Formalization of 3-independence of simple tabulation hashing

Wei De Leong and Yong Kiam Tan and Seng Joe Watt

July 14, 2026

Abstract

Simple tabulation hashing [2] is a computationally-efficient hashing algorithm to get values that behave independently and are uniformly distributed for any 3 distinct keys. This entry formalizes the 3-independence and non-4-independence of simple tabulation hashing, based on the written proofs provided by [1, Solutions 8][4]

Contents

theory *Xor*
imports
Main
begin

This theory abstractly defines

- the binary operator *xor* (aliases (*XOR*) ($\oplus$))
- the n-ary version *xor-fold*
- and the Big version *xor-sum* (alias $\bigoplus i{\in}J.\ f\ i$).

An implementation of xor should abide by the following laws:

- identity element
- commutative
- associative
- left/right neutrality
- self-inverse

which constitutes an abelian group.
Instantiations of abel_group_xor have been provided for bool, nat and int.
Other theories relating to xor:

- *abstract-boolean-algebra-sym-diff*
- *xor*
- (AFP) `Approximate_Model_Counting.RandomXOR` [3]

1 Preliminaries

1.1 The XOR operator

lemma (**in** *monoid-list*) *list-conv-take-drop*:
shows $F\ xs = F\ (take\ i\ xs) * F\ (drop\ i\ xs)$
by (*metis append append-take-drop-id*)

lemma (**in** *monoid-list*) *list-conv-take-nth-drop*:
assumes $i < length\ xs$
shows $F\ xs = F\ (take\ i\ xs) * xs\ !\ i * F\ (drop\ (Suc\ i)\ xs)$
by (*metis Cons-nth-drop-Suc append append-take-drop-id assms assoc local.Cons*)

lemma (**in** *comm-monoid-list*) *absorb-right*:
assumes $i < length\ xs$
shows $F\ xs * v = F\ (xs[i := xs\ !\ i * v])$
by (*smt* (*verit*, *del-insts*) *append assms assoc commute list-conv-take-nth-drop local.Cons*
upd-conv-take-nth-drop)

lemma (**in** *semiring-bit-operations*) *eq-iff*:
shows $semiring\text{-}bit\text{-}operations\text{-}class.xor\ a\ b = 0 \longleftrightarrow a = b$
by (*metis local.xor.assoc local.xor.left-neutral local.xor-self-eq*)

1.1.1 Definition of XOR

class *xor* =
fixes *xor* :: ‹'a ⇒ 'a ⇒ 'a› (**infixr** ‹*XOR*› *59*) — Definition of 2-ary xor

class *abel-group-xor* = *zero* + *xor* +
assumes *xor-commute*: $a\ XOR\ b = b\ XOR\ a$
assumes *xor-assoc*: $(a\ XOR\ b)\ XOR\ c = a\ XOR\ b\ XOR\ c$
assumes *xor-right-neutral*: $a\ XOR\ 0 = a$
assumes *eq-iff* [*iff*]: $a\ XOR\ b = 0 \longleftrightarrow a = b$
begin

lemma *self-inv* [*simp*]:
shows $a\ XOR\ a = 0$
by *simp*

The above properties show that XOR forms an abelian group

sublocale *group xor 0 id*
using *xor-commute xor-assoc* **by** *unfold-locales* (*simp-all add*: *xor-right-neutral*)

sublocale *abel-semigroup xor* **by** *unfold-locales* (*simp add*: *xor-commute*)

sublocale *comm-monoid xor 0* **by** *unfold-locales simp*

sublocale *xor-sum*: *comm-monoid-set xor 0*
defines *xor-sum* = *xor-sum.F* **..**

abbreviation *Xor-Sum* ≡ *xor-sum* ($\lambda x.\ x$)

sublocale *xor-list*: *monoid-list xor 0*
defines *xor-fold* = *xor-list.F* **..** — Definition of n-ary xor

sublocale *xor-comm-list*: *comm-monoid-list xor 0*
rewrites *monoid-list.F xor 0* = *xor-fold*
proof −
show *comm-monoid-list xor 0* **..**
then interpret *comm-monoid-list xor 0* **.**
from *xor-fold-def* **show** *monoid-list.F xor 0* = *xor-fold* **by** *simp*

qed

sublocale *xor-list*: *comm-monoid-list-set xor 0*
  rewrites *monoid-list.F xor 0* = *xor-fold* **and** *comm-monoid-set.F xor 0* = *xor-sum*
proof −
  show *comm-monoid-list-set xor 0* **..**
  then interpret *xor-sum*: *comm-monoid-list-set xor 0* **.**
  from *xor-fold-def* **show** *monoid-list.F xor 0* = *xor-fold* **by** *simp*
  from *xor-sum-def* **show** *comm-monoid-set.F xor 0* = *xor-sum* **by** (*auto intro*: *sym*)
qed

lemma *xor-cong*:
  assumes *a XOR b* = *0*
  shows *a* = *b*
  using *assms eq-iff* **by** *blast*

lemma *inj-on-UNIV*:
  shows *inj-on xor UNIV*
  by (*rule inj-onI*) (*metis right-neutral*)

lemma *map-indices-conv-list-update-conv*:
  shows *map* (λ*j*. *if j* = *i then g j else xs* ! *j*) [*0*..<*length xs*] = *xs*[*i* := *g i*]
  by (*fastforce intro*: *nth-equalityI*)

lemma *fold-absorb*:
  assumes *i* < *length xs*
  shows *xor-fold xs XOR v* =
    xor-fold (*map* (λ*j*. *if j* = *i then xs* ! *j XOR v else xs* ! *j*) [*0*..<*length xs*])
  unfolding *map-indices-conv-list-update-conv*
  using *xor-comm-list.absorb-right*[*OF assms*] **.**
end

hide-fact *xor-commute xor-assoc xor-right-neutral*

1.1.2 Standard Instantiations

instantiation *bool* :: *abel-group-xor* **begin**
definition *zero-bool* ≡ *False*
definition *xor-bool* :: *bool* ⇒ *bool* ⇒ *bool* **where** *xor-bool a b* ≡ (*a* ≠ *b*)
instance by *standard* (*auto simp add*: *xor-bool-def zero-bool-def*)
end

instantiation *nat* **and** *int* :: *abel-group-xor* **begin**

definition *xor-nat* :: *nat* ⇒ *nat* ⇒ *nat* **where** *xor-nat* ≡ *semiring-bit-operations-class.xor*
definition *xor-int* :: *int* ⇒ *int* ⇒ *int* **where** *xor-int* ≡ *semiring-bit-operations-class.xor*
instance by *standard* (*simp-all add*: *xor-nat-def xor-int-def ac-simps semiring-bit-operations-class.eq-iff*)

end

1.1.3 Syntactic sugar

open-bundle *xor-syntax* **begin**

notation *xor* (**infixr** ‹$\oplus$› *59*)
notation *Xor-Sum* (‹$\bigoplus$›)

