      if x = 1 then return composite
      if x = n − 1 then do next WitnessLoop
   return composite
return probably prime
```

- **Abstractions:**
  - Randomized algorithms;
  - Probabilistic Turing machines.
  - Labelled Markov chains.

Randomized Computation

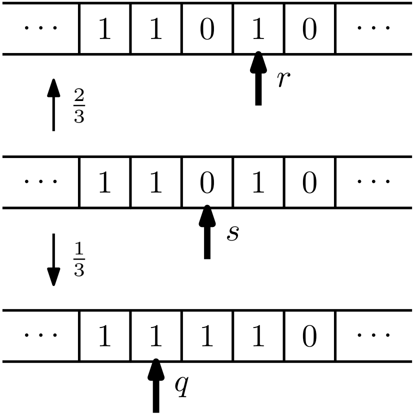


Randomized Computation



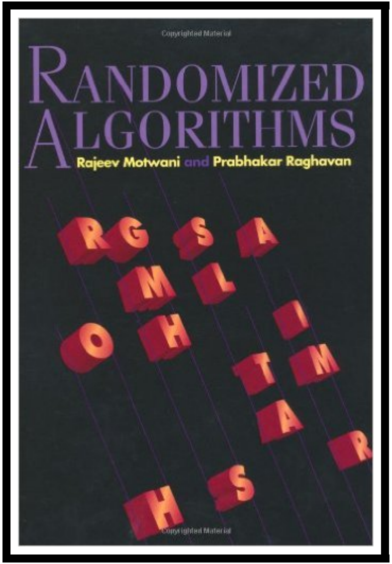
$$\delta(s,0) = \{(q,1,\leftarrow)^{\frac{1}{2}}, (r,0,\rightarrow)^{\frac{2}{3}}\}$$

Randomized Computation



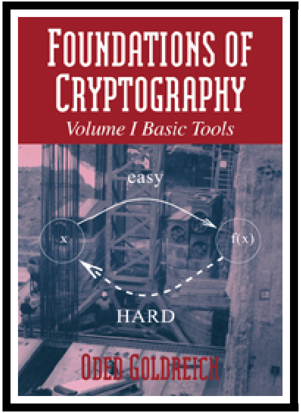
$$\delta(s, 0) = \{(q, 1, \leftarrow)^{\frac{1}{2}}, (r, 0, \rightarrow)^{\frac{2}{3}}\}$$

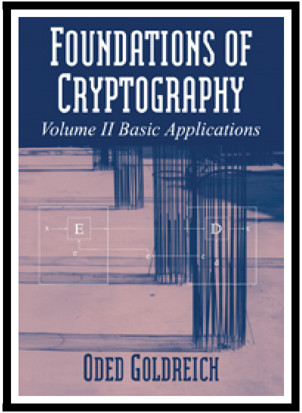
Randomized Computation



ALGORITHMICS

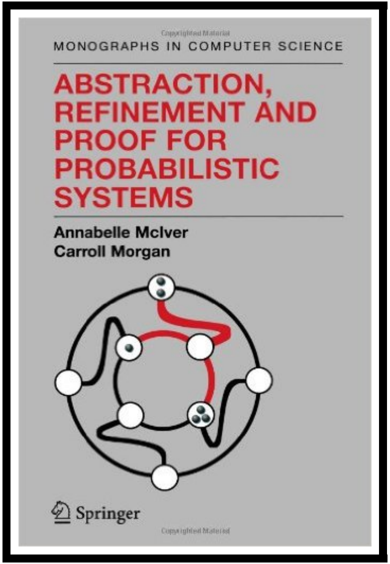
Randomized Computation





CRYPTOGRAPHY

Randomized Computation



PROGRAM VERIFICATION

What Algorithms Compute

- **Deterministic Computation**
  - For every input x, there is *at most* one output y any algorithm $\mathcal{A}$ produces when fed with x.
  - As a consequence:

$$\mathcal{A} \qquad \rightsquigarrow \qquad [\![\mathcal{A}]\!] : \mathbb{N} \rightharpoonup \mathbb{N}.$$

What Algorithms Compute

- **Deterministic Computation**
  - For every input x, there is *at most* one output y any algorithm $\mathcal{A}$ produces when fed with x.
  - As a consequence:

$$\mathcal{A} \qquad \rightsquigarrow \qquad [\![\mathcal{A}]\!] : \mathbb{N} \rightharpoonup \mathbb{N}.$$

- **Randomized Computation**
  - For every input x, any algorithm $\mathcal{A}$ outputs y with a probability $0 \leq p \leq 1$.
  - As a consequence:

$$\mathcal{A} \qquad \rightsquigarrow \qquad [\![\mathcal{A}]\!] : \mathbb{N} \rightarrow \mathscr{D}(\mathbb{N}).$$

  - The distribution $[\![\mathcal{A}]\!](n)$ sums to anything between 0 and 1, thus accounting for the probability of divergence.

Higher-Order Computation

- Mainly useful in **programming**.

Higher-Order Computation

- Mainly useful in **programming**.
- Functions are **first-class citizens**:
  - They can be passed as *arguments*;
  - They can be obtained as *results*.

Higher-Order Computation

- Mainly useful in **programming**.
- Functions are **first-class citizens**:
  - They can be passed as *arguments*;
  - They can be obtained as *results*.
- **Motivations**:
  - *Modularity*;
  - *Code reuse*;
  - *Conciseness*.

Higher-Order Computation

- Mainly useful in **programming**.
- Functions are **first-class citizens**:
  - They can be passed as *arguments*;
  - They can be obtained as *results*.
- **Motivations**:
  - *Modularity*;
  - *Code reuse*;
  - *Conciseness*.
- **Example**:

