

# Hidden Markov Models

Simon Wimmer

December 7, 2022

## Abstract

This entry contains a formalization of hidden Markov models [3] based on Johannes Hölzl's formalization of discrete time Markov chains [1]. The basic definitions are provided and the correctness of two main (dynamic programming) algorithms for hidden Markov models is proved: the forward algorithm for computing the likelihood of an observed sequence, and the Viterbi algorithm for decoding the most probable hidden state sequence. The Viterbi algorithm is made executable including memoization.

Hidden markov models have various applications in natural language processing. For an introduction see Jurafsky and Martin [2].

## Contents

<b>1</b>	<b>Hidden Markov Models</b>	<b>2</b>
1.1	Definitions . . . . .	2
1.2	Iteration Rule For Likelihood . . . . .	3
1.3	Computation of Likelihood . . . . .	5
1.4	Definition of Maximum Probabilities . . . . .	6
1.5	Iteration Rule For Maximum Probabilities . . . . .	6
1.6	Computation of Maximum Probabilities . . . . .	8
1.7	Decoding the Most Probable Hidden State Sequence . . . . .	9
<b>2</b>	<b>Implementation</b>	<b>11</b>
2.1	The Forward Algorithm . . . . .	12
2.2	The Viterbi Algorithm . . . . .	13
2.3	Misc . . . . .	14
2.4	Executing Concrete HMMs . . . . .	16
<b>3</b>	<b>Example</b>	<b>17</b>

# 1 Hidden Markov Models

```
theory Hidden-Markov-Model
  imports
    Markov-Models.Discrete-Time-Markov-Chain Auxiliary
    HOL-Library.IArray
begin
```

## 1.1 Definitions

Definition of Markov Kernels that are closed w.r.t. to a set of states.

```
locale Closed-Kernel =
  fixes K :: 's ⇒ 't pmf and S :: 't set
  assumes finite: finite S
    and wellformed: S ≠ {}
    and closed: ∀ s. K s ⊆ S
```

An HMM is parameterized by a Markov kernel for the transition probabilities between internal states, a Markov kernel for the output probabilities of observations, and a fixed set of observations.

```
locale HMM-defs =
  fixes K :: 's ⇒ 's pmf and O :: 's ⇒ 't pmf and Os :: 't set
```

```
locale HMM =
  HMM-defs + O: Closed-Kernel O Os
begin
```

```
lemma observations-finite: finite Os
  and observations-wellformed: Os ≠ {}
  and observations-closed: ∀ s. O s ⊆ Os
  using O.finite O.wellformed O.closed by -
```

end

Fixed set of internal states.

```
locale HMM2-defs = HMM-defs K O for K :: 's ⇒ 's pmf and O :: 's ⇒ 't pmf +
  fixes S :: 's set
```

```
locale HMM2 = HMM2-defs + HMM + K: Closed-Kernel K S
begin
```

```
lemma states-finite: finite S
  and states-wellformed: S ≠ {}
  and states-closed: ∀ s. K s ⊆ S
  using K.finite K.wellformed K.closed by -
```

end

The set of internal states is now given as a list to iterate over. This is needed for the computations on HMMs.

```
locale HMM3-defs = HMM2-defs Os K for Os :: 't set and K :: 's ⇒ 's pmf +
  fixes state-list :: 's list
```

```
locale HMM3 = HMM3-defs - - Os K + HMM2 Os K for Os :: 't set and K :: 's ⇒ 's pmf +
  assumes state-list-S: set state-list = S
```

```
context HMM-defs
begin
```

```
no-notation (ASCII) comp (infixl o 55)
```

The “default” observation.

**definition**

$obs \equiv SOME\ x.\ x \in \mathcal{O}_s$

**lemma (in HMM) obs:**

$obs \in \mathcal{O}_s$

**unfolding** *obs-def using observations-wellformed by (auto intro: someI-ex)*

The HMM is encoded as a Markov chain over pairs of states and observations. This is the Markov chain’s defining Markov kernel.

**definition**

$K \equiv \lambda\ (s_1, o_1 :: 't).\ bind\text{-}pmf\ (\mathcal{K}\ s_1)\ (\lambda\ s_2.\ map\text{-}pmf\ (\lambda\ o_2.\ (s_2, o_2))\ (\mathcal{O}\ s_2))$

**sublocale** *MC-syntax K .*

Uniform distribution of the pairs  $(s, o)$  for a fixed state  $s$ .

**definition**  $I\ (s :: 's) = map\text{-}pmf\ (\lambda\ x.\ (s, x))\ (pmf\text{-}of\text{-}set\ \mathcal{O}_s)$

The likelihood of an observation sequence given a starting state  $s$  is defined in terms of the trace space of the Markov kernel given the uniform distribution of pairs for  $s$ .

**definition**

$likelihood\ s\ os = T'\ (I\ s)\ \{\omega \in space\ S.\ \exists\ o_0\ xs\ \omega'.\ \omega = (s, o_0)\ \#\#\ xs\ @-\ \omega' \wedge map\ snd\ xs = os\}$

**abbreviation (input)**  $L\ os\ \omega \equiv \exists\ xs\ \omega'.\ \omega = xs\ @-\ \omega' \wedge map\ snd\ xs = os$

**lemma likelihood-alt-def:**  $likelihood\ s\ os = T'\ (I\ s)\ \{(s, o)\ \#\#\ xs\ @-\ \omega' \mid o\ xs\ \omega'.\ map\ snd\ xs = os\}$

**unfolding** *likelihood-def by (simp add: in-S)*

## 1.2 Iteration Rule For Likelihood

**lemma L-Nil:**

$L\ []\ \omega = True$

**by** *simp*

**lemma emeasure-T-observation-Nil:**

$T\ (s, o_0)\ \{\omega \in space\ S.\ L\ []\ \omega\} = 1$

**by** *simp*

**lemma L-Cons:**

$L\ (o\ \#\ os)\ \omega \longleftrightarrow snd\ (shd\ \omega) = o \wedge L\ os\ (stl\ \omega)$

**apply** *(cases  $\omega$ ; cases  $shd\ \omega$ ; safe; clarsimp)*

**apply** *force*

**subgoal for**  $x\ xs\ \omega'$

**by** *(force intro: exI[where  $x = (x, o)\ \#\ xs$ ])*

**done**

**lemma L-measurable[measurable]:**

$Measurable.\ pred\ S\ (L\ os)$

**apply** *(induction  $os$ )*

**apply** *(simp; fail)*

**subgoal premises that for**  $o\ os$

**by** *(subst L-Cons)*

*(intro Measurable.pred-intros-logic*

*measurable-compose[OF measurable-shd] measurable-compose[OF measurable-stl that];*

