Nested Queries in Nondeterministic and Probabilistic Programming Languages

Jeffrey Mark Siskind, qobi@purdue.edu

NEPLS, Tuesday 10 November 2015

Joint work with Frank Wood and Hongseok Yang
?- member(X, [a, b, c]).
   X = a

Prolog

?- member(X, [a, b, c]).
X = a ;
X = b

?- member(X, [a, b, c]).
X = a ;
X = b ;
X = c.
?- 

?- member(X, [a, b, c]).
X = a ;
X = b ;
X = c.
?- bagof(X, member(X, [a, b, c]), Xs).
Xs = [a, b, c].
?-


> (define (choose l)
  (if (null? l) (fail) (amb (first l) (choose (rest l)))))
> (choose '(a b c))
a
> 

> (define (choose l)
    (if (null? l) (fail) (amb (first l) (choose (rest l)))))
> (choose '(a b c))
a
> (fail)
b
>
(define (choose l)
  (if (null? l) (fail) (amb (first l) (choose (rest l))))))

(choose '(a b c))
a
(fail)
b
(fail)
c

> (define (choose l)
    (if (null? l) (fail) (amb (first l) (choose (rest l))))
> (choose '(a b c))
a
> (fail)
b
> (fail)
c
> (fail)
>>Error: Top-level fail
>

(define (choose l)
  (if (null? l) (fail) (amb (first l) (choose (rest l)))))

(choose '(a b c))
a
(b)
(c)

>Error: Top-level fail

(set (choose '(a b c)))
(a b c)


Stochastic Lisp/Probabilistic Scheme/Church

> (sample '((a . 0.5) (b . 0.25) (c . 0.25)))
a
Probability is: 0.5
>
> (sample ’((a . 0.5) (b . 0.25) (c . 0.25)))
a
Probability is: 0.5
> (fail)
b
Probability is: 0.25
>
> (sample '((a . 0.5) (b . 0.25) (c . 0.25)))
  a
  Probability is: 0.5
> (fail)
  b
  Probability is: 0.25
> (fail)
  c
  Probability is: 0.25
> 

> (sample '((a . 0.5) (b . 0.25) (c . 0.25)))

a
Probability is: 0.5

> (fail)
b
Probability is: 0.25

> (fail)
c
Probability is: 0.25

> (fail)

>>Error: Top-level fail

> (sample '((a . 0.5) (b . 0.25) (c . 0.25)))
a
Probability is: 0.5
> (fail)
b
Probability is: 0.25
> (fail)
c
Probability is: 0.25
> (fail)
>>Error: Top-level fail
> (distribution (sample '((a . 0.5) (b . 0.25) (c . 0.25)))
((a . 0.5) (b . 0.25) (c . 0.25))
>

Dichotomy

choose turns a set into a nondeterministic process

distribution turns a stochastic process into a distribution

(set (choose e)) = e

(set (choose e)) = e when e is deterministic

(distribution (sample e)) = e

(distribution (sample e)) = e when e is deterministic
**Dichotomy**

`choose` turns a set into a nondeterministic process.
choose turns a set into a nondeterministic process
set turns a nondeterministic process into a set
choose turns a set into a nondeterministic process
set turns a nondeterministic process into a set
sample turns a distribution into a stochastic process
**Dichotomy**

- `choose` turns a set into a nondeterministic process
- `set` turns a nondeterministic process into a set
- `sample` turns a distribution into a stochastic process
- `distribution` turns a stochastic process into a distribution
Dichotomy

- **choose** turns a set into a nondeterministic process
- **set** turns a nondeterministic process into a set
- **sample** turns a distribution into a stochastic process
- **distribution** turns a stochastic process into a distribution

\[
\begin{align*}
(\text{choose } (\text{set } e)) &= e \\
(\text{set } (\text{choose } e)) &= e \\
(\text{sample } (\text{distribution } e)) &= e \\
(\text{distribution } (\text{sample } e)) &= e
\end{align*}
\]

when \( e \) is deterministic
(define (nondeterministic-boolean) (error #f "Top-level nondeterministic-boolean"))

(define (nondeterministic-fail) (error #f "Top-level nondeterministic-fail"))

(define (choose x)
  (let loop ((x x) (p 1))
    (cond ((or (null? x) (zero? p)) (nondeterministic-fail))
          ((nondeterministic-boolean (/ (cdr (first x)) p)) (first x))
          (else (loop (rest x) (- p (cdr (first x)))))))


(define (probabilistic-boolean alpha) (error #f "Top-level probabilistic-boolean"))