Now: lots of fancy syntax. First, *xor-sum* $(\lambda x.\ e)\ A$ is written $\bigoplus x{\in}A.\ e$.

syntax (*ASCII*)
-xor-sum :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b*::*abel-group-xor* (‹(‹*indent=3 notation=*‹*binder XORSUM*››*XORSUM* (-/:-)./ -)› [*0*, *51*, *10*] *10*)
syntax
-xor-sum :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b*::*abel-group-xor* (‹(‹*indent=2 notation=*‹*binder* $\bigoplus$››$\bigoplus$ (-/∈-)./ -)› [*0*, *51*, *10*] *10*)

syntax-consts
-xor-sum ⇌ *xor-sum*

translations — Beware of argument permutation!
$\bigoplus i{\in}A.\ b$ ⇌ *CONST xor-sum* $(\lambda i.\ b)\ A$
end

unbundle *no xor-syntax*
end
theory *Dependent-Product*
imports *HOL−Probability.Probability*
begin

The following lemma was reproduced from [3, Lemma *measure_pmf_prob_dependent_product_bound_eq*] The AFP imports `Monad_Normalisation.Monad_Normalisation` which alters some Isabelle syntax, which we want to avoid by copy pasting the lemmas here

lemma *measure-pmf-prob-dependent-product-bound-eq*:
assumes *countable A* $\bigwedge i.$ *countable* $(B\ i)$
assumes $\bigwedge a.\ a \in A \Longrightarrow$ *measure-pmf.prob* $N\ (B\ a) = r$
shows *measure-pmf.prob* (*pair-pmf M N*) (*Sigma A B*) = *measure-pmf.prob M A* $*$ *r*
(**is** $?L = ?R$)
proof −
have $?L =$
$(\sum_a (a,\ b) \in$ *Sigma A B. pmf M a* $*$ *pmf N b*)
by (*auto intro*!: *infsetsum-cong simp add*: *measure-pmf-conv-infsetsum pmf-pair*)

also have … = $(\sum_a a{\in}A.\ \sum_a b{\in}B\ a.$ *pmf M a* $*$ *pmf N b*)
apply (*subst infsetsum-Sigma*)
using *assms abs-summable-on-cong*[*of - pmf* (*pair-pmf M N*) λ*uub. pmf M* (*fst uub*) $*$ *pmf N* (*snd uub*)] *pmf-pair*[*of M N*]

by (*fastforce simp*: *case-prod-beta*)+

also have ... = ($\sum_a a \in A.$ *pmf M a* ∗ (*measure-pmf.prob N* (*B a*)))
by (*simp add*: *infsetsum-cmult-right measure-pmf-conv-infsetsum pmf-abs-summable*)

also have ... = ($\sum_a a \in A.$ *pmf M a* ∗ *r*)
using *assms*(*3*) **by** *fastforce*

also have ... = *?R*
by (*simp add*: *infsetsum-cmult-left pmf-abs-summable measure-pmf-conv-infsetsum*)

finally show *?thesis* **.**
qed

The following lemma was reproduced from [3, Lemma *measure_pmf_prob_dependent_product_bound_eq'*] The AFP imports `Monad_Normalisation.Monad_Normalisation` which alters some Isabelle syntax, which we want to avoid by copy pasting the lemmas here

lemma *measure-pmf-prob-dependent-product-bound-eq'*:
— N is the pmf we want to fix depending on values from M
assumes $\bigwedge a.\ a \in A \cap$ *set-pmf M* $\Longrightarrow$ *measure-pmf.prob N* (*B a*) = *r*
shows *measure-pmf.prob* (*pair-pmf M N*) (*Sigma A B*) = *measure-pmf.prob M A* ∗ *r*
(**is** *?L* = *?R*)
proof −
have *Sigma A B* ∩ (*set-pmf M* × *set-pmf N*) =
Sigma (*A* ∩ *set-pmf M*) (λ*i*. *B i* ∩ *set-pmf N*)
by *auto*

then have *?L* =
measure-pmf.prob (*pair-pmf M N*) (*Sigma* (*A* ∩ *set-pmf M*) (λ*i*. *B i* ∩ *set-pmf N*))
by (*metis measure-Int-set-pmf set-pair-pmf*)

also have ... = *measure-pmf.prob M* (*A* ∩ *set-pmf M*) ∗ *r*
by (*fastforce intro*: *measure-pmf-prob-dependent-product-bound-eq simp*: *assms measure-Int-set-pmf*)

also have ... = *?R* **by** (*simp add*: *measure-Int-set-pmf*)

finally show *?thesis* **.**
qed

end
theory *Xor-Inst*
imports
Xor
Fixed-Length-Vector.Fixed-Length-Vector
begin

More instances for *abel-group-xor*

unbundle *Fixed-Length-Vector.vec-syntax*

Sequences containing a base type that can be xor'd can itself also be xor'd, via element-wise xor

instantiation *vec* :: (*abel-group-xor*,*index1*) *abel-group-xor* **begin**

XORing two vectors is defined as element-wise XOR

definition *xor-vec* :: ('*a*, '*b*) *vec* ⇒
('*a*, '*b*) *vec* ⇒
('*a*, '*b*) *vec* **where**
xor-vec a b ≡ *map-vec* (λ(*a*',*b*'). *a*' *XOR b*') (*zip-vec a b*)

The identity vector is a vector where every element is zero

definition *zero-vec* ≡ *replicate-vec 0* :: ('*a*,'*b*) *vec*

instance proof
fix *a b c* :: ('*a*, '*b*) *vec*
show *eq-iff*: *a XOR b* = *0* ⟷ *a* = *b* **unfolding** *xor-vec-def zero-vec-def*
by (*metis* (*no-types*, *opaque-lifting*) *map-nth-vec replicate-nth-vec split-beta vec-cong*
abel-group-xor-class.eq-iff zip-vec-fst zip-vec-snd)
qed (*simp-all add*: *xor-vec-def zero-vec-def ac-simps vec-cong*)
end

lemma *nth-conv* [*simp*]:
shows (*a XOR b*) \$ *i* = *a* \$ *i XOR b* \$ *i*
unfolding *xor-vec-def* **by** *simp*

unbundle *no Fixed-Length-Vector.vec-syntax*
end
theory *Vec-Extras*
imports
HOL−Analysis.Finite-Cartesian-Product
Fixed-Length-Vector.Fixed-Length-Vector
begin

unbundle *no Finite-Cartesian-Product.vec-syntax*
unbundle *Fixed-Length-Vector.vec-syntax*

setup-lifting *type-definition-vec*

The most important lemmas to know when working with *Fixed-Length-Vector.vec* are

- *vec-of-list-inject*: ⟦*x* ∈ {*xs. length xs* = *CARD*('*b*)}; *y* ∈ {*xs. length xs* = *CARD*('*b*)}⟧ ⟹ (*vec-of-list x* = *vec-of-list y*) = (*x* = *y*)

- *vec-of-list-inverse*: $y \in \{xs.\ length\ xs = CARD('b)\} \Longrightarrow list\text{-}of\text{-}vec\ (vec\text{-}of\text{-}list\ y) = y$