*measurable)*

**done**

**lemma init-measurable[measurable]:**

$Measurable.\ pred\ S\ (\lambda x.\ \exists\ o_0\ xs\ \omega'.\ x = (s, o_0)\ \#\#\ xs\ @-\ \omega' \wedge map\ snd\ xs = os)$

```

(is Measurable.pred S ?f)
proof -
  have *: ?f  $\omega \longleftrightarrow$  fst (shd  $\omega$ ) = s  $\wedge$  L os (stl  $\omega$ ) for  $\omega$ 
  by (cases  $\omega$ ) auto
  show ?thesis
  by (subst *)
  (intro Measurable.pred-intros-logic measurable-compose[OF measurable-shd]; measurable)
qed

```

```

lemma T-init-observation-eq:
  T (s, o) { $\omega \in$  space S. L os  $\omega$ } = T (s, o') { $\omega \in$  space S. L os  $\omega$ }
  apply (subst emeasure-Collect-T[unfolded space-T], (measurable; fail))
  apply (subst (2) emeasure-Collect-T[unfolded space-T], (measurable; fail))
  apply (simp add: K-def)
done

```

Shows that it is equivalent to define likelihood in terms of the trace space starting at a single pair of an internal state  $s$  and the default observation  $obs$ .

```

lemma (in HMM) likelihood-init:
  likelihood s os = T (s, obs) { $\omega \in$  space S. L os  $\omega$ }
proof -
  have *: ( $\sum$   $o \in \mathcal{O}_s$ . emeasure (T (s, o)) { $\omega \in$  space S. L os  $\omega$ }) =
    of-nat (card  $\mathcal{O}_s$ ) * emeasure (T (s, obs)) { $\omega \in$  space S. L os  $\omega$ }
  by (subst sum-constant[symmetric]) (fastforce intro: sum.cong T-init-observation-eq[simplified])
  show ?thesis
  unfolding likelihood-def
  apply (subst emeasure-T')
  subgoal
  by measurable
  using *
  apply (simp add: I-def in-S observations-finite observations-wellformed nn-integral-pmf-of-set)
  apply (subst mult.commute)
  apply (simp add: observations-finite observations-wellformed mult-divide-eq-ennreal)
  done
qed

```

```

lemma emeasure-T-observation-Cons:
  T (s, o0) { $\omega \in$  space S. L (o1 # os)  $\omega$ } =
  ( $\int$   $t$ . ennreal (pmf ( $\mathcal{O}$  t) o1) * T (t, o1) { $\omega \in$  space S. L os  $\omega$ })  $\partial(\mathcal{K}$  s) (is ?l = ?r)
proof -
  have *:
     $\int$   $y$ . T (s', y) { $x \in$  space S.  $\exists$  xs. ( $\exists$   $\omega'$ . (s', y) ##  $x =$  xs @-  $\omega'$ )  $\wedge$  map snd xs = o1 # os}
     $\partial$ measure-pmf ( $\mathcal{O}$  s') =
    ennreal (pmf ( $\mathcal{O}$  s') o1) * T (s', o1) { $\omega \in$  space S.  $\exists$  xs. ( $\exists$   $\omega'$ .  $\omega =$  xs @-  $\omega'$ )  $\wedge$  map snd xs = os}
    (is ?L = ?R) for s'
  proof -
    have ?L =  $\int$   $x$ . ennreal (pmf ( $\mathcal{O}$  s') x) *
      T (s', x) { $\omega \in$  space S.  $\exists$  xs. ( $\exists$   $\omega'$ . (s', x) ##  $\omega =$  xs @-  $\omega'$ )  $\wedge$  map snd xs = o1 # os}
       $\partial$ count-space UNIV
    by (rule nn-integral-measure-pmf)
    also have ... =
       $\int$   $o_2$ . (if o2 = o1
        then ennreal (pmf ( $\mathcal{O}$  s') o1) * T (s', o1) { $\omega \in$  space S. L os  $\omega$ }
        else 0)
       $\partial$ count-space UNIV
    apply (rule nn-integral-cong-AE
      [where v =  $\lambda$  o2. if o2 = o1
        then ennreal (pmf ( $\mathcal{O}$  s') o1) * T (s', o1) { $\omega \in$  space S. L os  $\omega$ } else 0]
      )
    apply (rule AE-I2)
  qed

```

```

    apply (split if-split, safe)
  subgoal
    by (auto intro!: arg-cong2[where f = times, OF HOL.refl] arg-cong2[where f = emeasure];
        metis list.simps(9) shift.simps(2) snd-conv
    )
  subgoal
    by (subst arg-cong2[where f = emeasure and d = {}, OF HOL.refl]) auto
  done
  also have ... =  $\int^+ o_2 \in \{o_1\}$ .
    (ennreal (pmf ( $\mathcal{O}$  s') o1) * T (s', o1) { $\omega \in \text{space } S. L \text{ os } \omega$ })
     $\partial$ count-space UNIV
    by (rule nn-integral-cong-AE) auto
  also have ... = ?R
    by simp
  finally show ?thesis .
qed
have ?l =  $\int^+ t. T t \{x \in \text{space } S. \exists xs \omega'. t \#\# x = xs @- \omega' \wedge \text{map snd } xs = o_1 \# os\} \partial (K (s, o_0))$ 
  by (subst emeasure-Collect-T[unfolded space-T], measurable)
also have ... = ?r
  using * by (simp add: K-def)
finally show ?thesis .
qed

```

### 1.3 Computation of Likelihood

```

fun backward where
  backward s [] = 1 |
  backward s (o # os) = ( $\int^+ t. \text{ennreal (pmf } (\mathcal{O} \ t) \ o) * \text{backward } t \ os \ \partial \text{measure-pmf } (\mathcal{K} \ s)$ )

```

```

lemma emeasure-T-observation-backward:
  emeasure (T (s, o)) { $\omega \in \text{space } S. L \text{ os } \omega$ } = backward s os
  using emeasure-T-observation-Cons by (induction os arbitrary: s o; simp)

```

```

lemma (in HMM) likelihood-backward:
  likelihood s os = backward s os
  unfolding likelihood-init emeasure-T-observation-backward ..

```

end

```

context HMM2
begin

```

```

fun (in HMM2-defs) forward where
  forward s t-end [] = indicator {t-end} s |
  forward s t-end (o # os) =
    ( $\sum t \in \mathcal{S}. \text{ennreal (pmf } (\mathcal{O} \ t) \ o) * \text{ennreal (pmf } (\mathcal{K} \ s) \ t) * \text{forward } t \ t\text{-end } os$ )

```

```

lemma forward-split:
  forward s t (os1 @ os2) = ( $\sum t' \in \mathcal{S}. \text{forward } s \ t' \ os1 * \text{forward } t' \ t \ os2$ )
  if s ∈ S
  using that
  apply (induction os1 arbitrary: s)
  subgoal for s
    apply (simp add: sum-indicator-mult[OF states-finite])
    apply (subst sum.cong[where B = {s}])
    by auto
  subgoal for a os1 s
    apply simp
    apply (subst sum-distrib-right)
    apply (subst sum.swap)
    apply (simp add: sum-distrib-left algebra-simps)

