Hidden Markov Models

Simon Wimmer

March 17, 2025

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

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 :: $'s \Rightarrow 't$ *pmf* **and** S :: $'t$ *set*
  assumes *finite*: *finite* S
    and *wellformed*: $S \neq \{\}$
    and *closed*: $\forall\ s.\ K\ s \subseteq S$

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

locale *HMM-defs* =
  fixes $\mathcal{K}$ :: $'s \Rightarrow 's$ *pmf* **and** $\mathcal{O}$ :: $'s \Rightarrow 't$ *pmf* **and** $\mathcal{O}_s$ :: $'t$ *set*

locale *HMM* =
  HMM-defs + *O*: *Closed-Kernel* $\mathcal{O}$ $\mathcal{O}_s$
begin

lemma *observations-finite*: *finite* $\mathcal{O}_s$
  and *observations-wellformed*: $\mathcal{O}_s \neq \{\}$
  and *observations-closed*: $\forall\ s.\ \mathcal{O}\ s \subseteq \mathcal{O}_s$
  using *O.finite O.wellformed O.closed* **by** −

end

Fixed set of internal states.

locale *HMM2-defs* = *HMM-defs* $\mathcal{K}$ $\mathcal{O}$ **for** $\mathcal{K}$ :: $'s \Rightarrow 's$ *pmf* **and** $\mathcal{O}$ :: $'s \Rightarrow 't$ *pmf* +
  fixes $\mathcal{S}$ :: $'s$ *set*

locale *HMM2* = *HMM2-defs* + *HMM* + *K*: *Closed-Kernel* $\mathcal{K}$ $\mathcal{S}$
begin

lemma *states-finite*: *finite* $\mathcal{S}$
  and *states-wellformed*: $\mathcal{S} \neq \{\}$
  and *states-closed*: $\forall\ s.\ \mathcal{K}\ s \subseteq \mathcal{S}$
  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* $\mathcal{O}_s$ $\mathcal{K}$ **for** $\mathcal{O}_s$ :: $'t$ *set* **and** $\mathcal{K}$ :: $'s \Rightarrow 's$ *pmf* +
  fixes *state-list* :: $'s$ *list*

locale *HMM3* = *HMM3-defs* - - $\mathcal{O}_s$ $\mathcal{K}$ + *HMM2* $\mathcal{O}_s$ $\mathcal{K}$ **for** $\mathcal{O}_s$ :: $'t$ *set* **and** $\mathcal{K}$ :: $'s \Rightarrow 's$ *pmf* +
  assumes *state-list-*$\mathcal{S}$: *set state-list* = $\mathcal{S}$

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*: $likelihood\ s\ os = T'\ (I\ s)\ \{(s,\ o)\ \#\#\ xs\ @-\ \omega'\ |o\ xs\ \omega'.\ map\ snd\ xs = os\}$
  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* ω; *cases shd* ω; *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 ω* ⟷ *fst* (*shd ω*) = *s* ∧ *L os* (*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 *T-init-observation-eq*:
T (*s*, *o*) {*ω* ∈ *space S*. *L os ω*} = *T* (*s*, *o′*) {*ω* ∈ *space S*. *L os 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

