INDUCING PROBABILISTIC PROGRAMS BY BAYESIAN PROGRAM MERGING

Paper by: **Irvin Hwang, Andreas Stuhlmuller, Noah Goodman**

Today's presenter: **Yura Perov**

[**Paper reference:** http://arxiv.org/abs/1110.5667]

Learn generative models from data

$$f(X) = Y$$

$$f(X) = Y$$

$$P(Y \mid X)$$

$$f(X) = Y$$

$$T \sim P(Y \mid X)$$

$$T \sim P(Y \mid X)$$

T ~ P(Y)

```
[ASSUME marsaglia-std-normal
  (lambda ()
    (let [x (uniform-continuous -1.0 1.0)
          y (uniform-continuous -1.0 1.0)
          s (+ (* x x) (* y y))]
      (if (< s 1)
        (* x (sqrt (* -2 (/ (log s) s ))))
        (marsaglia-std-normal))))]
```

$$T \sim P(Y)$$

T ~ P(Y)

```
[ASSUME create-ai (production-rules …)]
[ASSUME ai (create-ai)]
[OBSERVE (ai task1-input) task1-output]
          …
[OBSERVE (ai task10^20-input) task10^20-output]
[PREDICT (ai my-task)]
```

```
(define (bubble-sort x gt?)
  (letrec
    ((fix (lambda (f i)
       (if (equal? i (f i))
           i
           (fix f (f i)))))

     (sort-step (lambda (l)
        (if (or (null? l) (null? (cdr l)))
            l
            (if (gt? (car l) (cadr l))
                (cons (cadr l) (sort-step (cons (car l) (cddr l))))
                (cons (car  l) (sort-step (cdr l))))))))

  (fix sort-step x)))
```

O(n^2)

```
(define (split-by l p k)
  (let loop ((low '())
             (high '())
             (l l))
    (cond ((null? l)
           (k low high))
          ((p (car l))
           (loop low (cons (car l) high) (cdr l)))
          (else
           (loop (cons (car l) low) high (cdr l))))))

(define (quicksort l gt?)
  (if (null? l)
      '()
      (split-by (cdr l)
                (lambda (x) (gt? x (car l)))
                (lambda (low high)
                  (append (quicksort low gt?)
                          (list (car l))
                          (quicksort high gt?))))))

(quicksort '(1 3 5 7 9 8 6 4 2) >)
```

O(n log n)

How to find T given $\hat{Y}$?

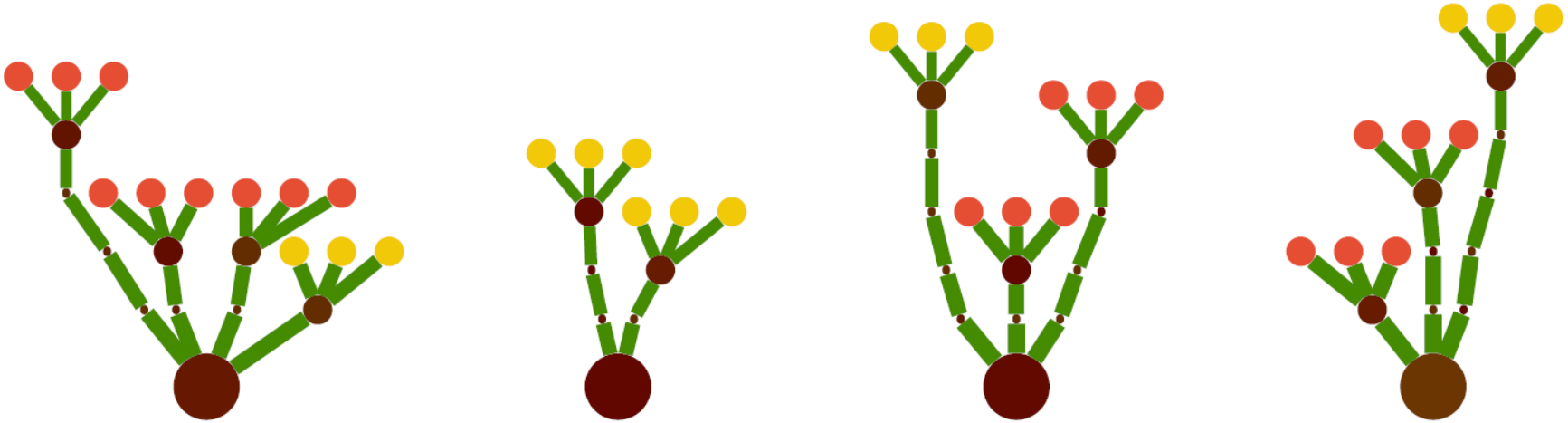
How to find T given $\hat{Y}$?