```

**done**  
**done**

**lemma** (in -)  
 $(\sum t \in S. f t) = f t$  if finite  $S$   $t \in S \forall s \in S - \{t\}. f s = 0$   
**thm** *sum.empty sum.insert sum.mono-neutral-right*[of  $S \{t\}$ ]  
**apply** (*subst sum.mono-neutral-right*[of  $S \{t\}$ ])  
**using** *that*  
**apply** *auto*  
**done**

**lemma** *forward-backward*:  
 $(\sum t \in S. forward\ s\ t\ os) = backward\ s\ os$  if  $s \in S$   
**using**  $\langle s \in S \rangle$   
**apply** (*induction os arbitrary: s*)  
**subgoal** for  $s$   
**by** (*subst sum.mono-neutral-right*[of  $S \{s\}$ , *OF states-finite*])  
*(auto split: if-split-asm simp: indicator-def)*  
**subgoal** for  $a\ os\ s$   
**apply** (*simp add: sum.swap sum-distrib-left*[*symmetric*])  
**apply** (*subst nn-integral-measure-pmf-support*[**where**  $A = S$ ])  
**using** *states-finite states-closed* **by** (*auto simp: algebra-simps*)  
**done**

**theorem** *likelihood-forward*:  
 $likelihood\ s\ os = (\sum t \in S. forward\ s\ t\ os)$  if  $\langle s \in S \rangle$   
**unfolding** *likelihood-backward forward-backward*[*symmetric, OF*  $\langle s \in S \rangle$ ] ..

## 1.4 Definition of Maximum Probabilities

**abbreviation** (*input*)  $V\ os\ as\ \omega \equiv (\exists\ \omega'.\ \omega = zip\ as\ os\ @-\ \omega')$

**definition**  
 $max-prob\ s\ os =$   
 $Max\ \{T'\ (I\ s)\ \{\omega \in space\ S.\ \exists\ o\ \omega'.\ \omega = (s,\ o)\ \#\#\ zip\ as\ os\ @-\ \omega'\}$   
 $\mid as.\ length\ as = length\ os \wedge set\ as \subseteq S\}$

**fun** *viterbi-prob* **where**  
 $viterbi-prob\ s\ t-end\ [] = indicator\ \{t-end\}\ s$   
 $viterbi-prob\ s\ t-end\ (o\ \#\ os) =$   
 $(MAX\ t \in S. ennreal\ (pmf\ (\mathcal{O}\ t)\ o * pmf\ (\mathcal{K}\ s)\ t) * viterbi-prob\ t\ t-end\ os)$

**definition**  
 $is-decoding\ s\ os\ as \equiv$   
 $T'\ (I\ s)\ \{\omega \in space\ S.\ \exists\ o\ \omega'.\ \omega = (s,\ o)\ \#\#\ zip\ as\ os\ @-\ \omega'\} = max-prob\ s\ os \wedge$   
 $length\ as = length\ os \wedge set\ as \subseteq S$

## 1.5 Iteration Rule For Maximum Probabilities

**lemma** *emeasure-T-state-Nil*:  
 $T\ (s,\ o_0)\ \{\omega \in space\ S.\ V\ []\ as\ \omega\} = 1$   
**by** *simp*

**lemma** *max-prob-T-state-Nil*:  
 $Max\ \{T\ (s,\ o)\ \{\omega \in space\ S.\ V\ []\ as\ \omega\} \mid as.\ length\ as = length\ [] \wedge set\ as \subseteq S\} = 1$   
**by** (*simp add: emeasure-T-state-Nil*)

**lemma** *V-Cons*:  $V\ (o\ \#\ os)\ (a\ \#\ as)\ \omega \longleftrightarrow fst\ (shd\ \omega) = a \wedge snd\ (shd\ \omega) = o \wedge V\ os\ as\ (stl\ \omega)$   
**by** (*cases*  $\omega$ ) *auto*

```

lemma measurable-V[measurable]:
  Measurable.pred S (λω. V os as ω)
proof (induction os as rule: list-induct2')
  case (4 x xs y ys)
  then show ?case
    by (subst V-Cons)
      (intro Measurable.pred-intros-logic
        measurable-compose[OF measurable-shd] measurable-compose[OF measurable-stl];
        measurable)
qed simp+

lemma init-V-measurable[measurable]:
  Measurable.pred S (λx. ∃ o ω'. x = (s, o) ## zip as os @- ω') (is Measurable.pred S ?f)
proof -
  have *: ?f ω ⟷ fst (shd ω) = s ∧ V os as (stl ω) for ω
    by (cases ω) auto
  show ?thesis
    by (subst *)
      (intro Measurable.pred-intros-logic measurable-compose[OF measurable-shd]; measurable)
qed

lemma max-prob-Cons':
  Max {T (s, o₁) {ω ∈ space S. V (o # os) as ω} | as. length as = length (o # os) ∧ set as ⊆ S} =
  (
    MAX t ∈ S. ennreal (pmf (O t) o * pmf (K s) t) *
    (MAX as ∈ {as. length as = length os ∧ set as ⊆ S}. T (t, o) {ω ∈ space S. V os as ω})
  ) (is ?l = ?r)
  and T-V-Cons:
  T (s, o₁) {ω ∈ space S. V (o # os) (t # as) ω}
  = ennreal (pmf (O t) o * pmf (K s) t) * T (t, o) {ω ∈ space S. V os as ω}
  (is ?l' = ?r')
  if length as = length os
proof -
  let ?S = λ os. {as. length as = length os ∧ set as ⊆ S}
  have S-finite: finite (?S os) for os :: 't list
    using finite-lists-length-eq[OF states-finite] by (rule finite-subset[rotated]) auto
  have S-nonempty: ?S os ≠ {} for os :: 't list
proof -
  let ?a = SOME a. a ∈ S let ?as = replicate (length os) ?a
  from states-wellformed have ?a ∈ S
    by (auto intro: someI-ex)
  then have ?as ∈ ?S os
    by auto
  then show ?thesis
    by force
qed
  let ?f = λ t as os. T t {ω ∈ space S. V os as (t ## ω)}
  let ?g = λ t as os. T t {ω ∈ space S. V os as ω}
  have *: ?f t as (o # os) = ?g t (tl as) os * indicator {(hd as, o)} t
    if length as = Suc n for t as n
    unfolding indicator-def using that by (cases as) auto
  have **: K (s, o₁) {(t, o)} = pmf (O t) o * pmf (K s) t for t
    unfolding K-def
  apply (simp add: vimage-def)
  apply (subst arg-cong2[where
    f = nn-integral and d = λ x. O x {xa. xa = o ∧ x = t} * indicator {t} x,
    OF HOL.refl])
  subgoal
    by (auto simp: indicator-def)
  by (simp add: emeasure-pmf-single ennreal-mult')
  have ?l = (MAX as ∈ ?S (o # os). ∫+ t. ?f t as (o # os) ∂K (s, o₁))