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*) {*ω* ∈ *space S*. *L os ω*}
proof −
have ∗: ($\sum$ *o*∈$\mathcal{O}_s$. *emeasure* (*T* (*s*, *o*)) {*ω* ∈ *space S*. *L os ω*}) =
of-nat (*card* $\mathcal{O}_s$) ∗ *emeasure* (*T* (*s*, *obs*)) {*ω* ∈ *space S*. *L os ω*}
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*, o_0) {*ω* ∈ *space S*. *L* (o_1 # *os*) *ω*} =
($\int^+$ *t*. *ennreal* (*pmf* ($\mathcal{O}$ *t*) o_1) ∗ *T* (*t*, o_1) {*ω* ∈ *space S*. *L os ω*} ∂($\mathcal{K}$ *s*)) (**is** *?l* = *?r*)
proof −
have ∗:
$\int^+$ *y*. *T* (*s′*, *y*) {*x* ∈ *space S*. ∃*xs*. (∃*ω′*. (*s′*, *y*) ## *x* = *xs* @− *ω′*) ∧ *map snd xs* = o_1 # *os*}
∂*measure-pmf* ($\mathcal{O}$ *s′*) =
ennreal (*pmf* ($\mathcal{O}$ *s′*) o_1) ∗ *T* (*s′*, o_1) {*ω* ∈ *space S*. ∃*xs*. (∃*ω′*. *ω* = *xs* @− *ω′*) ∧ *map snd xs* = *os*}
(**is** *?L* = *?R*) **for** *s′*
proof −
have *?L* = $\int^+$ *x*. *ennreal* (*pmf* ($\mathcal{O}$ *s′*) *x*) ∗
T (*s′*, *x*) {*ω* ∈ *space S*. ∃*xs*. (∃*ω′*. (*s′*, *x*) ## *ω* = *xs* @− *ω′*) ∧ *map snd xs* = o_1 # *os*}
∂*count-space UNIV*
by (*rule nn-integral-measure-pmf*)
also have … =
$\int^+$ o_2. (*if* o_2 = o_1
then ennreal (*pmf* ($\mathcal{O}$ *s′*) o_1) ∗ *T* (*s′*, o_1) {*ω* ∈ *space S*. *L os ω*}
else 0)
∂*count-space UNIV*
apply (*rule nn-integral-cong-AE*
[**where** *v* = λ o_2. *if* o_2 = o_1
then ennreal (*pmf* ($\mathcal{O}$ *s′*) o_1) ∗ *T* (*s′*, o_1) {*ω* ∈ *space S*. *L os ω*} *else 0*]
)
apply (*rule AE-I2*)

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 ... $= (\int^+ o_2 \in \{o_1\}.\ (ennreal\ (pmf\ (\mathcal{O}\ s')\ o_1) * T\ (s', o_1)\ \{\omega \in space\ S.\ L\ os\ \omega\})$
$\partial count\text{-}space\ UNIV)$
by (*rule nn-integral-cong-AE*) *auto*
also have ... = *?R*
by *simp*
finally show *?thesis* **.**
qed
have $?l = \int^+ t.\ T\ t\ \{x \in space\ S.\ \exists xs\ \omega'.\ t\ \#\#\ x = xs\ @-\ \omega' \wedge map\ snd\ xs = o_1\ \#\ os\}\ \partial\ (K\ (s, o_0))$
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*) $= (\int^+ t.\ ennreal\ (pmf\ (\mathcal{O}\ t)\ o) * backward\ t\ os\ \partial measure\text{-}pmf\ (\mathcal{K}\ s))$

lemma *emeasure-T-observation-backward*:
emeasure $(T\ (s, o))\ \{\omega \in space\ S.\ L\ os\ \omega\}$ = *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*) =
$(\sum t \in \mathcal{S}.\ ennreal\ (pmf\ (\mathcal{O}\ t)\ o) * ennreal\ (pmf\ (\mathcal{K}\ s)\ t) * forward\ t\ t\text{-}end\ os)$

lemma *forward-split*:
forward s t (*os1* @ *os2*) $= (\sum t' \in \mathcal{S}.\ forward\ s\ t'\ os1 * forward\ t'\ t\ os2)$
if $s \in \mathcal{S}$
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** −)
(∑ *t* ∈ *S*. *f t*) = *f t* **if** *finite S t* ∈ *S* ∀ *s* ∈ *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*:
(∑ *t* ∈ 𝒮. *forward s t os*) = *backward s os* **if** *s* ∈ 𝒮
using ‹*s* ∈ 𝒮›
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 = (∑ *t* ∈ 𝒮. *forward s t os*) **if** ‹*s* ∈ 𝒮›
unfolding *likelihood-backward forward-backward*[*symmetric*, *OF* ‹*s* ∈ 𝒮›] **..**

