Supplement for Metaprob

Anonymous Author(s)

1 Mixture-modeling DSL inference procedure synthesizer

Figure 1 shows the implementation of the inference procedure synthesizer for the mixture-modeling DSL. It is invoked by the top-level **`multi-mixture`** function shown in the main paper. It accepts as input a set of **`varsets`** that each contain information about (a) cluster probabilities, (b) parameters for each variable for each cluster, and (c) base distributions for each variable. It then creates an inference procedure that samples exactly from the posterior of the entire model in accordance with intervention and observation traces.

2 Metacircular interpreter

We now present the implementation of the metacircular interpreter, which is less than 300 lines long. The metacircular interpreter is a program written in Metaprob that implements the language's core inference engine: given a Metaprob procedure, as well as partial traces representing interventions and observations, the **`infer-apply`** function interprets the function body, and produces a *return value*, an *execution trace*, and a *score* that quantifies the accuracy with which the observation trace's constraints were incorporated.

The implementation is split across multiple figures. Figure 2 shows the top-level definition of **`infer-apply`**. Note that this definition uses **`inf`**: **`infer-apply`**'s execution can itself be traced and constrained by observations and interventions. The implementation makes use of several helper functions that apply various kinds of procedures: these are shown in 4.

The meat of the implementation is in **`infer-eval`** (Figure 3), which evaluates an expression, and which is called by **`infer-apply-native`** to interpret generate code. It relies on the helper functions in Figure 5.

Finally, the interpreter uses a datatype called a "tracing context" (Figure **??**), which encapsulates an intervention and observation trace into a single object, and allows them to be manipulated jointly. These tracing contexts can be "captured" by the **`with-explicit-tracer`** construct, in which case they also track state that enables that feature to work correctly.

3 User-space general inference algorithms

In Figure 7, we show rejection sampling and an MCMC operator, implemented as short user-space programs in Metaprob. Rejection sampling samples repeatedly from a model until some condition is satisfied. The **`markov-chain-operator`** function samples an initial trace subject to some constraints, then repeatedly calls a user-provided Markov transition to the trace to get closer to a true posterior sample.

Conference'19, July 2019, Washington, DC, USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00

```
(define multi-mixture-inference-procedure-for
  (gen [varsets]
    ;; varsets is a list of [dists cluster-probs cluster-params]
    ;; for each set of variables.
    (gen [[] traces]
      (reduce
        (gen [[v tr s] [dists probs params]]
          (define variables (keys dists))
          (define __cluster_addr
            (str "cluster-for-" (join "," variables)))

          ;; Compute the log posterior probability of a cluster.
          (define cluster-score
            (gen [cluster-num]
              (define logprior (log (nth probs cluster-num)))
              (define params (nth params cluster-num))
              (define var-score
                (gen [var-name]
                  (if (observed? traces var-name)
                    (block
                      (define [_ _ s]
                        (infer-apply
                          :procedure (get dists var-name),
                          :inputs (get params var-name),
                          :observation-trace
                          (trace-subtrace (get traces :observation-trace) var-name)))
                      s)
                    0)))
              (+ logprior (apply + (map var-score variables)))))

          ;; Sample a cluster from the posterior (or use the constrained
          ;; value), and score for this varset. If cluster has not been
          ;; constrained, this involves marginalizing over the choice of
          ;; cluster.
          (define [cluster marginal]
            (if (contains? traces __cluster_addr)
              (block
                (define k (specified-value traces __cluster_addr))
                (define cluster-prob (log (nth probs k)))
                (define marg (- (cluster-score k)
                  (if (intervened? traces __cluster_addr) cluster-prob 0)))
                [k marg])
              (block
                (define scores (map cluster-score (range (count probs))))
                [(log-categorical scores) (logsumexp scores)])))

          (define params (nth params cluster))
          (reduce
            (gen [[v' tr' s'] var]
              (define sample
                (if (contains? traces var)
                  (specified-value traces var)
                  (apply (get dists var) (get params var))))
              [(cons sample v') (trace-set-value tr' var sample) s'])
            [v (trace-set-value tr __cluster_addr cluster) (+ s marginal)]))

        ['() {} 0]
        varsets))))
```