$$P(T \mid \hat{Y})$$

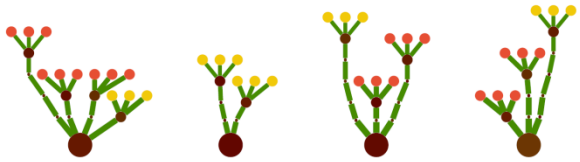
How to find T given $\hat{Y}$?

$$P(T \mid \hat{Y}) = \frac{P(T)P(\hat{Y} \mid T)}{P(\hat{Y})}$$

$\hat{Y}$ in our case …



$\widehat{Y}$ in our case ...



$\langle tree \rangle$ ::= nil

$\langle tree \rangle$ ::= (node $\langle data \rangle$ $\langle tree \rangle$ $\langle tree \rangle$ ...)

$\langle data \rangle$ ::= (data $\langle color \rangle$ $\langle size \rangle$)

$\langle color \rangle$ ::= (color [number])

$\langle size \rangle$ ::= (size [number])

Example of T

```
(define (tree)
  (uniform-choice
   (node (body) (branch))
   (node (body) (branch) (branch))
   (node (body) (branch) (branch) (branch))
   (node (body) (branch) (branch) (branch) (branch))))

(define (body)
  (data (color (gaussian 30 10)) (size .7)))

(define (branch)
  (if (flip .2)
      (flower (if (flip .5) 150 255))
      (node (branch-info) (branch))))

(define (branch-info)
  (data (color (gaussian 0 25)) (size .1)))

(define (flower shade)
  (node (data (color (gaussian 0 25)) (size .3))
        (petal shade)
        (petal shade)
        (petal shade)))

(define (petal shade)
  (node (data (color shade) (size .3))))
```

Bayesian Model Merging

Bayesian Model Merging
$(Program)$

Bayesian Model Merging
(*Program*)

Data

Model

Bayesian Model Merging
$(Program)$

Data

Model

1. Data incorporation

$$T_0 = f(\hat{Y})$$

Bayesian Model Merging
$(Program)$

Data

Model

1. Data incorporation
$$T_0 = f(\hat{Y})$$

2. Model merging
$$T_1 = m(T_0), T_2 = m(T_1), \ldots$$

Bayesian Model Merging
$(Program)$

Data

Model

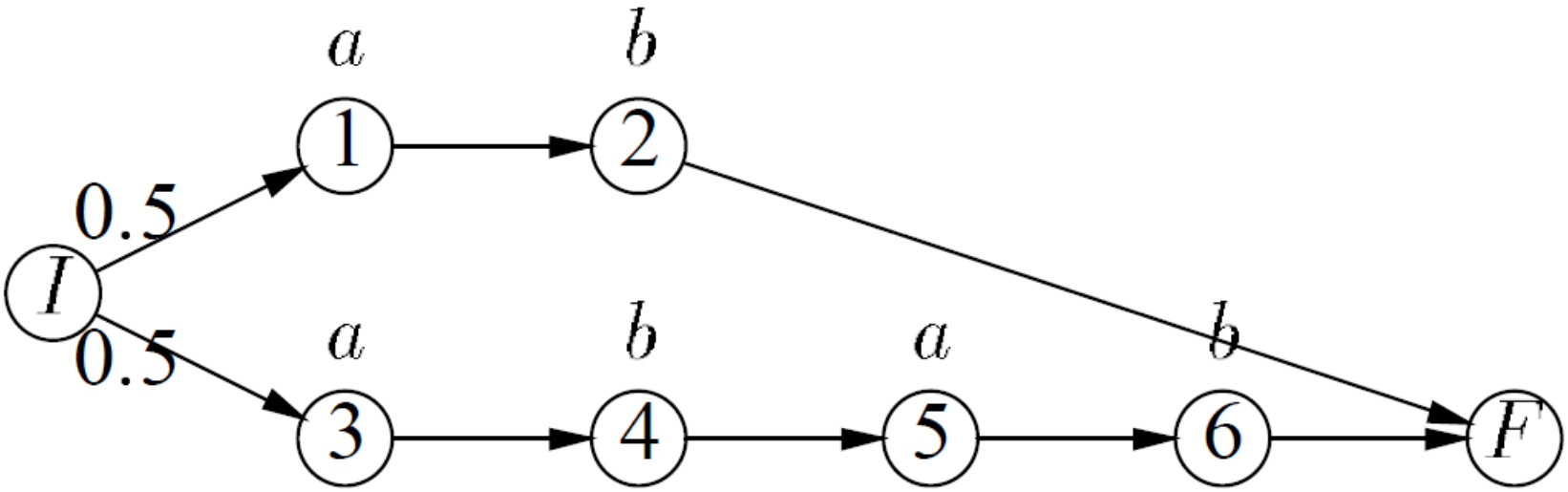
1. Data incorporation
$$T_0 = f(\hat{Y})$$