1.4 Definition of Maximum Probabilities

abbreviation (*input*) *V os as* ω ≡ (∃ ω′. ω = *zip as os* @− ω′)

definition
max-prob s os =
Max {*T*′ (*I s*) {ω ∈ *space S*. ∃ *o* ω′. ω = (*s*, *o*) ## *zip as os* @− ω′}
| *as*. *length as* = *length os* ∧ *set as* ⊆ 𝒮}

fun *viterbi-prob* **where**
viterbi-prob s t-end [] = *indicator* {*t-end*} *s* |
viterbi-prob s t-end (*o* # *os*) =
(*MAX t* ∈ 𝒮. *ennreal* (*pmf* (𝒪 *t*) *o* ∗ *pmf* (𝒦 *s*) *t*) ∗ *viterbi-prob t t-end os*)

definition
is-decoding s os as ≡
T′ (*I s*) {ω ∈ *space S*. ∃ *o* ω′. ω = (*s*, *o*) ## *zip as os* @− ω′} = *max-prob s os* ∧
length as = *length os* ∧ *set as* ⊆ 𝒮

1.5 Iteration Rule For Maximum Probabilities

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

lemma *max-prob-T-state-Nil*:
Max {*T* (*s*, *o*) {ω ∈ *space S*. *V* [] *as* ω} | *as*. *length as* = *length* [] ∧ *set as* ⊆ 𝒮} = *1*
by (*simp add*: *emeasure-T-state-Nil*)

lemma *V-Cons*: *V* (*o* # *os*) (*a* # *as*) ω ⟷ *fst* (*shd* ω) = *a* ∧ *snd* (*shd* ω) = *o* ∧ *V os as* (*stl* ω)
by (*cases* ω) *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 = λt as os. T t {ω ∈ space S. V os as (t ## ω)}
  let ?g = λt 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 ... $= (MAX\ as \in\ ?S\ (o\ \#\ os).\ \int^{+}\ t.\ ?g\ t\ (tl\ as)\ os * indicator\ \{(hd\ as,o)\}\ t\ \partial K\ (s,o_1))$
by (*simp cong*: *Max-image-cong-simp add*: $*$)
also have ... $= (MAX(t,\ as) \in \mathcal{S} \times\ ?S\ os.\ ennreal\ (pmf\ (\mathcal{O}\ t)\ o * pmf\ (\mathcal{K}\ s)\ t) *\ ?g\ (t,\ o)\ as\ os)$
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 $?l' = \int^{+}\ t'.\ ?f\ t'\ (t\ \#\ as)\ (o\ \#\ os)\ \partial K\ (s,\ o_1)$
by (*rule emeasure-Collect-T*[*unfolded space-T*]; *rule measurable-V*)
also from *that* **have** ... $= \int^{+}\ t'.\ ?g\ t'\ as\ os * indicator\ \{(t,o)\}\ t'\ \partial K\ (s,o_1)$
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*:
$T\ (s,\ o)\ \{\omega \in space\ S.\ V\ os\ as\ \omega\} = T\ (s,\ o')\ \{\omega \in space\ S.\ V\ os\ as\ \omega\}$
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

lemma *T′-I-T*:
$T'\ (I\ s)\ \{\omega \in space\ S.\ \exists\, o\ \omega'.\ \omega = (s,\ o)\ \#\#\ zip\ as\ os\ @-\ \omega'\} = T\ (s,o)\ \{\omega \in space\ S.\ V\ os\ as\ \omega\}$
proof −
have $(\sum o{\in}\mathcal{O}_s.\ T\ (s,\ o)\ \{\omega \in space\ S.\ V\ os\ as\ \omega\}) =$
$of\text{-}nat\ (card\ \mathcal{O}_s) * T\ (s,\ o)\ \{\omega \in space\ S.\ V\ os\ as\ \omega\}$ **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 = Max {T (s,o) {ω ∈ space S. V os as ω} | as. length as = length os ∧ set as ⊆ 𝒮}
  unfolding max-prob-def by (simp add: T'-I-T[symmetric])