(define (probabilistic-fail) (error #f "Top-level probabilistic-fail"))

(define (sample x)
  (let loop ((x x) (p 1))
    (cond ((or (null? x) (zero? p)) (probabilistic-fail))
      ((probabilistic-boolean (/ (cdr (first x)) p)) (car (first x)))
      (else (loop (rest x) (- p (cdr (first x)))))))))


(define-syntax set
  (syntax-rules ()
    ((set e)
      (call-with-current-continuation
       (lambda (c)
         (let ((values '())
               (saved-nondeterministic-boolean nondeterministic-boolean)
               (saved-nondeterministic-fail nondeterministic-fail)
             )
           (set! nondeterministic-boolean
                (lambda ()
                  (call-with-current-continuation
                   (lambda (c)
                     (let ((saved-nondeterministic-fail nondeterministic-fail)
                           )
                       (set! nondeterministic-fail
                            (lambda ()
                              (set! nondeterministic-fail saved-nondeterministic-fail)
                              (c #f)))))
                  (c #f)))))
               (t))))
    (set! nondeterministic-fail
         (lambda ()
           (set! nondeterministic-boolean saved-nondeterministic-boolean)
           (set! nondeterministic-fail saved-nondeterministic-fail)
           (c (reverse values))))
    (set! values (cons e values))
    (nondeterministic-fail)))))
(define-syntax distribution
  (syntax-rules ()
    ((distribution e)
     (call-with-current-continuation
      (lambda (c)
        (let ((values '())
              (saved-probabilistic-boolean probabilistic-boolean)
              (saved-probabilistic-fail probabilistic-fail)
              (p 1))
          (set! probabilistic-boolean
            (lambda (alpha)
              (unless (<= 0 alpha 1) (error #f "Alpha not probability"))
              (call-with-current-continuation
               (lambda (c)
                 (let ((saved-probabilistic-fail probabilistic-fail)
                       (saved-p p))
                   (set! probabilistic-fail
                     (lambda ()
                       (set! probabilistic-fail saved-probabilistic-fail)
                       (set! p (* (- 1 alpha) saved-p))
                       (c #f))
                     (set! p (* alpha p))
                     (#t))))))
          (set! values (cons (cons e p) values))
          (probabilistic-fail))))))
Utility of a Query or Aggregation Construct

- satisfiability
  
  \( \text{not (null? (set e))} \)

- model counting
  
  \( \text{length (set e)} \)

- optimization
  
  \( \text{fold max minus-infinity (map f (set e))} \)

Jeffrey Mark Siskind (Purdue)  Nested Nondeterministic and Stochastic Queries  NEPLS, 10 November 2015  8 / 31
satisfiability

\[(\text{not} \ (\text{null?} \ (\text{set} \ e)))\]
Utility of a Query or Aggregation Construct

- **satisfiability**
  
  \((\text{not } (\text{null? } (\text{set } e)))\)

- **model counting**
  
  \((\text{length } (\text{set } e))\)
Utility of a Query or Aggregation Construct

- **satisfiability**
  
  \[\text{not (null? (set e))}\]

- **model counting**
  
  \[\text{length (set e)}\]

- **optimization**
  
  \[\text{fold max minus-infinity (map f (set e))}\]
Utility of a Query or Aggregation Construct

- Marginal probability
  \[ \text{fold} + 0 \left( \text{map cdr (remove-if-not car (distribution e))} \right) \]

- Expectation
  \[ \text{fold} + 0 \left( \text{map (lambda (pair) (* (car pair) (cdr pair))) (distribution e)} \right) \]

- Maximum likelihood
  \[ \text{fold max minus-infinity (map cdr (distribution e))} \]

- Entropy
  \[ \text{fold} + 0 \left( \text{map (lambda (pair) (* (cdr pair) (log (cdr pair)))) (distribution e)} \right) \]

- Mutual information

- KL divergence
Utility of a Query or Aggregation Construct

- marginal probability

\[(\text{fold } + 0 \ (\text{map } \text{cdr} \ (\text{remove-if-not car} \ (\text{distribution } e)))\)\]
Utility of a Query or Aggregation Construct

- **marginal probability**

  \[
  \text{(fold } + 0 \text{ (map cdr (remove-if-not car (distribution e))))}
  \]

- **expectation**

  \[
  \text{(fold } + 0 \text{ (map (lambda (pair) (* (car pair) (cdr pair))) (distribution e)))}
  \]
Utility of a Query or Aggregation Construct

» marginal probability

\[(\text{fold } + 0 \ (\text{map } \text{cdr} \ (\text{remove-if-not car} \ (\text{distribution } e))))\]