2. Model merging
$$T_1 = m(T_0), T_2 = m(T_1), \ldots$$

3. $$\mathrm{P}(T_i | \hat{\mathrm{Y}}) = \frac{\mathrm{P}(T_i)\mathrm{P}(\hat{\mathrm{Y}} | T_i)}{\mathrm{P}(\hat{\mathrm{Y}})}$$

Example of merging Markov Model

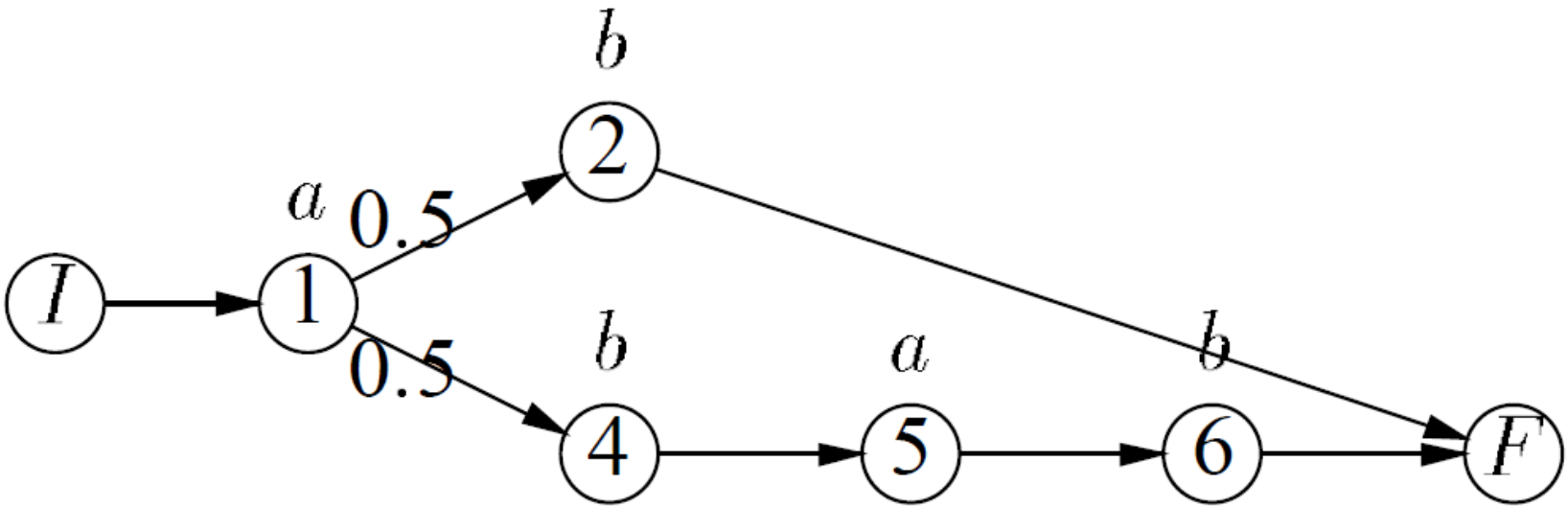
$$\hat{Y} = \{ab, abab\}$$



These images are from Stolcke, Andreas, and Stephen Omohundro. "Inducing probabilistic grammars by Bayesian model merging." Grammatical inference and applications. Springer Berlin Heidelberg, 1994. 106-118.