```

```

  by (subst Max-to-image2; subst emeasure-Collect-T[unfolded space-T]; rule measurable-V HOL.refl)
also have ... = (MAX as ∈ ?S (o # os). ∫+ t. ?g t (tl as) os * indicator {(hd as,o)} t ∂K (s,o1))
  by (simp cong: Max-image-cong-simp add: *)
also have ... = (MAX(t, as) ∈ S × ?S os. ennreal (pmf (O t) o * pmf (K s) t) * ?g (t, o) as os)
proof ((rule Max-eq-image-if; clarsimp?), goal-cases)
  case 1
  from S-finite[of o # os] show ?case
  by simp
next
  case 2
  from states-finite show ?case
  by (blast intro: S-finite)
next
  case (3 as)
  then show ?case
  by - (rule bexI[where x = hd as]; cases as; auto simp: algebra-simps **)
next
  case (4 x as)
  then show ?case
  by - (rule exI[where x = x # as], simp add: algebra-simps **)
qed
also have ... = ?r
  by (subst Max-image-left-mult[symmetric], fact+)
  (rule sym, rule Max-image-pair, rule states-finite, fact+)
finally show ?l = ?r .
have ?l' = ∫+ t'. ?f t' (t # as) (o # os) ∂K (s, o1)
  by (rule emeasure-Collect-T[unfolded space-T]; rule measurable-V)
also from that have ... = ∫+ t'. ?g t' as os * indicator {(t,o)} t' ∂K (s,o1)
  by (subst *[of - length as]; simp)
also have ... = ?r'
  by (simp add: **, simp only: algebra-simps)
finally show ?l' = ?r' .
qed

```

lemmas max-prob-Cons = max-prob-Cons'[OF length-replicate]

## 1.6 Computation of Maximum Probabilities

lemma T-init-V-eq:

```

T (s, o) {ω ∈ space S. V os as ω} = T (s, o') {ω ∈ space S. V os as ω}
apply (subst emeasure-Collect-T[unfolded space-T], (measurable; fail))
apply (subst (2) emeasure-Collect-T[unfolded space-T], (measurable; fail))
apply (simp add: K-def)
done

```

lemma T'-I-T:

```

T' (I s) {ω ∈ space S. ∃ o ω'. ω = (s, o) ## zip as os @- ω'} = T (s,o) {ω ∈ space S. V os as ω}
proof -
  have (∑ o ∈ Os. T (s, o) {ω ∈ space S. V os as ω}) =
    of-nat (card Os) * T (s, o) {ω ∈ space S. V os as ω} for as
  by (subst sum-constant[symmetric]) (fastforce intro: sum.cong T-init-V-eq[simplified])
  then show ?thesis
  unfolding max-prob-def
  apply (subst emeasure-T')
  subgoal
  by measurable
  apply (simp add: I-def in-S observations-finite observations-wellformed nn-integral-pmf-of-set)
  apply (subst mult.commute)
  apply (simp add: observations-finite observations-wellformed mult-divide-eq-ennreal)
  done
qed

```



**lemma** *max-prob-init*:

*max-prob*  $s\ os = \text{Max} \{T\ (s, o) \{\omega \in \text{space } S. V\ os\ as\ \omega\} \mid as.\ \text{length } as = \text{length } os \wedge \text{set } as \subseteq \mathcal{S}\}$   
**unfolding** *max-prob-def* **by** (*simp* *add*:  $T'-I-T[\text{symmetric}]$ )

**lemma** *max-prob-Nil*[*simp*]:

*max-prob*  $s\ [] = 1$   
**unfolding** *max-prob-init*[**where**  $o = \text{obs}$ ] **by** *auto*

**lemma** *Max-start*:

$(\text{MAX } t \in \mathcal{S}. (\text{indicator } \{t\} s :: \text{ennreal})) = 1$  **if**  $s \in \mathcal{S}$   
**using** *states-finite* **that** **by** (*auto* *simp*: *indicator-def* *intro*: *Max-eqI*)

**lemma** *Max-V-viterbi*:

$(\text{MAX } t \in \mathcal{S}. \text{viterbi-prob } s\ t\ os) =$   
 $\text{Max} \{T\ (s, o) \{\omega \in \text{space } S. V\ os\ as\ \omega\} \mid as.\ \text{length } as = \text{length } os \wedge \text{set } as \subseteq \mathcal{S}\}$  **if**  $s \in \mathcal{S}$   
**using** *that* *states-finite* *states-wellformed*  
**by** (*induction* *os* *arbitrary*:  $s\ o$ ; *simp*  
*add*: *Max-start* *max-prob-Cons*[*simplified*] *Max-image-commute* *Max-image-left-mult* *Max-to-image2*  
*cong*: *Max-image-cong*  
 )

**lemma** *max-prob-viterbi*:

$(\text{MAX } t \in \mathcal{S}. \text{viterbi-prob } s\ t\ os) = \text{max-prob } s\ os$  **if**  $s \in \mathcal{S}$   
**using** *max-prob-init*[*of*  $s\ os$ ] *Max-V-viterbi*[*OF*  $\langle s \in \mathcal{S} \rangle$ , *symmetric*] **by** *simp*

**end**

## 1.7 Decoding the Most Probable Hidden State Sequence

**context** *HMM3*

**begin**

**fun** *viterbi* **where**

*viterbi*  $s\ t\ \text{-end}\ [] = ([], \text{indicator } \{t\ \text{-end}\} s) \mid$   
*viterbi*  $s\ t\ \text{-end}\ (o \# os) = \text{fst} (\text{argmax } \text{snd} (\text{map}$   
 $(\lambda t. \text{let } (xs, v) = \text{viterbi } t\ t\ \text{-end}\ os \text{ in } (t \# xs, \text{ennreal } (\text{pmf } (\mathcal{O } t) o * \text{pmf } (\mathcal{K } s) t) * v))$   
 $\text{state-list}))$

**lemma** *state-list-nonempty*:

*state-list*  $\neq []$   
**using** *state-list-S* *states-wellformed* **by** *auto*

**lemma** *viterbi-viterbi-prob*:

*snd* (*viterbi*  $s\ t\ \text{-end}\ os$ ) = *viterbi-prob*  $s\ t\ \text{-end}\ os$

**proof** (*induction* *os* *arbitrary*:  $s$ )

**case** *Nil*

**then show** *?case*

**by** *simp*

**next**

**case** (*Cons*  $o\ os$ )

**let** *?f* =

$\lambda t. \text{let } (xs, v) = \text{viterbi } t\ t\ \text{-end}\ os \text{ in } (t \# xs, \text{ennreal } (\text{pmf } (\mathcal{O } t) o * \text{pmf } (\mathcal{K } s) t) * v)$

**let** *?xs* = *map* *?f* *state-list*

**from** *state-list-nonempty* **have** *map* *?f* *state-list*  $\neq []$

**by** *simp*

**from** *argmax*(2,3)[*OF* *this*, *of* *snd*] **have** \*:

*snd* (*fst* (*argmax* *snd* *?xs*)) = *snd* (*argmax* *snd* *?xs*)

*snd* (*argmax* *snd* *?xs*) = ( $\text{MAX } x \in \text{set } ?xs. \text{snd } x$ ).