- *nth-vec.rep-eq*: $(\$)\ x = (!)\ (list\text{-}of\text{-}vec\ x) \circ from\text{-}index$

Tip: When using *vec-of-list* to create a vector v from xs, it is essential to separately prove that $xs \in \{xs.\ length\ xs = CARD('b)\}$ for above lemmas to apply

lift-definition *vec-of-fcp* :: ('*a*, '*b*::{*finite*, *index1*}) *Finite-Cartesian-Product.vec* ⇒
('*a*, '*b*) *Fixed-Length-Vector.vec* **is**
‹λ*x*. *map* (λ*i*. *Finite-Cartesian-Product.vec-nth x i* :: '*a*) (*indexes* :: '*b list*)› **by** *simp*

definition *fcp-of-vec* :: ('*a*, '*b*::{*finite*, *index1*}) *Fixed-Length-Vector.vec* ⇒
('*a*, '*b*) *Finite-Cartesian-Product.vec* **where**
‹*fcp-of-vec* ≡ λ*x*. *Finite-Cartesian-Product.vec-lambda* (λ*i*. *x* $ *i*)›
lift-definition *enumerate-vec* :: *nat* ⇒ ('*a*, '*b*) *Fixed-Length-Vector.vec* ⇒
(*nat* × '*a*, '*b*) *Fixed-Length-Vector.vec* **is**
‹*List.indexed-from*› **by** *simp*

lemma *infinite-UNIV-vec*:
assumes *infinite* (*UNIV* :: '*a set*) *finite* (*UNIV* :: '*b set*)
shows *infinite* (*UNIV* :: ('*a*,'*b*) *Fixed-Length-Vector.vec set*)
proof −
let *?UNIVA* = *UNIV* :: '*a set*
let *?UNIVB* = *UNIV* :: '*b set*
let *?UNIVV* = *UNIV* :: ('*a*,'*b*) *Fixed-Length-Vector.vec set*

have *CARD*('*b*) > *0* **using** *assms*(*2*) *finite-UNIV-card-ge-0* **by** *blast*
then have (λ*x*. *nth-vec*' *x 0*) ‘ *?UNIVV* = *?UNIVA*
apply (*intro surjI*[**where** *?f* = λ*x*. *replicate-vec x*])
by (*simp add*: *nth-vec*'.*rep-eq*)
moreover have *card* ((λ*x*. *nth-vec*' *x 0*) ‘ *?UNIVV*) ≤ *card ?UNIVV*
by (*simp add*: *calculation assms*)
ultimately show *?thesis* **by** (*metis assms finite-imageI*)
qed

lemma *infinite-index-UNIV-vec*:
assumes *infinite* (*UNIV* :: '*b set*)
shows (*UNIV* :: ('*a*,'*b*) *Fixed-Length-Vector.vec set*) = {*vec-of-list* []}
by (*metis* (*mono-tags*, *lifting*) *UNIV-eq-I assms card.infinite insertI1 length-0-conv list-of-vec-inverse list-of-vec-length*)

lemma *card-UNIV-vec*:
shows *CARD*(('*a*, '*b*) *Fixed-Length-Vector.vec*) = *CARD*('*a*) ^ *CARD*('*b*) (**is** *?lhs* = *?rhs*)
proof −

let *?UNIVA* = *UNIV* :: *'a set*
let *?UNIVB* = *UNIV* :: *'b set*
let *?UNIVV* = *UNIV* :: (*'a*,*'b*) *Fixed-Length-Vector.vec set*

consider *finite ?UNIVA* | *infinite ?UNIVA finite ?UNIVB* | *infinite ?UNIVA infinite ?UNIVB*
by *fast*
then show *?thesis*
proof *cases*
case *1*
have *?lhs* = *card* ({*xs*. *length xs* = *CARD*(*'b*)} :: *'a list set*)
using *Fixed-Length-Vector.vec.type-definition-vec type-definition.card* **by** *blast*
also have ... = *?rhs* **using** *card-lists-length-eq*[*of ?UNIVA*, *OF 1*] **by** *simp*
finally show *?thesis* **.**
next
case *2*
have *infinite ?UNIVV* **using** *infinite-UNIV-vec*[*OF 2*] **.**
then show *?thesis* **using** *2 finite-UNIV-card-ge-0* **by** (*simp add*: *card-eq-0-iff*)
next
case *3*
have *card ?UNIVV* = *Suc 0* **unfolding** *infinite-index-UNIV-vec*[*OF 3*(*2*)] **by** *simp*
then show *?thesis* **using** *3* **by** (*simp add*: *card-eq-0-iff*)
qed
qed

lemma (**in** *index1*) *from-index-neq* [*simp*]:
assumes $a \neq b$
shows *from-index a* ≠ *from-index b*
by (*metis assms local.from-to-index*)

lemma *list-of-vec-not-empty* [*simp*]:
fixes *xs* :: (*'a*, *'b* :: *index1*) *Fixed-Length-Vector.vec*
shows *list-of-vec xs* ≠ []
by (*metis from-index-bound gr-implies-not0 list.size*(*3*) *list-of-vec-length*)

lemma *id-take-nth-drop-vec*:
fixes *xs* :: (*'a*, *'b* :: *index1*) *Fixed-Length-Vector.vec*
assumes *i* < *CARD*(*'b*)
shows
xs = *vec-of-list* (
take i (*list-of-vec xs*) @
list-of-vec xs ! *i* #
drop (*Suc i*) (*list-of-vec xs*))
by (*simp add*: *Cons-nth-drop-Suc assms*)

lemma *id-take-nth-drop-vec'*:

fixes *xs* :: ('*a*, '*b* :: *index1*) *Fixed-Length-Vector.vec*
assumes $i < CARD('b)$
shows
xs = *vec-of-list* (
take i (*list-of-vec xs*) @
xs $ *to-index i* #
drop (*Suc i*) (*list-of-vec xs*))
by (*simp add*: *Cons-nth-drop-Suc assms nth-vec.rep-eq to-from-index*)

lemma *nth-not-mem-enumerate*:
assumes *y* ! *i* ≠ *x* ! *i*
shows (*i*, *y* ! *i*) ∉ *set* (*List.indexed-from 0 x*)
apply (*subst in-set-conv-nth*, *clarsimp*)
using *assms nth-indexed-from-eq* **by** *fastforce*

lemma *nth-not-mem-enumerate-take*:
assumes *y* ! *j* ≠ *x* ! *j*
shows (*j*, *y* ! *j*) ∉ *set* (*List.indexed-from 0* (*take i x*))
apply (*subst in-set-conv-nth*, *clarsimp*)
by (*metis arith-simps*(*49*) *assms length-take min-less-iff-conj nth-indexed-from-eq*
nth-take
prod.sel(*1*,*2*))