Example of merging Markov Model

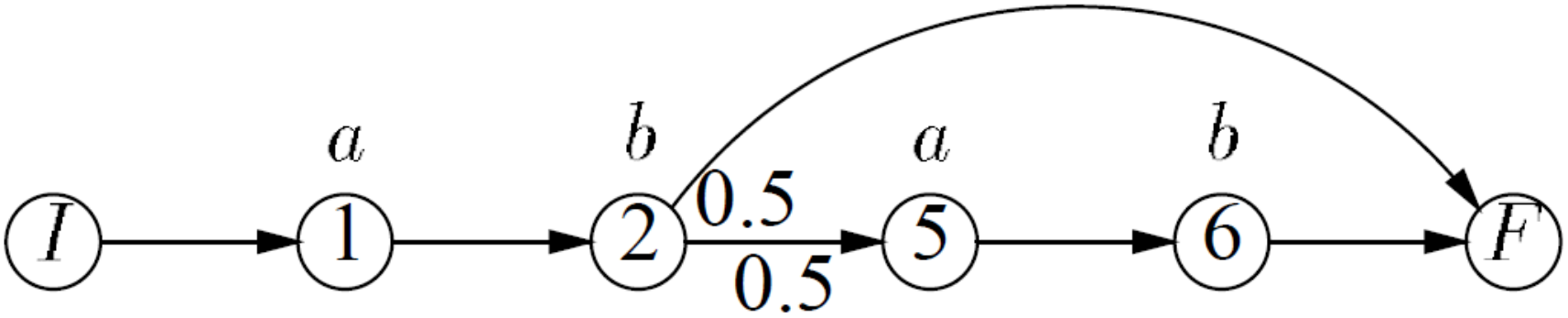
$$\hat{Y} = \{ab, abab\}$$



These images are from Stolcke, Andreas, and Stephen Omohundro. "Inducing probabilistic grammars by Bayesian model merging." Grammatical inference and applications. Springer Berlin Heidelberg, 1994. 106-118.

Example of merging Markov Model

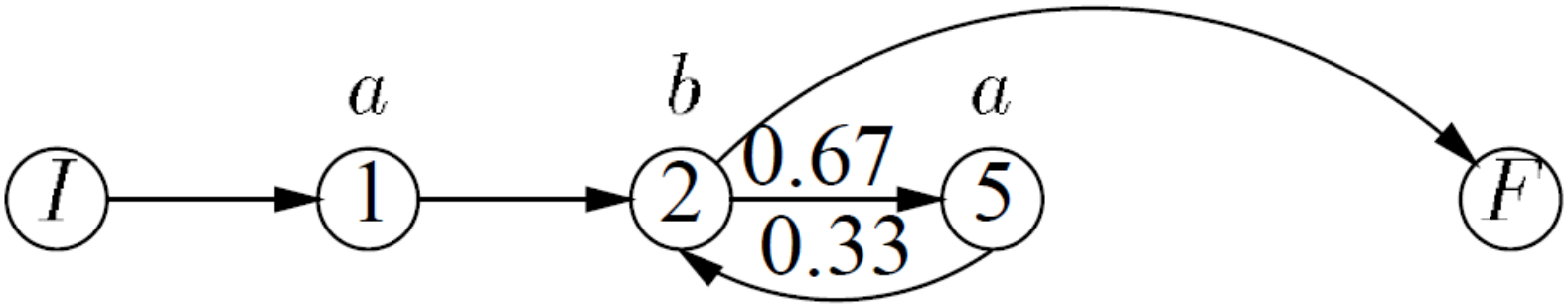
$$\hat{Y} = \{ab, abab\}$$



These images are from Stolcke, Andreas, and Stephen Omohundro. "Inducing probabilistic grammars by Bayesian model merging." Grammatical inference and applications. Springer Berlin Heidelberg, 1994. 106-118.

Example of merging Markov Model

$$\hat{Y} = \{ab, abab\}$$



These images are from Stolcke, Andreas, and Stephen Omohundro. "Inducing probabilistic grammars by Bayesian model merging." Grammatical inference and applications. Springer Berlin Heidelberg, 1994. 106-118.

Bayesian Program Merging

Data

Model

1. Data incorporation

$$T_0 = f(\hat{Y})$$

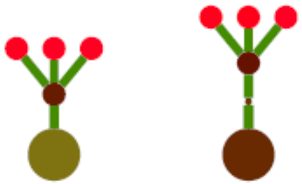
2. Transformations

$$T' = m(T)$$

3. $$\mathrm{P}(T_i | \hat{\mathrm{Y}}) = \frac{\mathrm{P}(T_i)\mathrm{P}(\hat{\mathrm{Y}} | T_i)}{\mathrm{P}(\hat{\mathrm{Y}})}$$