**then show** *?case*

```

apply (simp add: state-list-S)
apply (rule Max-eq-image-if)
  apply (intro finite-imageI states-finite; fail)
  apply (intro finite-imageI states-finite; fail)
subgoal
  apply clarsimp
  subgoal for x
    using Cons.IH[of x] by (auto split: prod.splits)
  done
apply clarsimp
subgoal for x
  using Cons.IH[of x] by (force split: prod.splits)
done
qed

```

```

context
begin

```

```

private fun val-of where
  val-of s [] [] = 1 |
  val-of s (t # xs) (o # os) = ennreal (pmf (O t) o * pmf (K s) t) * val-of t xs os

```

```

lemma val-of-T:
  val-of s as os = T (s, o1) {ω ∈ space S. V os as ω} if length as = length os
  using that by (induction arbitrary: o1 rule: val-of.induct; (subst T-V-Cons)?; simp)

```

```

lemma viterbi-sequence:
  snd (viterbi s t-end os) = val-of s (fst (viterbi s t-end os)) os
if snd (viterbi s t-end os) > 0
using that
proof (induction os arbitrary: s)
  case Nil
  then show ?case
    by (simp add: indicator-def split: if-split-asm split-of-bool-asm)
next
  case (Cons o os s)
  let ?xs = map
    (λt. let (xs, v) = viterbi t t-end os in (t # xs, ennreal (pmf (O t) o * pmf (K s) t) * v))
    state-list
  from state-list-nonempty have ?xs ≠ []
  by simp
  from argmax(1)[OF this, of snd] obtain t where
    t ∈ set state-list
    fst (argmax snd ?xs) =
    (t # fst (viterbi t t-end os), ennreal (pmf (O t) o * pmf (K s) t) * snd (viterbi t t-end os))
  by (auto split: prod.splits)
  with Cons show ?case
  by (auto simp: ennreal-zero-less-mult-iff)
qed

```

```

lemma viterbi-valid-path:
  length as = length os ∧ set as ⊆ S if viterbi s t-end os = (as, v)
using that proof (induction os arbitrary: s as v)
  case Nil
  then show ?case
    by simp
next
  case (Cons o os s as v)
  let ?xs = map
    (λt. let (xs, v) = viterbi t t-end os in (t # xs, ennreal (pmf (O t) o * pmf (K s) t) * v))
    state-list

```

**from** *state-list-nonempty* **have**  $?xs \neq []$   
**by** *simp*  
**from** *argmax(1)[OF this, of snd]* **obtain**  $t$  **where**  $t \in \mathcal{S}$   
 $fst (argmax\ snd\ ?xs) =$   
 $(t \# fst (viterbi\ t\ t\text{-end}\ os),\ ennreal (pmf (\mathcal{O}\ t)\ o * pmf (\mathcal{K}\ s)\ t) * snd (viterbi\ t\ t\text{-end}\ os))$   
**by** (*auto simp: state-list-S split: prod.splits*)  
**with** *Cons.premis* **show**  $?case$   
**by** (*cases viterbi t t-end os; simp add: Cons.IH*)  
**qed**

**definition**

*viterbi-final s os = fst (argmax snd (map ( $\lambda t.$  viterbi s t os) state-list))*

**lemma** *viterbi-finalE*:

**obtains**  $t$  **where**

$t \in \mathcal{S}$  *viterbi-final s os = viterbi s t os*  
 $snd (viterbi\ s\ t\ os) = Max ((\lambda t. snd (viterbi\ s\ t\ os)) \text{' } \mathcal{S})$

**proof** –

**from** *state-list-nonempty* **have**  $map (\lambda t. viterbi\ s\ t\ os)\ state\text{-list} \neq []$   
**by** *simp*

**from** *argmax[OF this, of snd]* **show**  $?thesis$

**by** (*auto simp: state-list-S image-comp comp-def viterbi-final-def intro: that*)

**qed**

**theorem** *viterbi-final-max-prob*:

**assumes** *viterbi-final s os = (as, v) s  $\in \mathcal{S}$*

**shows**  $v = max\text{-prob}\ s\ os$

**proof** –

**obtain**  $t$  **where**  $t \in \mathcal{S}$  *viterbi-final s os = viterbi s t os*  
 $snd (viterbi\ s\ t\ os) = Max ((\lambda t. snd (viterbi\ s\ t\ os)) \text{' } \mathcal{S})$

**by** (*rule viterbi-finalE*)

**with** *assms* **show**  $?thesis$

**by** (*simp add: viterbi-viterbi-prob max-prob-viterbi*)

**qed**

**theorem** *viterbi-final-is-decoding*:

**assumes** *viterbi-final s os = (as, v) v > 0 s  $\in \mathcal{S}$*

**shows** *is-decoding s os as*

**proof** –

**from** *viterbi-valid-path[of s - os as v] assms* **have**  $as: length\ as = length\ os\ set\ as \subseteq \mathcal{S}$   
**by** – (*rule viterbi-finalE[of s os]; simp*)**+**

**obtain**  $t$  **where**  $t \in \mathcal{S}$  *viterbi-final s os = viterbi s t os*

**by** (*rule viterbi-finalE*)

**with** *assms viterbi-sequence[of s t os]* **have**  $val\text{-of}\ s\ as\ os = v$

**by** (*cases viterbi s t os*) (*auto simp: snd-def split!: prod.splits*)

**with** *val-of-T as* **have**  $max\text{-prob}\ s\ os = T (s, obs) \{\omega \in space\ S. V\ os\ as\ \omega\}$

**by** (*simp add: viterbi-final-max-prob[OF assms(1,3)]*)

**with** *as* **show**  $?thesis$

**unfolding** *is-decoding-def* **by** (*simp only: T'-I-T*)

**qed**

**end**

**end**

**end**

## 2 Implementation

**theory** *HMM-Implementation*

```

imports
  Hidden-Markov-Model
  Monad-Memo-DP.State-Main
begin

```

## 2.1 The Forward Algorithm

```

locale HMM4 = HMM3 - -  $\mathcal{O}_s \mathcal{K}$  for  $\mathcal{O}_s :: 't \text{ set}$  and  $\mathcal{K} :: 's \Rightarrow 's \text{ pmf} +$ 
  assumes states-distinct: distinct state-list

```

```

context HMM3-defs
begin

```

```

context
  fixes os :: 't iarray
begin

```

Alternative definition using indices into the list of states. The list of states is implemented as an immutable array for better performance.

```

function forward-ix-rec where
  forward-ix-rec s t-end n = (if  $n \geq \text{IArray.length } os$  then indicator {t-end} s else
    ( $\sum t \leftarrow \text{state-list.}$ 
      ennreal (pmf ( $\mathcal{O}$  t) (os !! n)) * ennreal (pmf ( $\mathcal{K}$  s) t) * forward-ix-rec t t-end (n + 1)))

```

```

  by auto
termination
  by (relation Wellfounded.measure ( $\lambda(-, -, n). \text{IArray.length } os - n$ )) auto