```
foldr                :: (a -> b -> b) -> b -> [a] -> b
foldr f acc []       = acc
foldr f acc (x:xs) = f x (foldr f acc xs)
```

Higher-Order Computation

- Mainly useful in **programming**.
- Functions are **first-class citizens**:
  - They can be passed as *arguments*;
  - They can be obtained as *results*.
- **Motivations**:
  - *Modularity*;
  - *Code reuse*;
  - *Conciseness*.
- **Example**:

```
foldr              :: (a -> b -> b) -> b -> [a] -> b
foldr f acc []      = acc
foldr f acc (x:xs)  = f x (foldr f acc xs)
```

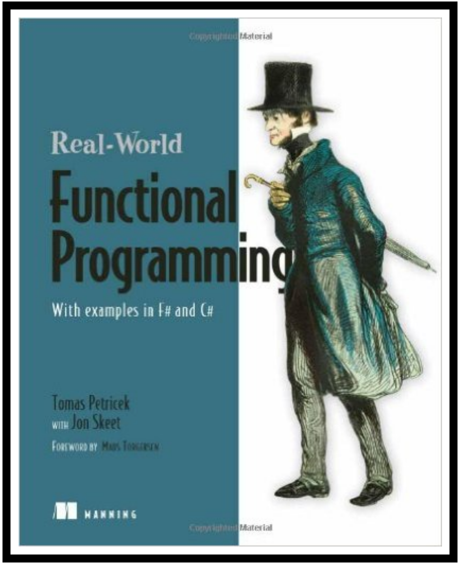
Higher-Order Computation

- Mainly useful in **programming**.
- Functions are **first-class citizens**:
  - They can be passed as *arguments*;
  - They can be obtained as *results*.
- **Motivations**:
  - *Modularity*;
  - *Code reuse*;
  - *Conciseness*.
- **Example**:

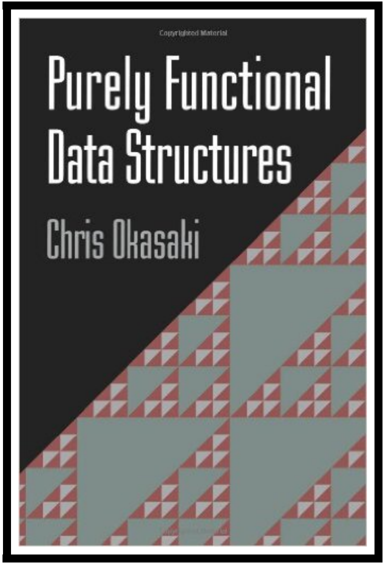
```
foldr              :: (a -> b -> b) -> b -> [a] -> b
foldr f acc []      = acc
foldr f acc (x:xs)  = f x (foldr f acc xs)
```

- **Models**:
  - λ-calculus

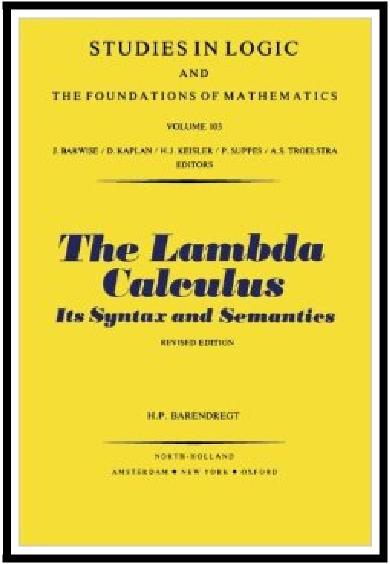
Higher-Order Computation



FUNCTIONAL
PROGRAMMING



FUNCTIONAL
DATA STRUCTURES



λ-CALCULUS

Does it Make Sense?

Does it Make Sense?