» expectation

\[(\text{fold } + 0 \ (\text{map } (\lambda \ (\text{pair}) \ (* \ (\text{car} \ \text{pair}) \ (\text{cdr} \ \text{pair}))) \ (\text{distribution } e))))\]

» maximum likelihood

\[(\text{fold } \text{max} \ \text{minus-infinity} \ (\text{map } \text{cdr} \ (\text{distribution } e))))\]
Utility of a Query or Aggregation Construct

- **marginal probability**
  \[
  \text{fold} + 0 \ (\text{map} \ \text{cdr} \ \text{remove-if-not} \ \text{car} \ (\text{distribution} \ e))
  \]

- **expectation**
  \[
  \text{fold} + \\
  0 \\
  \ \text{map} \ \text{lambda} \ (\text{pair}) \ (* \ (\text{car} \ \text{pair}) \ (\text{cdr} \ \text{pair})) \\
  \ \text{(distribution} \ e))
  \]

- **maximum likelihood**
  \[
  \text{fold} \ \text{max} \ \text{minus-infinity} \ \text{map} \ \text{cdr} \ \text{distribution} \ e)
  \]

- **entropy**
  \[
  \text{fold} + \\
  0 \\
  \ \text{map} \ \text{lambda} \ (\text{pair}) \ (* \ (\text{cdr} \ \text{pair}) \ (\text{log} \ (\text{cdr} \ \text{pair}))) \\
  \ \text{(distribution} \ e))
  \]
Utility of a Query or Aggregation Construct

- **marginal probability**
  
  \[(\text{fold} + 0 (\text{map} \ \text{cdr} \ (\text{remove-if-not} \ \text{car} \ (\text{distribution} \ e))))\]

- **expectation**
  
  \[(\text{fold} + 0 \ (\text{map} \ (\lambda \ (\text{pair}) \ (* \ (\text{car} \ \text{pair}) \ (\text{cdr} \ \text{pair})))) \ (\text{distribution} \ e)))\]

- **maximum likelihood**
  
  \[(\text{fold} \ \text{max} \ \text{minus-infinity} \ (\text{map} \ \text{cdr} \ (\text{distribution} \ e)))\]

- **entropy**
  
  \[(\text{fold} + 0 \ (\text{map} \ (\lambda \ (\text{pair}) \ (* \ (\text{cdr} \ \text{pair}) \ (\log \ (\text{cdr} \ \text{pair})))) \ (\text{distribution} \ e)))\]

- **mutual information**

- **KL divergence**
Notice that this also makes it quite all right for set expressions to be nested. For example one possible way to express the query: "Which people drink each beverage?"

is:

?- setof(Beverage-People, setof(X, X drinks Beverage, People), S).

Warren (1982)
(set (let ((beverage (choose beverages)))
    (cons beverage
      (set (let ((person (choose people)))
        (if (drinks? person beverage)
          person
          (nondeterministic-fail)))))))
(probability
  (let ( ((my-move (sample my-moves)))
    (> (probability
        (let ( ((your-move (sample your-moves)))
          (you-win? my-move your-move)))
        0.5)))

What is the probability that the probability that you will win is greater than 0.5?
(possible?
  (let ((my-move (choose my-moves)))
    (> (probability
        (let ((your-move (sample your-moves)))
          (you-win? my-move your-move))
        0.5)))

Is it possible that the probability that you will win is greater than 0.5?
Nesting Query or Aggregation Constructs

(probability
  (let ((my-move (sample my-moves)))
    (possible?
      (let ((your-move (choose your-moves)))
        (you-win? my-move your-move)))))

What is the probability that it is possible for you to win?
Nesting Query or Aggregation Constructs

let ((my-move (choose my-moves)))
> (probability
let ((your-move (sample your-moves)))
you-win? my-move your-move)
0.5)

probability
let ((my-move (sample my-moves)))
possible?
let ((your-move (choose your-moves)))
you-win? my-move your-move)

possible?
> (probability (you-win? (choose my-moves) (sample your-moves)))
you-win? my-move your-move)

possible?
you-win? (playing-a-game)
possible-it-is-probable? (you-win? (playing-a-game))
(possible?
 (let ((my-move (choose my-moves)))
 (> (probability
     (let ((your-move (sample your-moves)))
       (you-win? my-move your-move)))
    0.5)))

(probability
 (let ((my-move (sample my-moves)))
  (possible?
   (let ((your-move (choose your-moves)))
     (you-win? my-move your-move)))))
(possible?
  (let ((my-move (choose my-moves)))
    (> (probability
        (let ((your-move (sample your-moves)))
          (you-win? my-move your-move))
        0.5)))