Figure 1. Implementation of the custom inference procedure synthesizer for the multi-mixture DSL.

```clojure
(define infer-apply
  (gen
    [& {:keys [procedure inputs intervention-trace observation-trace]}]
    (assert procedure "Cannot call `infer-apply` without a procedure.")
    (infer-apply' procedure
                  (or inputs [])
                  (make-top-level-tracing-context
                    (or intervention-trace {})
                    (or observation-trace {})))))

; infer-apply' itself is an inf, because it has custom tracing/proposal behavior.
; This is the model of infer-apply''s behavior.
(define model-of-infer-apply'
  (gen [proc inputs ctx]
    (cond
      (contains? proc :implementation)
      ((get proc :implementation) inputs ctx)

      (native-procedure? proc)
      (infer-apply-native proc inputs ctx)

      (captured-ctx? proc)
      (infer-apply-tc proc inputs ctx)

      (get proc :apply?)
      (infer-apply' (first inputs) (second inputs) ctx)

      (fn? proc)
      (infer-apply-foreign proc inputs ctx)

      true
      (error "infer-apply': not a procedure" proc))))

; infer-apply' with custom tracing: when a primitive is invoked by interpreted code,
; we pretend it is also invoked by the interpreter.
(define infer-apply'
  (inf
    "infer-apply'"
    model-of-infer-apply'
    (gen [[proc ins ctx] ctx']
      (if (get proc :primitive?)
          (if (or (constrained? ctx '()) (not (active-ctx? ctx')))
            ; Case 1: we have a constraint,
            ; so the interpreter (being traced)
            ; needs to generate no randomness.
            [((get proc :implementation) ins ctx) {} 0]

            ; Case 2: the interpreter needs to generate randomness,
            ; because proc's execution is unconstrained.
            ; Score is 0 at proc level.
            (block
              (define [v o s] ((get proc :implementation) ins ctx'))
              [[v o 0] o s]))
        ; Otherwise, use default tracing
        (infer-apply' model-of-infer-apply' [proc ins ctx] ctx')))))
```

Figure 2. Implementation of entry point to traced and scored program execution, **`infer-apply`**.

```
(define infer-eval
  (gen [exp env ctx]
    (assert (environment? env) ["bad env - eval" env])
    (define [v o s]
      (cond
        (variable? exp)
        [(env-lookup env exp) {} 0]

        (vector? exp)
        (infer-eval-expressions exp env ctx)

        (or (not (seq? exp)) (= '() exp))
        [exp {} 0]

        (quote-expr? exp)
        [(quote-quoted exp) {} 0]

        (with-explicit-tracer-expr? exp)
        (block
          (define captured-ctx (capture-tracing-context ctx infer-apply'))
          (define inactive-ctx (assoc ctx :active? false))
          ; Create an environment that has this captured context in it
          (define new-env (make-env env))
          (match-bind! (explicit-tracer-var-name exp) captured-ctx new-env)

          ; Evaluate the body
          (define [values _ s] (infer-eval-expressions (explicit-tracer-body exp) new-env inactive-ctx))

          (define [o-acc score-acc] (release-tracing-context captured-ctx))
          [(last values) o-acc (+ s score-acc)])

        (if-expr? exp)
        (block
          (define [pred-value pred-trace pred-s]
            (infer-eval (if-predicate exp) env (subcontext ctx "predicate")))
          (define clause-adr (if pred-value "then" "else"))
          (define [final-value clause-trace clause-s]
            (infer-eval
              (if pred-value (if-then-clause exp) (if-else-clause exp))
              env (subcontext ctx clause-adr)))
          (define output-trace
            (maybe-set-subtrace (maybe-set-subtrace {} clause-adr clause-trace) "predicate" pred-trace))
          [final-value output-trace (+ pred-s clause-s)])

        (definition? exp)
        (block
          (define [rhs-value out s]
            (infer-subeval (definition-rhs exp) (name-for-definiens (definition-pattern exp)) env ctx))
          [(match-bind! (definition-pattern exp) rhs-value env) out s])

        (block-expr? exp)
        (block
          (define new-env (make-env env))
          (define [values o s]
            (infer-eval-expressions (block-body exp) new-env ctx))
          [(last values) o s])

        (gen-expr? exp)
        [{:name (trace-name exp)
          :generative-source exp, ; (cons 'gen (cons (second exp) (map mp-expand (rest (rest exp)))))
          :environment env}) {} 0]

        ; It's an application:
        true
        (block
          (define key (application-result-key (first exp)))
          (define [evaluated o s] (infer-eval-expressions exp env ctx))
          (define [v app-o app-s] (infer-apply' (first evaluated) (rest evaluated) (subcontext ctx key)))
          [v (maybe-set-subtrace o key app-o) (+ s app-s)])))

    ; These may be nil, if no such value exists.
    (define ivalue (intervened-value ctx '()))
    (define tvalue (observed-value ctx '()))

    (cond
      ; intervention with no disagreeing observation
      (and (intervened? ctx '()) (or (not (observed? ctx '())) (= ivalue tvalue)))
      [(or-nil? ivalue v) o 0]

      ; observation and value (from intervention or execution) disagree
      (and (observed? ctx '()) (not= (or-nil? ivalue v) tvalue))
      (assert false (str "Unsatisfiable target constraint (target=" tvalue ", expected=", (or-nil? ivalue v) ")"))

      ; in all other cases, the existing values work fine:
      true
      [(or-nil? ivalue v) o s])))
```