What Kind of Metatheory Does it Have?

Does it Make Sense?

What Kind of Metatheory Does it Have?

Interesting Research Problems?

This Tutorial

1. **Motivating Examples.**
2. **A λ-Calculus Foundation.**
3. **Operational and Denotational Semantics.**
4. **Termination and Resource Analysis.**

This Tutorial

1. **Motivating Examples.**
2. **A λ-Calculus Foundation.**
3. **Operational and Denotational Semantics.**
4. **Termination and Resource Analysis.**

A webpage: `http://www.cs.unibo.it/~dallago/HOPP/`

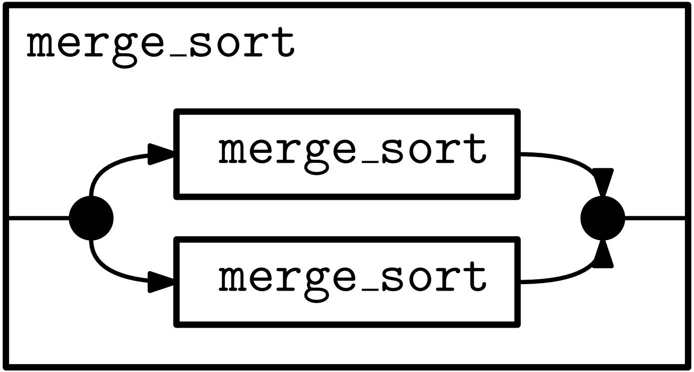
MergeSort (1)

```
let rec merge = function
      | list, []
      | [], list -> list
      | h1::t1, h2::t2 ->
          if h1 <= h2 then
            h1 :: merge (t1, h2::t2)
          else
            h2 :: merge (h1::t1, t2);;

let rec halve = function
      | []
      | [_] as t1 -> t1, []
      | h::t ->
          let t1, t2 = halve t in
            h::t2, t1;;
```

MergeSort (2)

```
let rec merge_sort = function
      | []
      | [_] as list -> list
      | list ->
          let l1, l2 = halve list in
            merge (merge_sort l1, merge_sort l2);;
```

The Structure of MergeSort



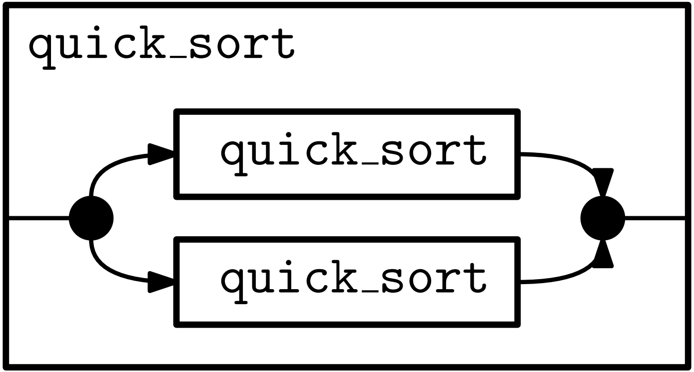
MergeSort, HO