(probability
  (let ((my-move (sample my-moves)))
    (possible?
      (let ((your-move (choose your-moves)))
        (you-win? my-move your-move))))

(possible?
  (> (probability (you-win? (choose my-moves) (sample your-moves)))
    0.5))

(probability
  (possible? (you-win? (sample my-moves) (choose your-moves))))
Nesting Query or Aggregation Constructs

(possible?
  (let ((my-move (choose my-moves)))
    (> (probability
        (let ((your-move (sample your-moves)))
          (you-win? my-move your-move))
      0.5)))

(probability
  (let ((my-move (sample my-moves)))
    (possible?
      (let ((your-move (choose your-moves)))
        (you-win? my-move your-move))))))

(possible?
  (> (probability (you-win? (choose my-moves) (sample your-moves)))
    0.5))

(probability
  (possible? (you-win? (sample my-moves) (choose your-moves))))

(possible-that-it-is-probable? (you-win? (playing-a-game)))

(probability-that-it-is-possible (you-win? (playing-a-game)))
The Rub

(choose '(#t #f))

((#t #f))

(distribution (distribution (sample '((#t . 0.5) (#f . 0.5)))))

((((#t . 0.5) (#f . 0.5)) . 1))

(set (distribution (sample '((#t . 0.5) (#f . 0.5)))))

(((#t . 0.5) (#f . 0.5)))

(set (distribution (choose '(#t #f))))

Unhandled exception

Condition components:
1. &error
2. &message: "Top-level probabilistic-fail"
3. &irritants: ()

(set (distribution (sample '((#t . 0.5) (#f . 0.5)))))

Unhandled exception

Condition components:
1. &error
2. &message: "Top-level nondeterministic-fail"
3. &irritants: ()
> (set (set (choose '(#t #f))))

((#t #f))
> (set (set (choose '(#t #f))))
  ((#t #f))
> (distribution (distribution (sample '((#t . 0.5) (#f . 0.5)))))
  (((#t . 0.5) (#f . 0.5)) . 1))

> (distribution (set (sample '((#t . 0.5) (#f . 0.5))))
Unhandled exception
Condition components:
1. &error
2. &message: "Top-level probabilistic-fail"
3. &irritants: ()

> (distribution (set (sample '((#t . 0.5) (#f . 0.5))))
Unhandled exception
Condition components:
1. &error
2. &message: "Top-level nondeterministic-fail"
3. &irritants: ()
> (set (set (choose '(#t #f)))))
  ((#t #f))
> (distribution (distribution (sample '((#t . 0.5) (#f . 0.5)))))
  (((#t . 0.5) (#f . 0.5)) . 1))
> (distribution (set (choose '(#t #f))))
  (((#t #f) . 1))

> (set (set (choose '(#t #f)))))
  ((#t #f))
> (distribution (distribution (sample '((#t . 0.5) (#f . 0.5)))))
  (((#t . 0.5) (#f . 0.5)) . 1))
> (distribution (set (choose '(#t #f))))
  (((#t #f) . 1))

Unhandled exception
Condition components:
1. &error
2. &message: "Top-level probabilistic-fail"
3. &irritants: ()
> (set (set (choose '(#t #f))))
  ((#t #f))
> (distribution (distribution (sample '(@(t . 0.5) @(f . 0.5))))))
  (((@t . 0.5) (@f . 0.5)) . 1))
> (distribution (set (choose '(@(t #f))))
  (((@t #f) . 1))
> (set (distribution (sample '(@(t . 0.5) @(f . 0.5))))))
  (((@t . 0.5) (@f . 0.5)))
> (set (set (choose '(#t #f))))
  ((#t #f))
> (distribution (distribution (sample '(((#t . 0.5) (#f . 0.5))))))
  (((#t . 0.5) (#f . 0.5)) . 1))
> (distribution (set (choose '('#t #f))))
  (((#t #f) . 1))
> (set (distribution (sample '(((#t . 0.5) (#f . 0.5))))))
  (((#t . 0.5) (#f . 0.5)))
> (set (distribution (choose '('#t #f))))
Unhandled exception
Condition components:
  1. &error
  2. &message: "Top-level probabilistic-fail"
  3. &irritants: ()
> (set (set (choose #'t #'f)))
  ('t 'f)
> (distribution (distribution (sample '((t . 0.5) (f . 0.5)))))
  '(((t . 0.5) (f . 0.5)) . 1))
> (distribution (set (choose #'t #'f)))
  '(((t #f) . 1))
> (set (distribution (sample '((t . 0.5) (f . 0.5)))))
  '(((t . 0.5) (f . 0.5)))
> (set (distribution (choose #'t #'f)))
Unhandled exception
  Condition components:
    1. &error
    2. &message: "Top-level probabilistic-fail"
    3. &irritants: ()
> (distribution (set (sample '((t . 0.5) (f . 0.5)))))
Unhandled exception
  Condition components:
    1. &error
    2. &message: "Top-level nondeterministic-fail"
    3. &irritants: ()
The Untyped Nondeterministic Lambda Calculus