Figure 3. Implementation of **infer-eval**, which traces and scores generative code expressions.

```
; Invoke a "foreign" (i.e., Clojure) procedure,
; handling intervention and observation traces.
(define infer-apply-foreign
  (gen [proc inputs ctx]
    ;; 'Foreign' generative procedure
    (define value (generate-foreign proc inputs))
    (define ivalue (if (intervened? ctx '()) (intervened-value ctx '()) value))
    [ivalue {} (if (and (observed? ctx '()) (not= (observed-value ctx '()) ivalue)) negative-infinity 0)]))

; Invoke a 'native' generative procedure, i.e. one written in
; Metaprob, with inference mechanics (traces and scores).
(define infer-apply-native
  (gen [proc inputs ctx]
    (define source (get proc :generative-source))
    (define body (gen-body source))
    (define environment (get proc :environment))
    (define new-env (make-env environment))
    ;; Extend the enclosing environment by binding formals to actuals
    (match-bind! (gen-pattern source) ; pattern. (source is of form '(gen [...] ...)
                 inputs
                 new-env)
    (infer-eval (if (empty? (rest body)) (first body) (cons 'block body)) ; body with implicit `block`
                new-env
                ctx)))

; Apply proc at sub-adr in the tracing context tc, given that we are currently in
; old-ctx.
(define infer-apply-tc
  (gen [tc [sub-adr proc & ins] old-ctx]
    (assert (captured-ctx? tc) "Using apply-tc on a non-captured tc.")
    (if (not= (get old-ctx :interpretation-id) (get tc :interpretation-id))
      [(apply tc (cons sub-adr (cons proc ins))) old-ctx 0]
      (block
        (define [out-atom score-atom applicator] (get tc :captured-state))
        (define modified-tc (subcontext (dissoc tc :captured-state) sub-adr))
        (define [v o s] (applicator proc ins modified-tc))
        (swap! out-atom trace-merge (maybe-set-subtrace {} sub-adr o))
        [v {} s]))))
```

Figure 4. Helper functions for applying procedures of different kinds in the metacircular interpreter.

```
(define infer-subeval
  (gen [sub-exp adr env ctx]
    (define [v sub-o s] (infer-eval sub-exp env (subcontext ctx adr)))
    [v (maybe-set-subtrace {} adr sub-o) s]))

(define infer-eval-expressions
  (gen [exp env ctx]
    (second
      (reduce
        (gen [[i [v o prev-s]] next]
          (define [next-v sub-o s]
            (infer-eval next env (subcontext ctx i)))
          [(+ i 1) [(conj v next-v) (maybe-set-subtrace o i sub-o) (+ s prev-s)]])
        [0 [[] {} 0]]
        exp))))
```