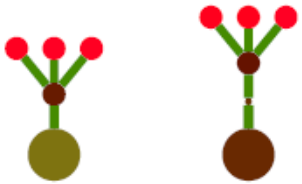
Data Incorporation

```
(node (data (color 70) (size 0.7))
      (node (data (color 37) (size 0.3))
            (node (data (color 213) (size 0.3)))
            (node (data (color 207) (size 0.3)))
            (node (data (color 211) (size 0.3)))))
(node (data (color 43) (size 0.7))
      (node (data (color 47) (size 0.1))
            (node (data (color 33) (size 0.3))
                  (node (data (color 220) (size 0.3)))
                  (node (data (color 224) (size 0.3)))
                  (node (data (color 207) (size 0.3))))))
```



Data Incorporation

```
(node (data (color 70) (size 0.7))
      (node (data (color 37) (size 0.3))
            (node (data (color 213) (size 0.3)))
            (node (data (color 207) (size 0.3)))
            (node (data (color 211) (size 0.3)))))
(node (data (color 43) (size 0.7))
      (node (data (color 47) (size 0.1))
            (node (data (color 33) (size 0.3))
                  (node (data (color 220) (size 0.3)))
                  (node (data (color 224) (size 0.3)))
                  (node (data (color 207) (size 0.3))))))
```



```
(λ ()
  (uniform-choice
   (node (data (color (gaussian 70 25)) (size .7))
         (node (data (color (gaussian 37 25)) (size 0.3))
               (node (data (color (gaussian 213 25)) (size 0.3)))
               (node (data (color (gaussian 207 25)) (size 0.3)))
               (node (data (color (gaussian 211 25)) (size 0.3)))))
   (node (data (color (gaussian  43)) (size .7))
         (node (data (color (gaussian 47 25)) (size 0.1))
               (node (data (color (gaussian 33 25)) (size 0.3))
                     (node (data (color (gaussian 220 25)) (size 0.3)))
                     (node (data (color (gaussian 224 25)) (size 0.3)))
                     (node (data (color (gaussian 207 25)) (size 0.3))))))))
```

Program transformations

1. **Abstraction**
   - Find a pattern in sub-programs, represent this pattern by a new function, and insert this new function to the code.

2. **Deargumentation**
   - Decrease number of arguments in sub-functions.

Abstraction

```
(uniform-choice
 (node a (node a (node b) (node b)))
 (node a (node a (node c) (node c))))
```

```
(begin
  (define (F1 V1 V2)
    (node a (node a (node V1) (node V2))))
  (uniform-choice (F1 b b) (F1 c c)))
```

Abstraction

For this, to the best of my knowledge, they:

1. Extract **all** sub-expressions from the program body (including already extracted abstractions in the form of compound procedures). **(This is different from what we presumed during the reading group session.)**

2. Try to extract a pattern from *each* pair of sub-expressions they extracted.
    - E.g., from (+ 1 2) and (+ 1 3) we can extract a pattern (lambda (x) (+ 1 x))

3. For each extracted pattern they represent it as a new compound procedure, and insert anywhere possible.

Abstraction

If they indeed consider all possible sub-expressions, this is computationally expensive. I have not found in their report a description how they deal with this complexity.

Obviously, for short programs this is doable.

An idea: for complex long programs somebody can consider a random subset of sub-expressions.

Abstraction

N.B. Since they consider all sub-expressions,

for the sub-expression

```
(node (node a))
```

there is exist a derivable pattern

```
(define f1 (lambda (x) (node x)))
```

so after unification (pattern insertion) we have

```
(f1 (f1 a))
```

Deargumentation

“Deargumentation is a program transformation that takes a function F in a program and changes its definition by removing one of the function arguments.”

In the paper they consider a few situations where we wish to remove one of the function arguments.

Deargumentation.
1. Compactly representing noisy data

Remove
an argument
entirely, replacing
all its values by
some distribution.

```
(begin
  (define flower
    (λ  V1  V2  V3  V4)
      (node (data (color (gaussian V1 25)) (size 0.3))
            (node (data (color (gaussian V2 25)) (size 0.3)))
            (node (data (color (gaussian V3 25)) (size 0.3)))
            (node (data (color (gaussian V4 25)) (size 0.3))))))
  (uniform-choice
   (flower 200 213 207 211)
   (flower 33 220 224 207)))
```

```
(begin
  (define flower
    (λ  (V1  V3  V4)
      ((λ  (V2)
         (node (data (color (gaussian V1 25)) (size 0.3))
               (node (data (color (gaussian V2 25)) (size 0.3)))
               (node (data (color (gaussian V3 25)) (size 0.3)))
               (node (data (color (gaussian V4 25)) (size 0.3)))))
       216.5))))
  (uniform-choice
   (flower 200 207 211)
   (flower 33 224 207)))
```

Deargumentation.
1. Compactly representing noisy data

To do this, we just need to enumerate all arguments of all functions:

1. For each argument we should check that it has the same type signature,
   - like `real` in `(lambda (x) (gaussian x 25))`,
   - or like `(node real)` in `(lambda (x) (node x))`.