```
let rec merge = function
    | (list, []), _ -> list
    | ([], list), _ -> list
    | (h1::t1, h2::t2),el ->
        if h1 <= h2 then
          h1 :: merge ((t1, h2::t2),el)
        else
          h2 :: merge ((h1::t1, t2),el);;

let rec halve = function
    | []
    | [_] as t1 -> (t1, []),()
    | h::t ->
        let (t1, t2),el = halve t in
          (h::t2, t1),el;;

let rec dac divide conquer = function
    | []
    | [_] as list -> list
    | list ->
        let (l1, l2),el = divide list in
          conquer ((dac divide conquer l1, dac divide conquer l2),el);;

let rec merge_sort = dac halve merge;;
```

QuickSort

```
let app x y z = x @ (z::y);;

let partition = function
    | pivot :: rest -> (List.partition (( > ) pivot) (rest)),pivot;;

let rec quick_sort = function
    | []
    | [_] as list -> list
    | list ->
      let (l1, l2), el = partition list in
        app (quick_sort l1) (quick_sort l2) el;;
```

The Structure of MergeSort



QuickSort, HO

```
let app = function
    | (x,y),z -> x @ (z::y);;

let partition = function
    | pivot :: rest -> (List.partition (( > ) pivot) (rest)),pivot;;

let rec dac divide conquer = function
    | []
    | [_] as list -> list
    | list ->
        let (l1, l2),el = divide list in
          conquer ((dac divide conquer l1, dac divide conquer l2),el);;

let quick_sort = dac partition app;;
```

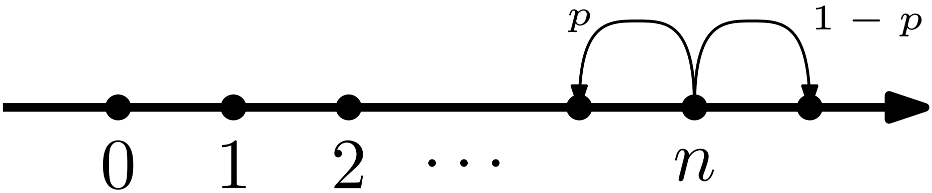
Randomized QuickSort (1)

```
let app = function
    | (x,y),z -> x @ (z::y);;

let rec extract = function
  | [],_ -> ([],0)
  | hd::tl,n ->
     if n==0 then
       (tl,hd)
     else
       let (l,el) = extract(tl,n-1) in
          (hd::l,el);;

let partition list =
  let (rest,pivot) = extract (list,(Random.int (List.length list))) in
    (List.partition (( > ) pivot) (rest)),pivot;;
```

Randomized QuickSort (2)

```
let rec dac divide conquer = function
      | []
      | [_] as list -> list
      | list ->
          let (l1, l2),el = divide list in
            conquer ((dac divide conquer l1, dac divide conquer l2),el);;

let rand_quick_sort = dac partition app;;
```

Random Walk



Two Kinds of Random Walks

```
let rec iter f g n = if n==0 then g else let m=pred(n) in f m (iter f g m);;

let mult m n = succ(m)*n;;

let fact = iter mult 1;;

let rec param_iter f g step n =
    if n==0 then g else let m=step(n) in f m (param_iter f g step m);;

let succ_2 m n = n+1;;

let updown_fair x = x+(2*Random.int(2)-1);;

let fair_random_walk = param_iter succ_2 0 updown_fair;;

let updown_biased x = if Random.int(3)==0 then x+1 else x-1;;