```

Memoization

```

memoize-fun forward-ixm: forward-ix-rec
  with-memory dp-consistency-mapping
  monadifies (state) forward-ix-rec.simps[unfolded Let-def]
  term forward-ixm'
memoize-correct
  by memoize-prover

```

The main theorems generated by memoization.

```

context
  includes state-monad-syntax
begin
thm forward-ixm'.simps forward-ixm-def
thm forward-ixm.memoized-correct
end

```

**end**

```

definition
  forward-ix os = forward-ix-rec (IArray os)

```

```

definition
  likelihood-compute s os  $\equiv$ 
    if s  $\in$  set state-list then Some ( $\sum t \leftarrow \text{state-list.}$  forward s t os) else None

```

**end**

Correctness of the alternative definition.

```

lemma (in HMM3) forward-ix-drop-one:
  forward-ix (o # os) s t (n + 1) = forward-ix os s t n
  by (induction length os - n arbitrary: s n; simp add: forward-ix-def)

```

```

lemma (in HMM4) forward-ix-forward:
  forward-ix os s t 0 = forward s t os
  unfolding forward-ix-def
proof (induction os arbitrary: s)
  case Nil
  then show ?case
    by simp
next
  case (Cons o os)
  show ?case
    using forward-ix-drop-one[unfolded forward-ix-def] states-distinct
    by (subst forward.simps, subst forward-ix-rec.simps)
      (simp add: Cons.IH state-list-S sum-list-distinct-conv-sum-set
        del: forward-ix-rec.simps forward.simps
      )
qed

```

Instructs the code generator to use this equation instead to execute *forward*. Uses the memoized version of *forward-ix*.

```

lemma (in HMM4) forward-code [code]:
  forward s t os = fst (run-state (forward-ixm' (IArray os) s t 0) Mapping.empty)
  by (simp only:
    forward-ix-def forward-ixm.memoized-correct forward-ix-forward[symmetric]
    states-distinct
  )

```

```

theorem (in HMM4) likelihood-compute:
  likelihood-compute s os = Some x  $\longleftrightarrow$  s  $\in$  S  $\wedge$  x = likelihood s os
  unfolding likelihood-compute-def
  by (auto simp: states-distinct state-list-S sum-list-distinct-conv-sum-set likelihood-forward)

```

## 2.2 The Viterbi Algorithm

```

context HMM3-defs
begin

```

```

context
  fixes os :: 't iarray
begin

```

Alternative definition using indices into the list of states. The list of states is implemented as an immutable array for better performance.

```

function viterbi-ix-rec where
  viterbi-ix-rec s t-end n = (if n  $\geq$  IArray.length os then ([], indicator {t-end} s) else
  fst (
    argmax snd (map
      ( $\lambda t$ . let (xs, v) = viterbi-ix-rec t t-end (n + 1) in
        (t # xs, ennreal (pmf (O t) (os !! n) * pmf (K s) t) * v)
        state-list)))

```

```

  by pat-completeness auto

```

```

termination
  by (relation Wellfounded.measure ( $\lambda(-, -, n)$ ). IArray.length os - n) auto

```

Memoization

```

memoize-fun viterbi-ixm: viterbi-ix-rec
  with-memory dp-consistency-mapping
  monadifies (state) viterbi-ix-rec.simps[unfolded Let-def]

```

```

memoize-correct

```

by *memoize-prover*

The main theorems generated by memoization.

**context**

includes *state-monad-syntax*

**begin**

**thm** *viterbi-ix<sub>m</sub>'*.simps *viterbi-ix<sub>m</sub>-def*

**thm** *viterbi-ix<sub>m</sub>*.memoized-correct

**end**

**end**

**definition**

*viterbi-ix os = viterbi-ix-rec (IArray os)*

**end**

**context** *HMM3*

**begin**

**lemma** *viterbi-ix-drop-one*:

*viterbi-ix (o # os) s t (n + 1) = viterbi-ix os s t n*

by (*induction length os - n arbitrary: s n; simp add: viterbi-ix-def*)

**lemma** *viterbi-ix-viterbi*:

*viterbi-ix os s t 0 = viterbi s t os*

**unfolding** *viterbi-ix-def*

**proof** (*induction os arbitrary: s*)

**case** *Nil*

**then show** *?case*

by *simp*

**next**

**case** (*Cons o os*)

**show** *?case*

using *viterbi-ix-drop-one*[*unfolded viterbi-ix-def*]

by (*subst viterbi.simps, subst viterbi-ix-rec.simps*)

(*simp add: Cons.IH del: viterbi-ix-rec.simps viterbi.simps*)

**qed**

**lemma** *viterbi-code* [*code*]:

*viterbi s t os = fst (run-state (viterbi-ix<sub>m</sub>' (IArray os) s t 0) Mapping.empty)*

by (*simp only: viterbi-ix-def viterbi-ix<sub>m</sub>.memoized-correct viterbi-ix-viterbi[symmetric]*)

**end**

## 2.3 Misc

**lemma** *pmf-of-alist-support-aux-1*:

**assumes**  $\forall (-, p) \in \text{set } \mu. p \geq 0$

**shows**  $(0 :: \text{real}) \leq (\text{case map-of } \mu \text{ of None} \Rightarrow 0 \mid \text{Some } p \Rightarrow p)$

**using** *assms* **by** (*auto split: option.split dest: map-of-SomeD*)

**lemma** *pmf-of-alist-support-aux-2*:

**assumes**  $\forall (-, p) \in \text{set } \mu. p \geq 0$

**and** *sum-list (map snd  $\mu$ ) = 1*

**and** *distinct (map fst  $\mu$ )*

**shows**  $\int^+ x. \text{ennreal } (\text{case map-of } \mu \text{ of None} \Rightarrow 0 \mid \text{Some } p \Rightarrow p) \partial \text{count-space UNIV} = 1$

**using** *assms*

**apply** (*subst nn-integral-count-space*)

**subgoal**

by (*rule finite-subset*[**where** *B = fst ' set  $\mu$* ];

```

    force split: option.split-asm simp: image-iff dest: map-of-SomeD)
apply (subst sum.mono-neutral-left[where  $T = \text{fst } \mu$ ])
  apply blast
subgoal
  by (smt ennreal-less-zero-iff map-of-eq-None-iff mem-Collect-eq option.case(1) subsetI)
subgoal
  by auto
subgoal premises prems
proof -
  have ( $\sum x = 0..<\text{length } \mu. \text{snd } (\mu ! x)$ )
    =  $\text{sum } (\lambda x. \text{case map-of } \mu x \text{ of None } \Rightarrow 0 \mid \text{Some } v \Rightarrow v) (\text{fst } \mu)$ 
  apply (rule sym)
  apply (rule sum.reindex-cong[where  $l = \lambda i. \text{fst } (\mu ! i)$ ])
    apply (auto split: option.split)
  subgoal
    using prems(3) by (intro inj-onI, auto simp: distinct-conv-nth)
  subgoal
    by (auto simp: in-set-conv-nth rev-image-eqI)
  subgoal
    by (simp add: map-of-eq-None-iff)
  subgoal
    using map-of-eq-Some-iff[OF prems(3)]
    by (metis fst-conv nth-mem option.inject prod-eqI snd-conv)
  done
with prems(2) show ?thesis
  by (smt pmf-of-alist-support-aux-1[OF assms(1)] atLeastLessThan-iff ennreal-1
    length-map nth-map sum.cong sum-ennreal sum-list-sum-nth
  )
qed
done