lemma max-prob-Nil[simp]:
  max-prob s [] = 1
  unfolding max-prob-init[where o = obs] by auto

lemma Max-start:
  (MAX t∈𝒮. (indicator {t} s :: ennreal)) = 1 if s ∈ 𝒮
  using states-finite that by (auto simp: indicator-def intro: Max-eqI)

lemma Max-V-viterbi:
  (MAX t ∈ 𝒮. viterbi-prob s t os) =
   Max {T (s, o) {ω ∈ space S. V os as ω} | as. length as = length os ∧ set as ⊆ 𝒮} if s ∈ 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:
  (MAX t ∈ 𝒮. viterbi-prob s t os) = max-prob s os if s ∈ 𝒮
  using max-prob-init[of s os] Max-V-viterbi[OF ‹s ∈ 𝒮›, symmetric] by simp

end
```

1.7 Decoding the Most Probable Hidden State Sequence

```
context HMM3
begin

fun viterbi where
  viterbi s t-end [] = ([], indicator {t-end} s) |
  viterbi s t-end (o # os) = fst (
    argmax snd (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))

lemma state-list-nonempty:
  state-list ≠ []
  using state-list-𝒮 states-wellformed by auto

lemma viterbi-viterbi-prob:
  snd (viterbi s t-end os) = viterbi-prob s t-end os
proof (induction os arbitrary: s)
  case Nil
  then show ?case
    by simp
next
  case (Cons o os)
  let ?f =
    λt. let (xs, v) = viterbi t t-end os in (t # xs, ennreal (pmf (𝒪 t) o * pmf (𝒦 s) t) * v)
  let ?xs = map ?f state-list
  from state-list-nonempty have map ?f state-list ≠ []
    by simp
  from argmax(2,3)[OF this, of snd] have *:
    snd (fst (argmax snd ?xs)) = snd (argmax snd ?xs)
    snd (argmax snd ?xs) = (MAX x ∈ set ?xs. 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 ≠ []
```

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-end os*), *ennreal* (*pmf* ($\mathcal{O}$ *t*) *o* ∗ *pmf* ($\mathcal{K}$ *s*) *t*) ∗ *snd* (*viterbi t t-end os*))
by (*auto simp*: *state-list-*$\mathcal{S}$ *split*: *prod.splits*)
with *Cons.prems* **show** *?case*
by (*cases viterbi t t-end os*; *simp add*: *Cons.IH*)
qed

definition
viterbi-final s os = *fst* (*argmax snd* (*map* (λ *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* ((λ*t*. *snd* (*viterbi s t os*)) ‘ $\mathcal{S}$)
proof −
from *state-list-nonempty* **have** *map* (λ *t*. *viterbi s t os*) *state-list* ≠ []
by *simp*
from *argmax*[*OF this*, *of snd*] **show** *?thesis*
by (*auto simp*: *state-list-*$\mathcal{S}$ *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-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* ((λ*t*. *snd* (*viterbi s t os*)) ‘ $\mathcal{S}$)
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* ⊆ $\mathcal{S}$
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-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-prob s os* = *T* (*s*, *obs*) {ω ∈ *space S*. *V os as* ω}