Figure 5. Helper functions for evaluating expressions in the metacircular interpreter.

```
(define make-top-level-tracing-context
  (gen [intervention-trace observation-trace]
    {:interpretation-id (gensym)
     :active? true
     :intervention-trace intervention-trace
     :observation-trace observation-trace}))

(define active-ctx?
  (gen [ctx] (get ctx :active?)))

(define captured-ctx?
  (gen [ctx] (not (nil? (get ctx :captured-state)))))

(define subtraces
  (gen [ctx adr]
    (assert (not (captured-ctx? ctx)) "Cannot take subtraces of a captured context")
    (if (active-ctx? ctx)
      (assoc ctx :intervention-trace (maybe-subtrace (get ctx :intervention-trace) adr),
             :observation-trace (maybe-subtrace (get ctx :observation-trace) adr))
      ctx)))

(define attach-clojure-implementation
  (gen [ctx]
    (define ctx' (unbox ctx))
    (if (active-ctx? ctx')
      (clojure.core/with-meta
        (clojure.core/fn [adr proc & args]
          (define [out-atom score-atom applicator] (get ctx' :captured-state))
          (define modified-tc (dissoc ctx' :captured-state))
          (define [v o s] (applicator proc args (subtraces modified-tc adr)))
          (swap! out-atom trace-merge (maybe-set-subtrace {} adr o))
          (swap! score-atom + s)
          v)
        ctx')
      (clojure.core/with-meta
        (clojure.core/fn [adr proc & args] (apply proc args))
        ctx'))))

(define capture-tracing-context
  (gen [ctx applicator]
    (attach-clojure-implementation
      (if (active-ctx? ctx)
        (assoc ctx :captured-state [(cell {}) (cell 0) applicator])
        ctx))))

(define release-tracing-context
  (gen [ctx]
    (if (captured-ctx? ctx)
      (block (define [out-atom score-atom _] (get ctx :captured-state)) [(deref out-atom) (deref score-atom)])
      [{} 0])))

(define observed?
  (gen [ctx adr]
    (trace-has-value? (get ctx :observation-trace) adr)))

(define observed-value
  (gen [ctx adr]
    (trace-value (get ctx :observation-trace) adr)))

(define intervened?
  (gen [ctx adr]
    (trace-has-value? (get ctx :intervention-trace) adr)))

(define intervene-value
  (gen [ctx adr]
    (trace-value (get ctx :intervention-trace) adr)))

(define constrained?
  (gen [ctx adr]
    (or (observed? ctx adr) (intervened? ctx adr))))

(define constrained-value
  (gen [ctx adr]
    (or-nil? (observed-value ctx adr) (intervene-value ctx adr))))
```

Figure 6. Helper functions for manipulating tracing contexts.

```
(define rejection-sample
  (gen [& {:keys [procedure inputs condition]}]
    (define [_ trace _]
      (infer-apply :procedure procedure, :inputs inputs))
    (if (condition trace)
      trace
      (rejection-sample
        :procedure procedure,
        :inputs inputs,
        :condition condition))))
```

(a) General-purpose inference metaprogram based on rejection sampling. This metaprogram uses rejection sampling to produce traces that are sampled exactly from the conditional distribution on executions of the input **`procedure`**, given the constraint that the predicate **`condition`** applied to that trace returns true.

```
(define markov-chain-operator
  (gen [& {:keys [procedure inputs observation-trace
                  iterations transition]}]
    (define transition (or transition single-site-mh-step))
    (define [_ initial-trace _]
      (infer-apply :procedure procedure, :inputs inputs,
        :observation-trace observation-trace))
    (define constraint-addrs (addresses-of observation-trace))
    (reduce
      (gen [t _] (transition procedure inputs t constraint-addrs))
      initial-trace
      (range iterations))))
```

(b) Higher-order procedure for simulating a Markov chain over the space of execution traces of **`procedure`**, to perform approximate inference given the constraints contained in **`observation-trace`**. The input procedure **`transition`** implements the transition kernel of the Markov Chain. This procedure can be used to implement both general-purpose and customized Markov chain Monte Carlo inference algorithms.

Figure 7. General-purpose inference algorithms that have been the basis of previously introduced probabilistic programming langauges can be written in Metaprob as short user-space inference metaprograms, using Metaprob's novel language constructs.