```

**lemma** pmf-of-alist-support:

```

assumes  $\forall (-, p) \in \text{set } \mu. p \geq 0$ 
  and  $\text{sum-list } (\text{map snd } \mu) = 1$ 
  and  $\text{distinct } (\text{map fst } \mu)$ 
shows  $\text{set-pmf } (\text{pmf-of-alist } \mu) \subseteq \text{fst } \mu$ 
unfolding pmf-of-alist-def
apply (subst set-embed-pmf)
subgoal for  $x$ 
  using assms(1) by (auto split: option.split dest: map-of-SomeD)
subgoal
  using pmf-of-alist-support-aux-2[OF assms] .
apply (force split: option.split-asm simp: image-iff dest: map-of-SomeD)+
done

```

Defining a Markov kernel from an association list.

```

locale Closed-Kernel-From =
  fixes  $K :: ('s \times ('t \times \text{real}) \text{ list}) \text{ list}$ 
    and  $S :: 't \text{ list}$ 
assumes wellformed:  $S \neq []$ 
  and closed:  $\forall (s, \mu) \in \text{set } K. \forall (t, -) \in \text{set } \mu. t \in \text{set } S$ 
  and is-pmf:
     $\forall (-, \mu) \in \text{set } K. \forall (-, p) \in \text{set } \mu. p \geq 0$ 
     $\forall (-, \mu) \in \text{set } K. \text{distinct } (\text{map fst } \mu)$ 
     $\forall (s, \mu) \in \text{set } K. \text{sum-list } (\text{map snd } \mu) = 1$ 
  and is-unique:
     $\text{distinct } (\text{map fst } K)$ 
begin

```

**definition**

```

 $K' s \equiv \text{case map-of } (\text{map } (\lambda (s, \mu). (s, \text{PMF-Impl.pmf-of-alist } \mu))) K) s \text{ of}$ 

```

*None*  $\Rightarrow$  *return-pmf* (*hd* *S*) |  
*Some* *s*  $\Rightarrow$  *s*

**sublocale** *Closed-Kernel* *K'* *set* *S*

**using** *wellformed* *closed* *is-pmf* *pmf-of-alist-support*

**unfolding** *K'-def* **by**  $-$  (*standard*; *fastforce* *split*: *option.split-asm* *dest*: *map-of-SomeD*)

**definition** [*code*]:

*K1* = *map-of* (*map* ( $\lambda$  (*s*,  $\mu$ ). (*s*, *map-of*  $\mu$ )) *K*)

**lemma** *pmf-of-alist-aux*:

**assumes** (*s*,  $\mu$ )  $\in$  *set* *K*

**shows**

*pmf* (*pmf-of-alist*  $\mu$ ) *t* = (*case* *map-of*  $\mu$  *t* of  
*None*  $\Rightarrow$  0

| *Some* *p*  $\Rightarrow$  *p*)

**using** *assms* *is-pmf* **unfolding** *pmf-of-alist-def*

**by** (*intro* *pmf-embed-pmf* *pmf-of-alist-support-aux-2*)

(*auto* 4 3 *split*: *option.split* *dest*: *map-of-SomeD*)

**lemma** *unique*:  $\mu = \mu'$  **if** (*s*,  $\mu$ )  $\in$  *set* *K* (*s*,  $\mu'$ )  $\in$  *set* *K*

**using** *that* *is-unique*

**by** (*smt* *Pair-inject* *distinct-conv-nth* *fst-conv* *in-set-conv-nth* *length-map* *nth-map*)

**lemma** (**in**  $-$ ) *map-of-NoneD*:

*x*  $\notin$  *fst* ' *set* *M* **if** *map-of* *M* *x* = *None*

**using** *that* **by** (*auto* *dest*: *weak-map-of-SomeI*)

**lemma** *K'-code* [*code-post*]:

*pmf* (*K'* *s*) *t* = (*case* *K1* *s* of  
*None*  $\Rightarrow$  (*if* *t* = *hd* *S* *then* 1 *else* 0)

| *Some*  $\mu$   $\Rightarrow$  *case*  $\mu$  *t* of

*None*  $\Rightarrow$  0

| *Some* *p*  $\Rightarrow$  *p*

)

**unfolding** *K'-def* *K1-def*

**apply** (*clarsimp* *split*: *option.split*, *safe*)

**apply** (*drule* *map-of-SomeD*, *drule* *map-of-NoneD*, *force*)+

**apply** (*fastforce* *dest*: *unique* *map-of-SomeD* *simp*: *pmf-of-alist-aux*)+

**done**

**end**

## 2.4 Executing Concrete HMMs

**locale** *Concrete-HMM-defs* =

**fixes** *K* :: ('*s*  $\times$  ('*s*  $\times$  *real*) *list*) *list*

**and** *O* :: ('*s*  $\times$  ('*t*  $\times$  *real*) *list*) *list*

**and** *O<sub>s</sub>* :: '*t* *list*

**and** *K<sub>s</sub>* :: '*s* *list*

**begin**

**definition**

*K'* *s*  $\equiv$  *case* *map-of* (*map* ( $\lambda$  (*s*,  $\mu$ ). (*s*, *PMF-Impl.pmf-of-alist*  $\mu$ )) *K*) *s* of

*None*  $\Rightarrow$  *return-pmf* (*hd* *K<sub>s</sub>*) |

*Some* *s*  $\Rightarrow$  *s*

**definition**

*O'* *s*  $\equiv$  *case* *map-of* (*map* ( $\lambda$  (*s*,  $\mu$ ). (*s*, *PMF-Impl.pmf-of-alist*  $\mu$ )) *O*) *s* of