\[ E ::= x \mid \lambda x. e \mid e_0 \; e_1 \mid \text{choose } e \mid \text{set } e \]
The Untyped Nondeterministic Lambda Calculus

\[ S_v = \{v\} \]

\[ M f \{v_1, \ldots, v_n, \ldots\} = \{(f v_1), \ldots, (f v_n), \ldots\} \]

\[ \bigcup \{\{v_{11}, \ldots, v_{1n}, \ldots\}, \ldots, \{v_{m1}, \ldots, v_{mn}, \ldots\}, \ldots\} = \{v_{11}, \ldots, v_{1n}, \ldots, \ldots, v_{m1}, \ldots, v_{mn}, \ldots, \ldots\} \]
\[ \mathcal{E} \, \rho \, x = S \, (\rho \, x) \]
\[ \mathcal{E} \, \rho \, \lambda x. e = S \, \lambda v. \mathcal{E} \, \rho[x \mapsto v] \, e \]
\[ \mathcal{E} \, \rho \, (e_0 \, e_1) = \bigcup (\mathcal{M} \, (\lambda v_0. \bigcup (\mathcal{M} \, (\lambda v_1. v_0 \, v_1) \, (\mathcal{E} \, \rho \, e_1)))) \, (\mathcal{E} \, \rho \, e_0)) \]
\[ \mathcal{E} \, \rho \, (\text{choose } e) = \bigcup (\mathcal{E} \, \rho \, e) \]
\[ \mathcal{E} \, \rho \, (\text{set } e) = S \, (\mathcal{E} \, \rho \, e) \]
The Untyped Stratified Nondeterministic Lambda Calculus

\[ E ::= x \mid \lambda x . e \mid e_0 \ e_1 \mid \text{choose}_1 \ e \mid \text{set}_1 \ e \mid \text{choose}_2 \ e \mid \text{set}_2 \ e \]
Note that $\bigcup$ is defined only when the row (column) cardinalities of all of the inner two-axis sets in each column (row) of a given row (column) in the outer two-axis set are the same.
$\text{inject}_1 \{v_1, \ldots, v_n, \ldots\} = \{ v_1 \quad \ldots \quad v_n \quad \ldots \}$

$\text{inject}_2 \{v_1, \ldots, v_n, \ldots\} = \{ v_1 \quad \ldots \quad v_n \quad \ldots \}$
\[
\text{project}_1 \left\{ \begin{array}{c} v_1 \\ \vdots \\ v_n \\ \vdots 
\end{array} \right\} = \{v_1, \ldots, v_n, \ldots\}
\]

\[
\text{project}_2 \left\{ \begin{array}{c} v_1 \\ \vdots \\ v_n \\ \vdots 
\end{array} \right\} = \{v_1, \ldots, v_n, \ldots\}
\]
\[ S_1 \begin{bmatrix} v_{11} & \cdots & v_{1n} & \cdots \\ \vdots & \ddots & \vdots & \cdots \\ v_{m1} & \cdots & v_{mn} & \cdots \end{bmatrix} = \begin{bmatrix} v_{11} \\ \vdots \\ v_{m1} \\ \vdots \end{bmatrix} \cdots \begin{bmatrix} v_{1n} \\ \vdots \end{bmatrix} \cdots \]