lemma *nth-not-mem-enumerate-drop*:
assumes *y* ! *j* ≠ *x* ! *j*
shows (*j*, *y* ! *j*) ∉ *set* (*List.indexed-from* (*Suc i*) (*drop* (*Suc i*) *x*))
apply (*subst in-set-conv-nth*, *clarsimp*)
by (*metis assms fst-eqD length-drop linorder-not-le nat-diff-split not-less-zero*
nth-drop
nth-indexed-from-eq snd-eqD)

lemma *nth-enumerate-eq-vec*:
fixes *xs* :: ('*a*, '*b* :: *index1*) *Fixed-Length-Vector.vec*
assumes $m < CARD('b)$
shows *enumerate-vec n xs* $ *to-index m* = (*n* + *m*, *xs* $ *to-index m*)
by (*simp add*: *nth-vec.rep-eq index.to-from-index index-axioms assms enumer-ate-vec.rep-eq*
nth-indexed-from-eq)

lemma *in-set-enumerate-eq-vec*:
fixes *xs* :: ('*a*, '*b* :: *index1*) *Fixed-Length-Vector.vec*
shows *p* ∈ *set-vec* (*enumerate-vec n xs*) ⟷
n ≤ *fst p* ∧ *fst p* < *CARD*('*b*) + *n* ∧ *xs* $ *to-index* (*fst p* − *n*) = *snd p*
apply (*simp add*:
in-set-indexed-from-eq nth-vec.rep-eq set-vec.rep-eq index.to-from-index
index-axioms enumerate-vec.rep-eq nth-indexed-from-eq)
by (*metis less-diff-conv2 to-from-index*)

```
lemma inj-vec-of-fcp:
  shows inj vec-of-fcp
  apply (clarsimp
    intro!: injI
    simp: vec-of-fcp-def vec-of-list-inject indexes-def vec-eq-iff Ball-def)
  by (metis from-to-index from-index-bound)

lemma inj-fcp-of-vec:
  shows inj fcp-of-vec
  by (auto intro!: injI vec-cong simp: fcp-of-vec-def)

lemma fcp-of-vec-inverse:
  shows vec-of-fcp (fcp-of-vec x) = x
  apply (simp add: vec-of-fcp-def fcp-of-vec-def)
  by (metis UNIV-I vec-explode vec-lambda.abs-eq vec-lambda-inverse)

lemma vec-of-fcp-inverse:
  shows fcp-of-vec (vec-of-fcp x) = x
  apply (simp add: vec-of-fcp-def fcp-of-vec-def)
  by (metis vec-lambda.abs-eq vec-lambda-nth vec-lambda-unique)

lemma surj-vec-of-fcp:
  shows surj vec-of-fcp
  using fcp-of-vec-inverse by (intro surjI)

lemma surj-fcp-of-vec:
  shows surj fcp-of-vec
  using vec-of-fcp-inverse by (intro surjI)

lemma bij-vec-of-fcp:
  shows bij vec-of-fcp
  by (intro bijI inj-vec-of-fcp surj-vec-of-fcp)

lemma bij-fcp-of-vec:
  shows bij fcp-of-vec
  by (intro bijI inj-fcp-of-vec surj-fcp-of-vec)

unbundle no Fixed-Length-Vector.vec-syntax
end
theory Simple-Tabulation-Hashing
imports
  Universal-Hash-Families.Universal-Hash-Families-More-Product-PMF
  Dependent-Product
  Xor-Inst
  Vec-Extras
begin

unbundle Xor.xor-syntax
unbundle Fixed-Length-Vector.vec-syntax
```

unbundle *no Finite-Cartesian-Product.vec-syntax*
hide-type *Finite-Cartesian-Product.vec*

1.2 Intro rules, contrapositives and bijections

lemma (**in** *prob-space*) *k-wise-indep-varsI*:
assumes $\bigwedge a\ J.\ [\![\ J \subseteq I;\ card\ J \leq k;\ finite\ J\ ]\!] \Longrightarrow$
$prob\ \{\omega.\ \forall i{\in}J.\ X\ i\ \omega = a\ i\} = (\prod i{\in}J.\ prob\ \{\omega.\ X\ i\ \omega = a\ i\})$
$M = measure\text{-}pmf\ p$
shows *k-wise-indep-vars k* (λ-. *count-space UNIV*) *X I*
apply (*simp only*: *k-wise-indep-vars-def prob-space-axioms*)
apply (*clarsimp*, *rule indep-vars-pmf*, *rule assms*(*2*))
by (*metis* (*no-types*, *lifting*) *assms*(*1*) *card-mono order-trans-rules*(*23*))

lemma (**in** *prob-space*) *indep-vars-pmf-contrapos*:
assumes $prob\ \{\omega.\ \forall x{\in}J.\ P\ x\ (X'\ x\ \omega)\} \neq (\prod x{\in}J.\ prob\ \{\omega.\ P\ x\ (X'\ x\ \omega)\})$
finite J M = *measure-pmf p*
shows ¬ *indep-vars* (λ-. *count-space UNIV*) *X′ J*
apply (*rule contrapos-nn*[*OF assms*(*1*)])
using *assms* **by** (*intro split-indep-events*; *force*)

locale *bij-betw-funcsetE* **begin**

These lemmas prove bijections between sets of 2-arity functions and their decompositions

For example, a function $f : [0, 100] \rightharpoonup \{A, B, C\} \rightharpoonup \{True, False\}$ can be decomposed into 3 parts:

$$\begin{aligned} & [0, 100] \rightharpoonup \{A, B, C\} \rightharpoonup \{True, False\} \\ \rightleftharpoons\ & [1, 100] \rightharpoonup \{A, B, C\} \rightharpoonup \{True, False\} \times \\ & \quad \{A, C\} \rightharpoonup \{True, False\} \times \\ & \quad\quad \{True, False\} \end{aligned}$$

- base: $[1, 100] \rightharpoonup \{A, B, C\} \rightharpoonup \{True, False\}$ (split the 0-index, leaving 1-100 intact)
- z_0: $\{A, C\} \rightharpoonup \{True, False\}$ (represents the 0-index, but further split the B-index)
- z_b: $B \in \{True, False\}$ (decides the value for $f\ 0\ B$)

f is equivalent to its reconstruction, $f = base(0 := z_0(B := z_b))$

context
fixes α **and** *A B* :: - *set*
assumes *α-in-A*: α ∈ *A*
begin

lemma *bij-PiE-remove-point*:
  bij-betw (λ (f, v). $f(\alpha := v)$) (($A - \{\alpha\} \rightarrow_E B) \times B$) ($A \rightarrow_E B$)
proof (*intro bij-betw-imageI inj-onI*, *goal-cases inj surj*)
  case (*inj - -*)
  then show *?case*
    apply (*simp add*: *case-prod-beta′ image-iff prod-eq-iff set-eq-iff mem-Times-iff fun-eq-iff*)
    by (*metis Diff-iff PiE-arb insertCI*)
next
  case *surj*
  then show *?case*
  proof (*intro equalityI subsetI*)
    fix h **assume** $h \in A \rightarrow_E B$
    with *α-in-A* **have** $h\ \alpha \in B$ $h(\alpha := \mathit{undefined}) \in A - \{\alpha\} \rightarrow_E B$
      by (*auto simp*: *PiE-def extensional-def*)
    then show $h \in (\lambda\ (f,\ v).\ f(\alpha := v))$ ‘ (($A - \{\alpha\} \rightarrow_E B) \times B$)
      apply (*simp add*: *case-prod-beta′ image-iff*)
      by (*metis fun-upd-upd fun-upd-triv*)
  next
    fix h **assume** $h \in (\lambda(f,\ v).\ f(\alpha := v))$ ‘ (($A - \{\alpha\} \rightarrow_E B) \times B$)
    with *α-in-A* **show** $h \in A \rightarrow_E B$
      apply (*simp add*: *case-prod-beta′ image-iff*)
      by (*metis insert-Diff PiE-fun-upd*)
  qed
qed