*None*  $\Rightarrow$  *return-pmf* (*hd* *O<sub>s</sub>*) |

*Some* *s*  $\Rightarrow$  *s*



end

**locale** *Concrete-HMM* = *Concrete-HMM-defs* +  
  **assumes** *observations-wellformed'*:  $\mathcal{O}_s \neq []$   
  **and** *observations-closed'*:  $\forall (s, \mu) \in \text{set } \mathcal{O}. \forall (t, -) \in \text{set } \mu. t \in \text{set } \mathcal{O}_s$   
  **and** *observations-form-pmf'*:  
     $\forall (-, \mu) \in \text{set } \mathcal{O}. \forall (-, p) \in \text{set } \mu. p \geq 0$   
     $\forall (-, \mu) \in \text{set } \mathcal{O}. \text{distinct } (\text{map } \text{fst } \mu)$   
     $\forall (s, \mu) \in \text{set } \mathcal{O}. \text{sum-list } (\text{map } \text{snd } \mu) = 1$   
  **and** *observations-unique*:  
     $\text{distinct } (\text{map } \text{fst } \mathcal{O})$   
  **assumes** *states-wellformed*:  $\mathcal{K}_s \neq []$   
  **and** *states-closed*:  $\forall (s, \mu) \in \text{set } \mathcal{K}. \forall (t, -) \in \text{set } \mu. t \in \text{set } \mathcal{K}_s$   
  **and** *states-form-pmf*:  
     $\forall (-, \mu) \in \text{set } \mathcal{K}. \forall (-, p) \in \text{set } \mu. p \geq 0$   
     $\forall (-, \mu) \in \text{set } \mathcal{K}. \text{distinct } (\text{map } \text{fst } \mu)$   
     $\forall (s, \mu) \in \text{set } \mathcal{K}. \text{sum-list } (\text{map } \text{snd } \mu) = 1$   
  **and** *states-unique*:  
     $\text{distinct } (\text{map } \text{fst } \mathcal{K}) \text{ distinct } \mathcal{K}_s$

**begin**

**interpretation** *O*: *Closed-Kernel-From*  $\mathcal{O} \mathcal{O}_s$

**rewrites**  $O.K' = \mathcal{O}'$

**proof** –

**show**  $\langle \text{Closed-Kernel-From } \mathcal{O} \mathcal{O}_s \rangle$

**using** *observations-wellformed'* *observations-closed'* *observations-form-pmf'* *observations-unique*  
    **by** *unfold-locales auto*

**show**  $\langle \text{Closed-Kernel-From}.K' \mathcal{O} \mathcal{O}_s = \mathcal{O}' \rangle$

**unfolding** *Closed-Kernel-From.K'-def*[*OF*  $\langle \text{Closed-Kernel-From } \mathcal{O} \mathcal{O}_s \rangle$ ] *O'-def*  
    **by** *auto*

**qed**

**interpretation** *K*: *Closed-Kernel-From*  $\mathcal{K} \mathcal{K}_s$

**rewrites**  $K.K' = \mathcal{K}'$

**proof** –

**show**  $\langle \text{Closed-Kernel-From } \mathcal{K} \mathcal{K}_s \rangle$

**using** *states-wellformed* *states-closed* *states-form-pmf* *states-unique* **by** *unfold-locales auto*

**show**  $\langle \text{Closed-Kernel-From}.K' \mathcal{K} \mathcal{K}_s = \mathcal{K}' \rangle$

**unfolding** *Closed-Kernel-From.K'-def*[*OF*  $\langle \text{Closed-Kernel-From } \mathcal{K} \mathcal{K}_s \rangle$ ] *K'-def*  
    **by** *auto*

**qed**

**lemmas**  $O\text{-code} = O.K'\text{-code } O.K1\text{-def}$

**lemmas**  $K\text{-code} = K.K'\text{-code } K.K1\text{-def}$

**sublocale** *HMM-interp*: *HMM4*  $\mathcal{O}' \text{ set } \mathcal{K}_s \mathcal{K}_s \text{ set } \mathcal{O}_s \mathcal{K}'$

**using** *O.Closed-Kernel-axioms* *K.Closed-Kernel-axioms* *states-unique(2)*

**by** (*intro-locales*; *intro HMM4-axioms.intro HMM3-axioms.intro HOL.refl*)

end

end

### 3 Example

**theory** *HMM-Example*

**imports**

*HMM-Implementation*

*HOL-Library.AList-Mapping*

**begin**

We would like to implement mappings as red-black trees but they require the key type to be linearly ordered. Unfortunately, *HOL-Analysis* fixes the product order to the element-wise order and thus we cannot restore a linear order, and the red-black tree implementation (from *HOL-Library*) cannot be used.

The ice cream example from Jurafsky and Martin [2].

**definition**

```
states = ["start", "hot", "cold", "end"]
```

**definition** *observations* :: int list **where**

```
observations = [0, 1, 2, 3]
```

**definition**

```
kernel =  
[  
  ("start", [(("hot", 0.8 :: real), ("cold", 0.2))],  
  ("hot", [(("hot", 0.6 :: real), ("cold", 0.3), ("end", 0.1))],  
  ("cold", [(("hot", 0.4 :: real), ("cold", 0.5), ("end", 0.1))],  
  ("end", [(("end", 1)])  
]
```

**definition**

```
emissions =  
[  
  ("hot", [(1, 0.2), (2, 0.4), (3, 0.4)]),  
  ("cold", [(1, 0.5), (2, 0.4), (3, 0.1)]),  
  ("end", [(0, 1)])  
]
```

**global-interpretation** *Concrete-HMM kernel emissions observations states*

**defines**

```
viterbi-rec = HMM-interp.viterbi-ixm'  
and viterbi = HMM-interp.viterbi  
and viterbi-final = HMM-interp.viterbi-final  
and forward-rec = HMM-interp.forward-ixm'  
and forward = HMM-interp.forward  
and likelihood = HMM-interp.likelihood-compute  
by (standard; eval)
```

**lemmas** [code] = HMM-interp.viterbi-ix<sub>m</sub>'.simps[unfolded O-code K-code]

**lemmas** [code] = HMM-interp.forward-ix<sub>m</sub>'.simps[unfolded O-code K-code]

**value** *likelihood* "start" [1, 1, 1]

If we enforce the last observation to correspond to "end", then *forward* and *likelihood* yield the same result.

**value** *likelihood* "start" [1, 1, 1, 0]

**value** *forward* "start" "end" [1, 1, 1, 0]

**value** *forward* "start" "end" [3, 3, 3, 0]

**value** *forward* "start" "end" [3, 1, 3, 0]

**value** *forward* "start" "end" [3, 1, 3, 1, 0]

```
value viterbi "start" "end" [1, 1, 1, 0]
value viterbi "start" "end" [3, 3, 3, 0]
value viterbi "start" "end" [3, 1, 3, 0]
value viterbi "start" "end" [3, 1, 3, 1, 0]
```

If we enforce the last observation to correspond to "end", then *viterbi* and *viterbi-final* yield the same result.

```
value viterbi-final "start" [3, 1, 3, 1, 0]
value viterbi-final "start" [1, 1, 1, 1, 1, 1, 1, 0]
value viterbi-final "start" [1, 1, 1, 1, 1, 1, 1, 1]
end
```

## References

- [1] J. Hölzl. Markov chains and Markov decision processes in Isabelle/HOL. *Journal of Automated Reasoning*, 2017.
- [2] D. Jurafsky and J. H. Martin. *Speech and language processing*. 2017.
- [3] A. A. Markov. Essai d'une recherche statistique sur le texte du roman "Eugene Onegin" illustrant la liaison des epreuve en chain ('Example of a statistical investigation of the text of "Eugene Onegin" illustrating the dependence between samples in chain'). *Izvestia Imperatorskoi Akademii Nauk (Bulletin de l'Académie Impériale des Sciences de St.-Pétersbourg)*, 7:153–162, 1913. English translation by Morris Halle, 1956.