\[ S_2 \begin{bmatrix} v_{11} & \cdots & v_{1n} & \cdots \\ \vdots & \ddots & \vdots & \cdots \\ v_{m1} & \cdots & v_{mn} & \cdots \end{bmatrix} = \begin{bmatrix} v_{11} & \cdots & v_{1n} \\ \vdots \\ v_{m1} & \cdots & v_{mn} \end{bmatrix} \cdots \]
\[ \begin{align*}
\mathcal{E} \rho x &= S (\rho x) \\
\mathcal{E} \rho \lambda x.e &= S \lambda v.\mathcal{E} \rho[x \mapsto v] e \\
\mathcal{E} \rho (e_0 e_1) &= \bigcup (\mathcal{M} (\lambda v_0.\bigcup (\mathcal{M} (\lambda v_1.v_0 v_1) (\mathcal{E} \rho e_1))) (\mathcal{E} \rho e_0)) \\
\mathcal{E} \rho (\text{choose}_1 e) &= \bigcup (\mathcal{M} \text{inject}_1 (\mathcal{E} \rho e)) \\
\mathcal{E} \rho (\text{set}_1 e) &= \mathcal{M} \text{project}_1 (S_1 (\mathcal{E} \rho e)) \\
\mathcal{E} \rho (\text{choose}_2 e) &= \bigcup (\mathcal{M} \text{inject}_2 (\mathcal{E} \rho e)) \\
\mathcal{E} \rho (\text{set}_2 e) &= \mathcal{M} \text{project}_2 (S_2 (\mathcal{E} \rho e))
\end{align*} \]
One Issue

Nested Nondeterministic and Stochastic Queries

NEPLS, 10 November 2015

Jeffrey Mark Siskind (Purdue)
> (choose1 '( (a b) (c d)))
{(a b)}
{(c d)}
\[ \text{choose1} '(((a \ b) \ (c \ d))) \]
\[ \{ (a \ b) \} \]
\[ \{ (c \ d) \} \]

\[ \text{choose2} (\text{choose1} '(((a \ b) \ (c \ d)))) \]
\[ \{ a \ b \} \]
\[ \{ c \ d \} \]
> (choose1 ’((a b) (c d)))
  { (a b) }
  { (c d) }
> (choose2 (choose1 ’((a b) (c d))))
  { a b }
  { c d }
> (set2 (choose2 (choose1 ’((a b) (c d)))))
  { (a b) }
  { (c d) }
> (set1 (set2 (choose2 (choose1 ’((a b) (c d)))))
> (set1 (choose2 (choose1 ’((a b) (c d)))))
> (choose1 '((a b) (c d)))
{ (a b) }{ (c d) }
> (choose2 (choose1 '((a b) (c d))))
{ a b }{ c d }
> (set2 (choose2 (choose1 '((a b) (c d)))))
{ (a b) }{ (c d) }
> (set1 (set2 (choose2 (choose1 '((a b) (c d)))))
((a b) (c d))

> (set1 (choose2 (choose1 '((a) (c d))))
>>Error: Invalid argument to \textbackslash \cup
One Issue

> (choose1 '((a b) (c d)))
{ (a b) }
{ (c d) }
> (choose2 (choose1 '((a b) (c d))))
{ a b }
{ c d }
> (set2 (choose2 (choose1 '((a b) (c d)))))
{ (a b) }
{ (c d) }
> (set1 (set2 (choose2 (choose1 '((a b) (c d))))))
((a b) (c d))
> (set1 (choose2 (choose1 '((a b) (c d)))))
{ (a c) (b d) }
> (choose1 '((a b) (c d)))
>   { (a b) } 
>   { (c d) } 
> > (choose2 (choose1 '((a b) (c d))))
>   { a b } 
>   { c d } 
> > (set2 (choose2 (choose1 '((a b) (c d)))))
>   { (a b) } 
>   { (c d) } 
> > (set1 (set2 (choose2 (choose1 '((a b) (c d))))))
>   ((a b) (c d)) 
> > (set1 (choose2 (choose1 '((a b) (c d)))))
>   { (a c) (b d) } 
> > (set2 (set1 (choose2 (choose1 '((a b) (c d))))))
>   ((a c) (b d))
One Issue

> (choose1 '((a b) (c d)))
{ (a b) }
{ (c d) }
> (choose2 (choose1 '((a b) (c d))))
{ a b }
{ c d }
> (set2 (choose2 (choose1 '((a b) (c d)))))
{ (a b) }
{ (c d) }
> (set1 (set2 (choose2 (choose1 '((a b) (c d))))))
((a b) (c d))
> (set1 (choose2 (choose1 '((a b) (c d))))))
{ (a c) (b d) }
> (set2 (set1 (choose2 (choose1 '((a b) (c d))))))
((a c) (b d))
> (set2 (set1 (choose2 (choose1 '((a) (c d))))))
>>Error: Invalid argument to \(\bigcup\)
The Untyped Stratified Probabilistic Lambda Calculus