Helper: "Absorb" a point into a function space bijection. This allows chaining: if we have a bijection F for a smaller domain $S \rightarrow (A - \{x\} \rightarrow B)$, we can automatically get a bijection for $S\ x\ B \rightarrow (A \rightarrow B)$.

lemma *bij-absorb-point*:
  assumes *bij-betw* F S ($A - \{\alpha\} \rightarrow_E B$)
  shows *bij-betw* (λ (s, v). (F s)($\alpha := v$)) ($S \times B$) ($A \rightarrow_E B$)
proof –
  — 1. Combine F with Identity on B
  from *assms* **have** *bij-betw* (*map-prod F id*) ($S \times B$) (($A - \{\alpha\} \rightarrow_E B) \times B$)
    by (*blast intro*: *bij-betw-map-prod*)
  — 2. Apply the base removal lemma
  with *bij-PiE-remove-point* **show** *?thesis*
    by (*auto*
      elim: *bij-betw-trans*[*simplified comp-def*, *elim-format*]
      simp: *map-prod-def case-prod-beta′*)
qed

Helper 2: "Lift Value" Attaches a computed value V to the function at point x. Used for the outer function (e.g., $a(\alpha := V)$).

lemma *bij-lift-value*:
  assumes *bij-betw* V S B
  shows *bij-betw* ($\lambda(f,\ s).\ f(\alpha := V\ s)$) (($A - \{\alpha\} \rightarrow_E B) \times S$) ($A \rightarrow_E B$)
proof –

```
  have bij-betw (map-prod id V) ((A − {α} →E B) × S) ((A − {α} →E B) × B)
    using assms by (simp add: bij-betw-map-prod)
  with bij-PiE-remove-point show ?thesis
    by (auto
      elim: bij-betw-trans[simplified comp-def, elim-format]
      simp: map-prod-def case-prod-beta′)
qed

end
```

Tuple Association Helper: $((A, B), C) \longleftrightarrow (A, B, C)$

```
lemma bij-assoc-right-to-left:
  bij-betw (λ (a, b, c). ((a, b), c)) (A × B × C) ((A × B) × C)
  by (auto intro: bij-betwI[where g=λ ((a, b), c). (a, b, c)])

lemma bij-assoc-left-to-right:
  bij-betw (λ((a, b), c). (a, b, c)) ((A × B) × C) (A × B × C)
  by (auto intro: bij-betwI[where g=λ(a, b, c). ((a, b), c)])

lemma bij-betw-combine-right:
  assumes bij-betw f B C
  shows bij-betw (λ(a, b). (a, f b)) (A × B) (A × C)
 using assms by (simp flip: map-prod-def id-def add: case-prod-beta′ bij-betw-map-prod)
```

1.2.1 Application

```
lemma remove1-remove1:
  assumes α ∈ A β ∈ B finite A finite B finite C C ≠ {}
  shows bij-betw
      (λ(a,b,c). a(α := b(β := c)))
      ((A − {α} →E B →E C) × (B − {β} →E C) × C)
      (A →E B →E C)
proof −
  — 1. Establish bijection for the value: b(β := c)
  have bij-val: bij-betw (λ(b, c). b(β := c)) ((B − {β} →E C) × C) (B →E C)
    using bij-PiE-remove-point[OF ‹β ∈ B›] by auto

  — 2. Lift this value to A
  have bij-lift: bij-betw (λ(a, (b,c)). a(α := b(β := c)))
                  ((A − {α} →E B →E C) × ((B − {β} →E C) × C))
                  (A →E B →E C)
    using bij-lift-value[OF ‹α ∈ A› bij-val] by (auto simp: case-prod-beta′)

  — 3. Fix tuple associativity (a, b, c) → (a, (b, c))
  have bij-assoc: bij-betw (λ(a,b,c). (a,(b,c)))
                   ((A − {α} →E B →E C) × (B − {β} →E C) × C)
                   ((A − {α} →E B →E C) × ((B − {β} →E C) × C))
    by (simp add: bij-betwI[where g=λ(a, (b, c)). (a, b, c)])
```

show *?thesis*
  using *bij-betw-trans*[*OF bij-assoc bij-lift*] **by** (*simp add*: *case-prod-beta*)
qed

lemma *remove1-remove2*:
  assumes $\beta 1 \neq \beta 2$ $\alpha \in A$ $\beta 1 \in B$ $\beta 2 \in B$
    finite A *finite* B *finite* C $C \neq \{\}$
  shows *bij-betw*
    $(\lambda(a,b,c,d).\ a(\alpha := b(\beta 1 := c,\ \beta 2 := d)))$
    $((A - \{\alpha\} \rightarrow_E B \rightarrow_E C) \times (B - \{\beta 1,\beta 2\} \rightarrow_E C) \times C \times C)$
    $(A \rightarrow_E B \rightarrow_E C)$
proof −
  let $?S\text{-}B = (B - \{\beta 1,\beta 2\} \rightarrow_E C) \times C \times C$

  — 1. Build bijection for value: $b(\beta 1 := c,\ \beta 2 := d)$
  — We use *bij-absorb-point* recursively

  have $\beta 1 \in B - \{\beta 2\}$ **using** *assms* **by** *simp*
  — Start with remove_point for $\beta 1$
  have *bij-B-start*: *bij-betw* $(\lambda(b,\ c).\ b(\beta 1 := c))$
    $((B - \{\beta 1,\beta 2\} \rightarrow_E C) \times C)$ $(B - \{\beta 2\} \rightarrow_E C)$
  using *bij-PiE-remove-point*[*OF* ‹$\beta 1 \in B - \{\beta 2\}$›] **by** (*metis Diff-insert*)

  — Absorb $\beta 2$
  have *bij-B-next*: *bij-betw* $(\lambda((b,\ c),\ d).\ b(\beta 1 := c,\ \beta 2 := d))$
    $(((B - \{\beta 1,\beta 2\} \rightarrow_E C) \times C) \times C)$ $(B \rightarrow_E C)$
  using *bij-absorb-point*[*OF* ‹$\beta 2 \in B$› *bij-B-start*] **by** (*simp add*: *case-prod-beta'*)