let biased_random_walk = param_iter succ_2 0 updown_biased;;
```

The Coupon Collector

- A brand distributes a large quantity of coupons, each labelled with $i \in \{1, \ldots, n\}$.

The Coupon Collector

- A brand distributes a large quantity of coupons, each labelled with $i \in \{1, \ldots, n\}$.
- Every day, you collect one coupon at a local store. Any label i has probability $\frac{1}{n}$ to occur.

The Coupon Collector

- A brand distributes a large quantity of coupons, each labelled with $i \in \{1, \ldots, n\}$.
- Every day, you collect one coupon at a local store. Any label i has probability $\frac{1}{n}$ to occur.
- You win a prize when you collect a set of n coupons, each with a distinct label.

The Coupon Collector

- A brand distributes a large quantity of coupons, each labelled with $i \in \{1, \ldots, n\}$.
- Every day, you collect one coupon at a local store. Any label i has probability $\frac{1}{n}$ to occur.
- You win a prize when you collect a set of n coupons, each with a distinct label.
- **Example**: if $n = 5$, you could get the following coupons:

$$3, 1, 5, 2, 3, 1, 5, 2, 3, 1, 5, 2, \ldots$$

The Coupon Collector

- A brand distributes a large quantity of coupons, each labelled with $i \in \{1, \ldots, n\}$.
- Every day, you collect one coupon at a local store. Any label i has probability $\frac{1}{n}$ to occur.
- You win a prize when you collect a set of n coupons, each with a distinct label.
- **Example**: if $n = 5$, you could get the following coupons:

$$3, 1, 5, 2, 3, 1, 5, 2, 3, 1, 5, 2, \ldots$$

- Are you guaranteed to win the prize with probability 1? After how many days, on the average?

The Coupon Collector

```
let rec base_param_iter f g step base e =
  if base(e) then g else let d=step(e) in f d (base_param_iter f g step base d);;

let second_zero = function
  | (_,0) -> true
  | _ -> false;;

let succ_2 m n = n+1;;

let step_2 = function
  | (n,m) -> if Random.int(n)<=m then (n,m-1) else (n,m);;

let coupon_collector x = base_param_iter succ_2 0 step_2 second_zero (x,x);;
```

Beyond Randomized Algorithms: The Grass Problem

- You get up in the morning, and notice that the grass in your garden is wet.

Beyond Randomized Algorithms: The Grass Problem

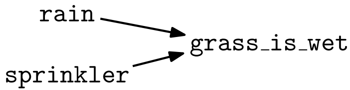
- You get up in the morning, and notice that the grass in your garden is wet.
- You know that there is 30% probability that it rains during the night.

Beyond Randomized Algorithms: The Grass Problem

- You get up in the morning, and notice that the grass in your garden is wet.
- You know that there is 30% probability that it rains during the night.
- The sprinkler is activated once every two days.

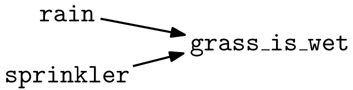
Beyond Randomized Algorithms: The Grass Problem

- You get up in the morning, and notice that the grass in your garden is wet.
- You know that there is 30% probability that it rains during the night.
- The sprinkler is activated once every two days.
- You further know that:
  - *If it rains*, there is 90% probability that the grass is wet the following morning.
  - *If the sprinkler is activated*, there is 80% probability that the grass is wet the following morning.
  - *In any other case*, there is anyway a 10% probability that the grass is wet.
- In other words, you are in a situation like the following:



Beyond Randomized Algorithms: The Grass Problem

- You get up in the morning, and notice that the grass in your garden is wet.
- You know that there is 30% probability that it rains during the night.
- The sprinkler is activated once every two days.
- You further know that:
  - *If it rains*, there is 90% probability that the grass is wet the following morning.
  - *If the sprinkler is activated*, there is 80% probability that the grass is wet the following morning.
  - *In any other case*, there is anyway a 10% probability that the grass is wet.
- In other words, you are in a situation like the following:



- Now: how likely it is that it rained?

The Grass Model

```
let grass_model () =    (* unit -> bool *)
  let rain = flip 0.3 and sprinkler = flip 0.5 in
  let grass_is_wet =
    (flip 0.9 && rain) || (flip 0.8 && sprinkler) || flip 0.1 in
  if not grass_is_wet then fail ();
  rain
```

Thank You!

Questions?