\[
S_1 \left\{ \langle v_{11}, p_{11}^1, p_{11}^2 \rangle \ldots \langle v_{1n}, p_{1n}^1, p_{1n}^2 \rangle \ldots \right\} = \left\{ \left\{ \langle v_{11}, p_{11}^1, p_{11}^2 + \cdots + p_{1n}^2 \rangle \right\} \ldots \left\{ \langle v_{1n}, p_{1n}^1, p_{1n}^2 + \cdots + p_{mn}^2 \rangle \right\} , 1, p_{11}^2 \right\} \ldots \left\{ \left\{ \langle v_{mn}, p_{mn}^1, p_{mn}^2 + \cdots + p_{mn}^2 \rangle \right\} , 1, p_{mi}^2 \right\} \ldots \right\}
\]

\[
S_2 \left\{ \langle v_{11}, p_{11}^1, p_{11}^2 \rangle \ldots \langle v_{1n}, p_{1n}^1, p_{1n}^2 \rangle \ldots \right\} = \left\{ \left\{ \langle v_{11}, p_{11}^1 + \cdots + p_{mi}^1, p_{1i}^2 \rangle \ldots \langle v_{1n}, p_{1n}^1 + \cdots + p_{mn}^1, p_{1n}^2 \rangle \ldots \right\} , p_{11}^1 \right\} \ldots \left\{ \left\{ \langle v_{mn}, p_{mn}^1 + \cdots + p_{mn}^1, p_{mn}^2 \rangle \ldots \right\} , p_{mi}^1 \right\} \ldots \right\}
\]

Note that \( S_1 \) is defined only when the row vectors of column probabilities are the same across all rows, and \( S_2 \) is defined only when the columns vectors of row probabilities are the same across all columns.
Another Issue
⚠️ Error: Invalid argument to $S$

```plaintext
> (sample1
  '(((a . 0.6) (b . 0.4)) . 0.9) (((c . 0.8) (d . 0.2)) . 0.1))

{ {(((a . 0.6) (b . 0.4)), 0.9, 1) } { (((c . 0.8) (d . 0.2)), 0.1, 1) } }
```
Another Issue

> (sample1
  '(((a . 0.6) (b . 0.4)) . 0.9) (((c . 0.8) (d . 0.2)) . 0.1))
{ ((a . 0.6) (b . 0.4)),0.9,1 } { ((c . 0.8) (d . 0.2)),0.1,1 }
> (sample2
  (sample1
    '(((a . 0.6) (b . 0.4)) . 0.9) (((c . 0.8) (d . 0.2)) . 0.1)))
  { a,0.9,0.6 } { b,0.9,0.4 }
  { c,0.1,0.8 } { d,0.1,0.2 }
Another Issue

> (sample1
  `'(((a . 0.6) (b . 0.4)) . 0.9) (((c . 0.8) (d . 0.2)) . 0.1)))
  { 
    {((a . 0.6) (b . 0.4)),0.9,1} 
    {((c . 0.8) (d . 0.2)),0.1,1} 
  }
> (sample2
  (sample1
   `'(((a . 0.6) (b . 0.4)) . 0.9) (((c . 0.8) (d . 0.2)) . 0.1)))
  { 
    {a,0.9,0.6} {b,0.9,0.4} 
    {c,0.1,0.8} {d,0.1,0.2} 
  }
> (distribution2
  (sample2
   (sample1
    `'(((a . 0.6) (b . 0.4)) . 0.9) (((c . 0.8) (d . 0.2)) . 0.1))))
    { 
      {((a . 0.6) (b . 0.4)),0.9,1} 
      {((c . 0.8) (d . 0.2)),0.1,1} 
    }
Another Issue

>` (sample1
  \(((a . 0.6) (b . 0.4)) . 0.9) (((c . 0.8) (d . 0.2)) . 0.1))`
\{ `(a . 0.6) (b . 0.4)) , 0.9, 1 \}
\{ `(c . 0.8) (d . 0.2)) , 0.1, 1 \}

>` (sample2
  (sample1
    \(((a . 0.6) (b . 0.4)) . 0.9) (((c . 0.8) (d . 0.2)) . 0.1))`
\{ `a,0.9,0.6 \} \{ `b,0.9,0.4 \}
\{ `c,0.1,0.8 \} \{ `d,0.1,0.2 \}

>` (distribution2
  (sample2
    (sample1
      \(((a . 0.6) (b . 0.4)) . 0.9) (((c . 0.8) (d . 0.2)) . 0.1))`
\{ `(a . 0.6) (b . 0.4) , 0.9 \}
\{ `(c . 0.8) (d . 0.2) , 0.1 \}