  — Flatten tuple for B
  have *bij-B-val*: *bij-betw* $(\lambda(b,c,d).\ b(\beta 1 := c,\ \beta 2 := d))$ $?S\text{-}B$ $(B \rightarrow_E C)$
    using *bij-betw-trans*[*OF bij-assoc-right-to-left bij-B-next*]
    by (*simp add*: *case-prod-beta' comp-def*)

  — 2. Lift to A
  have *bij-lift*: *bij-betw* $(\lambda(a,\ (b,c,d)).\ a(\alpha := b(\beta 1 := c,\ \beta 2 := d)))$
    $((A - \{\alpha\} \rightarrow_E B \rightarrow_E C) \times ?S\text{-}B)$ $(A \rightarrow_E B \rightarrow_E C)$
  using *bij-lift-value*[*OF* ‹$\alpha \in A$› *bij-B-val*] **by** (*simp add*: *case-prod-beta' comp-def*)

  — 3. Fix tuple associativity
  have *bij-assoc*: *bij-betw* $(\lambda(a,b,c,d).\ (a,(b,c,d)))$
    $((A - \{\alpha\} \rightarrow_E B \rightarrow_E C) \times ?S\text{-}B)$
    $((A - \{\alpha\} \rightarrow_E B \rightarrow_E C) \times ?S\text{-}B)$
  by (*simp add*: *bij-betwI* [**where** $g{=}\lambda(a,\ (b,c,d)).\ (a,b,c,d)$])

  show *?thesis*
    using *bij-betw-trans*[*OF bij-assoc bij-lift*] **by** (*simp add*: *case-prod-beta*)
qed

lemma *remove1-remove3*:

assumes $\beta 1 \neq \beta 2$ $\beta 2 \neq \beta 3$ $\beta 3 \neq \beta 1$ $\alpha \in A$ $\beta 1 \in B$ $\beta 2 \in B$ $\beta 3 \in B$
finite A finite B finite C $C \neq \{\}$
shows *bij-betw*
$(\lambda(a,b,c,d,e).\ a(\alpha := b(\beta 1 := c,\ \beta 2 := d,\ \beta 3 := e)))$
$((A - \{\alpha\} \rightarrow_E B \rightarrow_E C) \times (B - \{\beta 1,\beta 2,\beta 3\} \rightarrow_E C) \times C \times C \times C)$
$(A \rightarrow_E B \rightarrow_E C)$
proof −
let $?DomB3 = B - \{\beta 1,\beta 2,\beta 3\} \rightarrow_E C$
let $?S\text{-}B = ?DomB3 \times C \times C \times C$

— 1. Build bijection for value B recursively

— Step 1: Base $\beta 1$
from *assms bij-PiE-remove-point*[*of* $\beta 1$ $B - \{\beta 3,\ \beta 2\}$]
have *bij-betw* $(\lambda(b,c).\ b(\beta 1{:=}c))$ $(?DomB3 \times C)$ $(B - \{\beta 3,\ \beta 2\} \rightarrow_E C)$
by (*auto simp flip*: *Diff-insert simp*: *insert-commute*)

— Step 2: Absorb $\beta 2$
with *assms bij-absorb-point*[*of* $\beta 2$ $B - \{\beta 3\}$]
have *bij-betw* $(\lambda((b,c),d).\ b(\beta 1{:=}c,\ \beta 2{:=}d))$ $((?DomB3 \times C) \times C)$ $(B - \{\beta 3\}$
$\rightarrow_E C)$
by (*auto simp flip*: *Diff-insert simp*: *case-prod-beta′ insert-commute*)

— Step 3: Absorb $\beta 3$
with *assms bij-absorb-point*[*of* $\beta 3$ B]
have *bij-3*: *bij-betw* $(\lambda(((b,c),d),e).\ b(\beta 1{:=}c,\ \beta 2{:=}d,\ \beta 3{:=}e))$ $(((?DomB3 \times C)$
$\times C) \times C)$ $(B \rightarrow_E C)$
by (*auto simp*: *case-prod-beta′ insert-commute*)

moreover have *bij-betw* $(\lambda(b,c,d,e).\ (((b,c),d),e))$ $?S\text{-}B$ $(((?DomB3 \times C) \times C)$
$\times C)$
by (*auto intro*: *bij-betwI*[**where** $g{=}\lambda(((b,c),d),e).\ (b,c,d,e)$])

— Step 4: Flatten tuple for B
ultimately have *bij-betw* $(\lambda(b,c,d,e).\ b(\beta 1{:=}c,\ \beta 2{:=}d,\ \beta 3{:=}e))$ $?S\text{-}B$ $(B \rightarrow_E$
$C)$
by (*auto*
elim: *bij-betw-trans*[*simplified comp-def*, *elim-format*]
simp: *case-prod-beta′*)

— 2. Lift to A
with *bij-lift-value assms*
have *bij-lift*: *bij-betw* $(\lambda(a,\ (b,c,d,e)).\ a(\alpha := b(\beta 1{:=}c,\ \beta 2{:=}d,\ \beta 3{:=}e)))$
$((A - \{\alpha\} \rightarrow_E B \rightarrow_E C) \times ?S\text{-}B)$ $(A \rightarrow_E B \rightarrow_E C)$
by (*simp add*: *case-prod-beta′*)

— 3. Fix tuple associativity
have *bij-assoc*: *bij-betw* $(\lambda(a,b,c,d,e).\ (a,(b,c,d,e)))$
$((A - \{\alpha\} \rightarrow_E B \rightarrow_E C) \times ?S\text{-}B)$

$((A - \{\alpha\} \rightarrow_E B \rightarrow_E C) \times$ *?S-B*)
by (*auto intro*: *bij-betwI* [**where** $g{=}\lambda(a,\ (b,c,d,e)).\ (a,b,c,d,e)$])

with *bij-betw-trans*[*OF bij-assoc bij-lift*] **show** *?thesis*
by (*simp add*: *case-prod-beta*)
qed

lemma *remove2-remove2′remove1*:
assumes $\alpha 1 \neq \alpha 2$ $\beta 1 \neq \beta 2$ $\alpha 1 \in A$ $\alpha 2 \in A$ $\beta 1 \in B$ $\beta 2 \in B$ $\beta 3 \in B$
finite A finite B finite C $C \neq \{\}$
shows *bij-betw*
$(\lambda(a,b,c,d,e,f).\ a(\alpha 1 := b(\beta 1 := d,\ \beta 2 := e),\ \alpha 2 := c(\beta 3 := f)))$
$((A - \{\alpha 1,\ \alpha 2\} \rightarrow_E B \rightarrow_E C) \times (B - \{\beta 1,\ \beta 2\} \rightarrow_E C) \times (B - \{\beta 3\} \rightarrow_E C) \times C \times C \times C)$
$(A \rightarrow_E B \rightarrow_E C)$
proof −
let *?DomA* $= A - \{\alpha 1,\ \alpha 2\} \rightarrow_E B \rightarrow_E C$
let *?DomB1* $= B - \{\beta 1,\ \beta 2\} \rightarrow_E C$
let *?DomB2* $= B - \{\beta 3\} \rightarrow_E C$

have *bij-betw* $(\lambda(b,d,e).\ ((b,d),e))$ (*?DomB1* $\times C \times C$) ((*?DomB1* $\times C) \times C$)
by (*auto intro*: *bij-betwI*[**where** $g{=}\lambda((b,d),e).\ (b,d,e)$])

moreover from *assms bij-absorb-point*[*of* $\beta 2$ *B*] *bij-PiE-remove-point*[*of* $\beta 1$ *B* $- \{\beta 2\}$]
have *bij-betw* $(\lambda((b,d),e).\ b(\beta 1{:=}d,\ \beta 2{:=}e))$ ((*?DomB1* $\times C) \times C$) $(B \rightarrow_E C)$
by (*auto simp flip*: *Diff-insert simp*: *case-prod-beta′*)