   Checking that the argument has the same type requires some static analysis, which they most probably use but do not mention.

   Also N.B. that they pre-proccesed programs by replacing all reals by samples from Gaussian.

2. Once they found an argument, which takes the same type, they replace this argument values by the mean of all previous values of this argument.

Deargumentation.
1. Compactly representing noisy data

Once they found the argument, which takes the same type, they replace this argument values by the mean of all values of this argument.

E.g.

MEAN(213, 220) = 216.5

```
(begin
  (define flower
    (λ (V1 V2 V3 V4)
      (node (data (color (gaussian V1 25)) (size 0.3))
            (node (data (color (gaussian V2 25)) (size 0.3)))
            (node (data (color (gaussian V3 25)) (size 0.3)))
            (node (data (color (gaussian V4 25)) (size 0.3))))))
  (uniform-choice
   (flower 200 213 207 211)
   (flower 33 220 224 207)))
```

```
(begin
  (define flower
    (λ (V1 V3 V4)
      ((λ (V2)
         (node (data (color (gaussian V1 25)) (size 0.3))
               (node (data (color (gaussian V2 25)) (size 0.3)))
               (node (data (color (gaussian V3 25)) (size 0.3)))
               (node (data (color (gaussian V4 25)) (size 0.3)))))
       216.5)))
  (uniform-choice
   (flower 200 207 211)
   (flower 33 224 207)))
```

Deargumentation.
2. Merging similar variables

If there are two arguments with the same type, the transformation just replace one argument's values by values of another.

```
(begin
  (define flower
    (λ (V1 V2 V3 V4)
      (node (data (color (gaussian V1 25)) (size 0.3))
            (node (data (color (gaussian V2 25)) (size 0.3)))
            (node (data (color (gaussian V3 25)) (size 0.3)))
            (node (data (color (gaussian V4 25)) (size 0.3))))))
  (uniform-choice
   (flower 200 213 207 211)
   (flower 33 220 224 207))
```

```
(begin
  (define flower
    (λ (V1 V3 V4)
      ((λ (V2)
         (node (data (color (gaussian V1 25)) (size 0.3))
               (node (data (color (gaussian V2 25)) (size 0.3)))
               (node (data (color (gaussian V3 25)) (size 0.3)))
               (node (data (color (gaussian V4 25)) (size 0.3)))))
       V3)))
(uniform-choice
 (flower 200 207 211)
 (flower 33 224 207)))
```

Deargumentation.
3. Inducing recursive functions

If “arguments expressions” to some function F calls contains calls to this function itself…

```
(F1 (F1 a))
```

we calculate probabilities (frequencies) of all possible arguments expressions…

`a` — 1 appearance => 0.5 probability
`(F1 a)` = `(recur a)` — 1 appearance => 0.5 probability

Deargumentation.
3. Inducing recursive functions

… and use this frequencies as probabilities for the decision inside renewed F:

```
(begin
  (define (F1 x)
    ((λ (x)
       (node x))
     (if (flip .5)
         (F1 a)
         a)))
  (F1 (F1 a)))
```

… and we also can get rid of the argument, where recursion call was made:

```
(begin
  (define (F1)
    ((λ (x)
       (node x))
     (if (flip .5)
         (F1)
         a)))
  (F1))
```

Deargumentation.
4. Inducing noisy data constructors

To be honest, I do not understand why do we need this subsection.

If I understand correctly, the transformations from previous sections already do the transformation described in this subsection. Maybe in this subsection they describe a less expensive method to do the same thing.

I am going to email authors about this.

Beam search

We already discussed on our reading group the way they use beam search.

However, I do not understand how do they separate transformations from each other.

Each transformation described in the paper may have a few outputs (e.g. we can deargument this or that argument). Most probably, this means that when they do beam search, they consider each output of transformations as a separate transformation, and by this they create a graph of possible transformations they are making beam search on.

I am going to email authors with a several questions, and if anybody is interested, I can email you afterwards.

Thank you for attending the reading group last week!