>` (distribution1
  (distribution2
    (sample2
      (sample1
        \(((a . 0.6) (b . 0.4)) . 0.9) (((c . 0.8) (d . 0.2)) . 0.1)))
\(((a . 0.6) (b . 0.4)) . 0.9) (((c . 0.8) (d . 0.2)) . 0.1))
Another Issue

> (sample1
   '(((a . 0.6) (b . 0.4)) . 0.9) (((c . 0.8) (d . 0.2)) . 0.1))
{ ((a . 0.6) (b . 0.4)), 0.9, 1 } { ((c . 0.8) (d . 0.2)), 0.1, 1 }
> (sample2
   (sample1
    '(((a . 0.6) (b . 0.4)) . 0.9) (((c . 0.8) (d . 0.2)) . 0.1)))
{ a, 0.9, 0.6 } { b, 0.9, 0.4 } { c, 0.1, 0.8 } { d, 0.1, 0.2 }
> (distribution2
   (sample2
    (sample1
     '(((a . 0.6) (b . 0.4)) . 0.9) (((c . 0.8) (d . 0.2)) . 0.1))))
{ ((a . 0.6) (b . 0.4)), 0.9, 1 } { ((c . 0.8) (d . 0.2)), 0.1, 1 }
> (distribution1
   (distribution2
    (sample2
     (sample1
      '(((a . 0.6) (b . 0.4)) . 0.9) (((c . 0.8) (d . 0.2)) . 0.1)))))
(((a . 0.6) (b . 0.4)) . 0.9) (((c . 0.8) (d . 0.2)) . 0.1))
> (distribution1
   (sample2
    (sample1
     '(((a . 0.6) (b . 0.4)) . 0.9) (((c . 0.8) (d . 0.2)) . 0.1))))
>> Error: Invalid argument to $S_1$
{ ((a . 0.9) (c . 0.1)), 1, ? } { ((b . 0.9) (d . 0.1)), 1, ? }
Another Issue
Another Issue

> (sample2
   (sample1
    '(((a . 0.6) (b . 0.4)) . 0.9) (((c . 0.6) (d . 0.4)) . 0.1)))
{ ⟨a,0.9,0.6⟩ ⟨b,0.9,0.4⟩ } { ⟨c,0.1,0.6⟩ ⟨d,0.1,0.6⟩ }
Another Issue

> (sample2
>   (sample1
>     '(((a . 0.6) (b . 0.4)) . 0.9) (((c . 0.6) (d . 0.4)) . 0.1)))
>   {
>     ⟨a,0.9,0.6⟩ ⟨b,0.9,0.4⟩
>   }
>   ⟨c,0.1,0.6⟩ ⟨d,0.1,0.6⟩
>   > (distribution1
>       (sample2
>         (sample1
>           '(((a . 0.6) (b . 0.4)) . 0.9) (((c . 0.6) (d . 0.4)) . 0.1)))
>         {
>           ⟨((a . 0.9) (c . 0.1)),1,0.6⟩ ⟨((b . 0.9) (d . 0.1)),1,0.4⟩
>         }
>   > (distribution2
>       (distribution1(sample2(sample1'((((a . 0.6) (b . 0.4)) . 0.9) (((c . 0.8) (d . 0.2)) . 0.1))))))))
> (sample2
  (sample1
    '(((a . 0.6) (b . 0.4)) . 0.9) (((c . 0.6) (d . 0.4)) . 0.1)))
  ⟨a,0.9,0.6⟩ ⟨b,0.9,0.4⟩
  ⟨c,0.1,0.6⟩ ⟨d,0.1,0.6⟩
> (distribution1
  (sample2
    (sample1
      '(((a . 0.6) (b . 0.4)) . 0.9) (((c . 0.6) (d . 0.4)) . 0.1)))
    ⟨((a . 0.9) (c . 0.1)),1.06⟩ ⟨((b . 0.9) (d . 0.1)),1.04⟩
  )
> (distribution2
  (distribution1
    (sample2
      (sample1
        '(((a . 0.6) (b . 0.4)) . 0.9) (((c . 0.8) (d . 0.2)) . 0.1)))
        (((a . 0.9) (c . 0.1)) . 0.6) (((b . 0.9) (d . 0.1)) . 0.4))
Coherency Conditions

1. The number of choices at every stratum must be the same for all samples/choices of all other strata.

2. The distribution at every stratum must be the same for all samples/choices of all other strata.
The number of choices at every stratum must be the same for all samples/choices of all other strata.
Coherency Conditions

1. The number of choices at every stratum must be the same for all samples/choices of all other strata.

2. The distribution at every stratum must be the same for all samples/choices of all other strata.