— 1. Val 1: $b(\beta 1 := d,\ \beta 2 := e)$
ultimately have *bij-betw* $(\lambda(b,\ d,\ e).\ b(\beta 1 := d,\ \beta 2 := e))$ (*?DomB1* $\times C \times C$) $(B \rightarrow_E C)$
by (*auto elim*: *bij-betw-trans*[*simplified comp-def*, *elim-format*] *simp*: *case-prod-beta′*)

— 2. Val 2: $c(\beta 3 := f)$
moreover from *bij-PiE-remove-point*[*of* $\beta 3$ *B*] *assms*
have *bij-betw* $(\lambda(c,\ f).\ c(\beta 3 := f))$ (*?DomB2* $\times C$) $(B \rightarrow_E C)$ **by** *auto*

— 3. Combine values (Product Step)
— We explicitly form the tuple structure $((B1, C, C), (B2, C))$
ultimately have *bij-vals*: *bij-betw*
$(\lambda((b,d,e),\ (c,f)).\ (b(\beta 1{:=}d,\ \beta 2{:=}e),\ c(\beta 3{:=}f)))$
((*?DomB1* $\times C \times C) \times$ (*?DomB2* $\times C$))
$((B \rightarrow_E C) \times (B \rightarrow_E C))$
by (*fastforce dest*: *bij-betw-map-prod simp add*: *map-prod-def id-def case-prod-beta′*)

— 4. Lift to Parallel
from *bij-betw-map-prod*[*OF bij-betw-id bij-vals*] **have** *bij-parallel*:
bij-betw
$(\lambda(a,\ rest).\ (a,\ (\lambda((b,d,e),\ (c,f)).\ (b(\beta 1{:=}d,\ \beta 2{:=}e),\ c(\beta 3{:=}f)))\ rest))$

```
    (?DomA × ((?DomB1 × C × C) × (?DomB2 × C)))
    (?DomA × ((B →E C) × (B →E C)))
  by (auto simp: map-prod-def id-def case-prod-beta′)

have bij-A-step1: bij-betw (λ((a, v1), v2). (a(α1 := v1), v2))
               ((?DomA × (B →E C)) × (B →E C)) ((A − {α2} →E B →E C)
× (B →E C))
  using assms bij-lift-value[of α1 A − {α2}]
  by (force
    intro: bij-betw-map-prod simp flip: map-prod-def id-def Diff-insert
    simp: case-prod-beta′)

have bij-A-step2: bij-betw (λ(a, v2). a(α2 := v2))
               ((A − {α2} →E B →E C) × (B →E C)) (A →E B →E C)
  using assms bij-PiE-remove-point[of α2 A] by (auto simp: case-prod-beta′)

— 5. Explicit Shuffle
have bij-shuffle: bij-betw (λ(a, b, c, d, e, f). (a, (b, d, e), (c, f)))
    (?DomA × ?DomB1 × ?DomB2 × C × C × C)
    (?DomA × (?DomB1 × C × C) × (?DomB2 × C))
  by (auto intro: bij-betwI[where g=λ(a, (b, d, e), (c, f)). (a, b, c, d, e, f)])

— 6. Final Chain - broken into steps to avoid unification issues
have bij-betw (λ(a, (b,d,e), (c,f)). ((a, b(β1:=d, β2:=e)), c(β3:=f)))
      (?DomA × (?DomB1 × C × C) × (?DomB2 × C))
      ((?DomA × (B →E C)) × (B →E C))
proof −
  have bij-betw (λ(a, v1, v2). ((a, v1), v2))
        (?DomA × (B →E C) × (B →E C)) ((?DomA × (B →E C)) × (B →E
C))
    by (rule bij-betwI [where g=λ((a,v1),v2). (a,v1,v2)]) auto
  with bij-parallel show ?thesis
   by (auto elim: bij-betw-trans[simplified comp-def, elim-format] simp: case-prod-beta′)
qed

with bij-shuffle have
  bij-betw (λ(a, b, c, d, e, f). ((a, b(β1:=d, β2:=e)), c(β3:=f)))
    (?DomA × ?DomB1 × ?DomB2 × C × C × C)
    ((?DomA × (B →E C)) × (B →E C))
 by (auto elim: bij-betw-trans[simplified comp-def, elim-format] simp: case-prod-beta′)

moreover from bij-A-step1 bij-A-step2 have
  bij-betw (λ((a, v1), v2). a(α1 := v1, α2 := v2))
    ((?DomA × (B →E C)) × (B →E C))
    (A →E B →E C)
 by (auto elim: bij-betw-trans[simplified comp-def, elim-format] simp: case-prod-beta′)

ultimately show ?thesis
 by (auto elim: bij-betw-trans[simplified comp-def, elim-format] simp: case-prod-beta′)
```

qed

end

2 Definitions

locale *simple-tabulation-hashing* =
  fixes $n :: 'n :: \{index1, zero\}$ *itself* — key length, i.e. 4 for 4 'fragments
  fixes $r :: 'result :: \{abel\text{-}group\text{-}xor, finite\}$ *itself* — bool or any type that supports xor
  fixes $q :: 'q ::$ *index1 itself* — digest length, i.e. 3
  fixes $d :: 'fragment ::$ *finite itself* — domain of input key fragments
  assumes n: $n = TYPE('n)$
begin

In tabulation hashing, keys are split into n fragments, each piece hashed individually, before being combined into a final digest with the XOR operator. i.e. $x \equiv [x_0, x_1, x_2]$, $h(x) \equiv h_0 x_0 \oplus h_1 x_1 \oplus h_2 x_2$

Therefore, the type variable 'n, CARD('n) and n must reflect the number of parts in a key. In the example above, 'n = 3 (which is an alias for num1 bit1)