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 set* **and** $\mathcal{K}$ :: *'s* ⇒ *'s 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* ≥ *IArray.length os then indicator* {*t-end*} *s else*
(∑ *t* ← *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* (λ(-,-,*n*). *IArray.length os* − *n*)) *auto*

Memoization

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

The main theorems generated by memoization.

context
includes *state-monad-syntax*
begin
thm *forward-ix*$_m$*'.simps forward-ix*$_m$*-def*
thm *forward-ix*$_m$*.memoized-correct*
end

end

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

definition
likelihood-compute s os ≡
if s ∈ *set state-list then Some* (∑ *t* ← *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-$\mathcal{S}$ 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-ix$_m$'* (*IArray os*) *s t 0*) *Mapping.empty*)
by (*simp only*:
forward-ix-def forward-ix$_m$.memoized-correct forward-ix-forward[*symmetric*]
states-distinct
)

theorem (**in** *HMM4*) *likelihood-compute*:
likelihood-compute s os = *Some x* $\longleftrightarrow$ *s* $\in \mathcal{S} \wedge$ *x* = *likelihood s os*
unfolding *likelihood-compute-def*
by (*auto simp*: *states-distinct state-list-$\mathcal{S}$ 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*
(λ*t*. *let* (*xs*, *v*) = *viterbi-ix-rec t t-end* (*n* + *1*) *in*
(*t* # *xs*, *ennreal* (*pmf* ($\mathcal{O}$ *t*) (*os* !! *n*) * *pmf* ($\mathcal{K}$ *s*) *t*) * *v*))
state-list)))

by *pat-completeness auto*
termination
by (*relation Wellfounded.measure* (λ(-,-,*n*). *IArray.length os* − *n*)) *auto*

Memoization

memoize-fun *viterbi-ix$_m$*: *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*$_m$*'.simps viterbi-ix*$_m$*-def*
thm *viterbi-ix*$_m$*.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*$_m$*'* (*IArray os*) *s t 0*) *Mapping.empty*)
  by (*simp only*: *viterbi-ix-def viterbi-ix*$_m$*.memoized-correct viterbi-ix-viterbi*[*symmetric*])

end

2.3 Misc

lemma *pmf-of-alist-support-aux-1*:
  assumes ∀ (-, *p*) ∈ *set* μ. *p* ≥ *0*
  shows (*0* :: *real*) ≤ (*case map-of* μ *x of None* ⇒ *0* | *Some p* ⇒ *p*)
  using *assms* **by** (*auto split*: *option.split dest*: *map-of-SomeD*)

lemma *pmf-of-alist-support-aux-2*:
  assumes ∀ (-, *p*) ∈ *set* μ. *p* ≥ *0*
    and *sum-list* (*map snd* μ) = *1*
    and *distinct* (*map fst* μ)
  shows $\int^{+}$ *x*. *ennreal* (*case map-of* μ *x of None* ⇒ *0* | *Some p* ⇒ *p*) ∂*count-space UNIV* = *1*
  using *assms*
  apply (*subst nn-integral-count-space*)
  subgoal
    by (*rule finite-subset*[**where** *B* = *fst* ‘ *set* μ];
        force split: *option.split-asm simp*: *image-iff dest*: *map-of-SomeD*)

```
  apply (subst sum.mono-neutral-left[where T = fst ‘ set μ])
     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 (∑x = 0..<length μ. snd (μ ! x))
      = sum (λ x. case map-of μ x of None ⇒ 0 | Some v ⇒ v) (fst ‘ set μ)
      apply (rule sym)
      apply (rule sum.reindex-cong[where l = λ i. fst (μ ! 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 ∀ (-, p) ∈ set μ. p ≥ 0
    and sum-list (map snd μ) = 1
    and distinct (map fst μ)
  shows set-pmf (pmf-of-alist μ) ⊆ fst ‘ set μ
  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 × ('t × real) list) list
    and S :: 't list
  assumes wellformed: S ≠ []
      and closed: ∀ (s, μ) ∈ set K. ∀ (t, -) ∈ set μ. t ∈ set S
      and is-pmf:
        ∀ (-, μ) ∈ set K. ∀ (-, p) ∈ set μ. p ≥ 0
        ∀ (-, μ) ∈ set K. distinct (map fst μ)
        ∀ (s, μ) ∈ set K. sum-list (map snd μ) = 1
      and is-unique:
        distinct (map fst K)
begin

definition
  K' s ≡ case map-of (map (λ (s, μ). (s, PMF-Impl.pmf-of-alist μ)) K) s of
  None ⇒ return-pmf (hd S) |
```

Some s ⇒ 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* (λ (*s*, μ). (*s*, *map-of* μ)) *K*)

lemma *pmf-of-alist-aux*:
  assumes (*s*, μ) ∈ *set K*
  shows
    pmf (*pmf-of-alist* μ) *t* = (*case map-of* μ *t of*
      None ⇒ *0*
    | *Some p* ⇒ *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*: μ = μ′ **if** (*s*, μ) ∈ *set K* (*s*, μ′) ∈ *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 ∉ *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 ⇒ (*if t* = *hd S then 1 else 0*)
    | *Some* μ ⇒ *case* μ *t of*
        None ⇒ *0*
      | *Some p* ⇒ *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 $\mathcal{K}$ :: (*′s* × (*′s* × *real*) *list*) *list*
    and $\mathcal{O}$ :: (*′s* × (*′t* × *real*) *list*) *list*
    and $\mathcal{O}_s$ :: *′t list*
    and $\mathcal{K}_s$ :: *′s list*
begin

definition
  $\mathcal{K}'$ *s* ≡ *case map-of* (*map* (λ (*s*, μ). (*s*, *PMF-Impl.pmf-of-alist* μ)) $\mathcal{K}$) *s of*
    None ⇒ *return-pmf* (*hd* $\mathcal{K}_s$) |
    Some s ⇒ *s*

definition
  $\mathcal{O}'$ *s* ≡ *case map-of* (*map* (λ (*s*, μ). (*s*, *PMF-Impl.pmf-of-alist* μ)) $\mathcal{O}$) *s of*
    None ⇒ *return-pmf* (*hd* $\mathcal{O}_s$) |
    Some s ⇒ *s*

end

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

interpretation *O*: *Closed-Kernel-From* $\mathcal{O}$ $\mathcal{O}_s$
 rewrites *O.K'* = $\mathcal{O}'$
proof −
 show ‹*Closed-Kernel-From* $\mathcal{O}$ $\mathcal{O}_s$›
  using *observations-wellformed' observations-closed' observations-form-pmf' observations-unique*
  by *unfold-locales auto*
 show ‹*Closed-Kernel-From.K'* $\mathcal{O}$ $\mathcal{O}_s$ = $\mathcal{O}'$›
  unfolding *Closed-Kernel-From.K'-def*[*OF* ‹*Closed-Kernel-From* $\mathcal{O}$ $\mathcal{O}_s$›] $\mathcal{O}'$*-def*
  by *auto*
qed

interpretation *K*: *Closed-Kernel-From* $\mathcal{K}$ $\mathcal{K}_s$
 rewrites *K.K'* = $\mathcal{K}'$
proof −
 show ‹*Closed-Kernel-From* $\mathcal{K}$ $\mathcal{K}_s$›
  using *states-wellformed states-closed states-form-pmf states-unique* **by** *unfold-locales auto*
 show ‹*Closed-Kernel-From.K'* $\mathcal{K}$ $\mathcal{K}_s$ = $\mathcal{K}'$›
  unfolding *Closed-Kernel-From.K'-def*[*OF* ‹*Closed-Kernel-From* $\mathcal{K}$ $\mathcal{K}_s$›] $\mathcal{K}'$*-def*
  by *auto*
qed

lemmas *O-code* = *O.K'-code O.K1-def*
lemmas *K-code* = *K.K'-code K.K1-def*

sublocale *HMM-interp*: *HMM4* $\mathcal{O}'$ *set* $\mathcal{K}_s$ $\mathcal{K}_s$ *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-ix_m'
  and viterbi       = HMM-interp.viterbi
  and viterbi-final = HMM-interp.viterbi-final
  and forward-rec   = HMM-interp.forward-ix_m'
  and forward       = HMM-interp.forward
  and likelihood    = HMM-interp.likelihood-compute
  by (standard; eval)

lemmas [code] = HMM-interp.viterbi-ix_m'.simps[unfolded O-code K-code]

lemmas [code] = HMM-interp.forward-ix_m'.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'). *Izvistia Imperatorskoi Akademii Nauk (Bulletin de l'Académie Impériale des Sciences de St.-Pétersbourg)*, 7:153–162, 1913. English translation by Morris Halle, 1956.