The final digest length is indicated by the type variable 'q, and 'result is any type that supports xor. For example, 'result = bool. When using 'result = bool and 'q = 3, it can be interpreted as using a 3-bit word as the output, which can encode $2^3 = 8$ possibilities, e.g. [True, False, True]

abbreviation *cardn* :: *'n itself* ⇒ *nat* **where** *cardn* - ≡ $CARD('n)$

abbreviation N :: *'n itself* ⇒ *nat set* **where** N *type* ≡ {..<*cardn type*}

abbreviation D :: *'fragment set* **where** $D \equiv UNIV$

abbreviation *Dn* :: (*'fragment*, *'n*) *vec set* (‹D^n›) **where**
  — domain of all possible input keys s.t. each key is composed of exactly *n* fragments
  $D^n \equiv UNIV$

abbreviation R :: (*'result*, *'q*) *vec set* **where**
  — range of all possible outputs, s.t. each output is composed of exactly *q* bits
  $R \equiv UNIV$

abbreviation H :: *'n itself* ⇒ (*nat* ⇒ *'fragment* ⇒ (*'result*, *'q*) *vec*) *set* **where**
  — set of all possible assignments and combinations for every column-item pair
  H *type* ≡ N *type* $\rightarrow_E D \rightarrow_E R$

definition *tb-S* :: *'n itself* ⇒ (*nat* ⇒ *'fragment* ⇒ (*'result*, *'q*) *vec*) *pmf* **where**
  — Generator of instances of H
  tb-S - ≡ *pmf-of-set* (H n)

definition *tb-H* :: ($'fragment$, $'n$) *vec* ⇒ (*nat* ⇒ $'fragment$ ⇒ ($'result$, $'q$) *vec*) ⇒ ($'result$, $'q$) *vec* **where**
  — Applies the tabulation hashing algorithm on *k* using a hash function from H
  tb-H k h ≡ *xor-fold* (*map* (λ*i*. *h i* (*k* $ *to-index i*)) ([*0*..<*cardn n*]))

lemma *tb-S-alt-def*:
  shows *tb-S n* = *prod-pmf* (*N n*) (λ-. *prod-pmf UNIV* (λ-. *pmf-of-set R*))
  unfolding *tb-S-def*
  by (*simp add*: *Pi-pmf-of-set PiE-defaut-undefined-eq*)

lemma *tb-H-absorb*:
  assumes *i* < *CARD*($'n$)
  shows *tb-H x h* ⊕ *c* = *tb-H x* (*h*(*i* := (*h i*)(*x* $ *to-index i* := *h i* (*x* $ *to-index i*) ⊕ *c*)))
  unfolding *tb-H-def*
  by (*auto intro*!: *arg-cong*[**where** *f* = *xor-fold*] *simp*: *fold-absorb*[*of i*] *assms*)

lemma *tb-H-absorb'*:
  shows *tb-H x h* ⊕ *c* =
    tb-H x (*h*(*from-index i* := (*h* (*from-index i*))(*x* $ *i* := *h* (*from-index i*) (*x* $ *i*) ⊕ *c*)))
  by (*metis from-index-bound from-to-index tb-H-absorb*)

lemma *tb-H-extract*:
  assumes *i* < *CARD*($'n$)
  shows *tb-H x* (*h*(*i* := *g*(*x* $ *to-index i* := α))) = *tb-H x* (*h*(*i* := *g*(*x* $ *to-index i* := *0*))) ⊕ α
  by (*subst tb-H-absorb*[*OF assms*]) *simp*

lemma *tb-H-extract'*:
  shows *tb-H x* (*h*(*from-index i* := *g*(*x* $ *i* := α))) = *tb-H x* (*h*(*from-index i* := *g*(*x* $ *i* := *0*))) ⊕ α
  by (*metis from-index-bound from-to-index tb-H-extract*)

lemma *tb-H-exch*:
  assumes *i* < *CARD*($'n$)
  shows *tb-H x* (*a*(*i* := *aa*(*x* $ *to-index i* := *b*))) = α ⟷
    b = *tb-H x* (*a*(*i* := *aa*(*x* $ *to-index i* := α)))
  apply (*subst* (*1 2*) *tb-H-extract*[*OF assms*])
  by (*metis* (*no-types*, *lifting*) *assoc abel-group-xor-class.eq-iff*)

lemma *tb-H-exch'*:
  shows *tb-H x* (*a*(*from-index i* := *aa*(*x* $ *i* := *b*))) = α ⟷
    b = *tb-H x* (*a*(*from-index i* := *aa*(*x* $ *i* := α)))
  by (*metis from-index-bound from-to-index tb-H-exch*)

lemma *tb-H-discard*:
  assumes *x* $ *i* ≠ *y* $ *i*

shows *tb-H x* (*a*(*from-index i* := *b*(*y* $ *i* := *c*))) = *tb-H x* (*a*(*from-index i* := *b*))
unfolding *tb-H-def*
apply (*intro arg-cong*[**where** *?f* = *xor-fold*])
by (*simp-all add*: *assms*)

3 Proofs

3.1 Proof of 1-independence and uniformity

lemma *prob-tb-H-bin*:
shows *measure-pmf.prob* (*tb-S n*) {*h*. *tb-H x h* = α} = *1* / *real* (*card R*) (**is** *?lhs* = *?rhs*)
proof −
let *?N* = *N n*
let *?H* = *H n*
let *?tb-S* = *tb-S n*
let *?x0* = *x* $ *to-index 0*

note *finite-PiE*[*simp*] *PiE-eq-empty-iff*[*simp*] *measure-pmf-single*[*simp*]

have *bij*:
bij-betw (λ(*a*, *b*, *c*). *a*(*0* := *b*(*?x0* := *c*)))
((*?N* − {*0*} $\rightarrow_E$ *D* $\rightarrow_E$ *R*) × (*D* − {*?x0*} $\rightarrow_E$ *R*) × *R*)
?H
by (*fastforce intro*!: *bij-betw-funcsetE.remove1-remove1*)

let *?destructure* =
pair-pmf (*pmf-of-set* (*?N* − {*0*} $\rightarrow_E$ *D* $\rightarrow_E$ *R*))
(*pair-pmf* (*pmf-of-set* (*D* − {*?x0*} $\rightarrow_E$ *R*))
(*pmf-of-set R*))

have *asmap*: *?tb-S* =
map-pmf (λ(*a*, *b*, *c*). *a*(*0* := *b*(*?x0* := *c*))) *?destructure*
unfolding *tb-S-def*
by (*simp add*: *bij flip*: *pmf-of-set-prod-eq map-pmf-of-set-bij-betw del*: *PiE-UNIV*)

have *?lhs* =
measure-pmf.prob ?destructure
(*Sigma UNIV* (λ*h*.
Sigma UNIV (λ*Z*.
{*tb-H x* (*h*(*0* := *Z*(*?x0* := α)))})))
unfolding *asmap measure-map-pmf* **by** (*clarsimp intro*!: *measure-pmf-cong tb-H-exch*)

also have ... = *?rhs* **by** (*simp add*: *measure-pmf-prob-dependent-product-bound-eq*′)

finally show *?thesis* **.**
qed

theorem *uniform*:
  shows *prob-space.uniform-on* (*tb-S n*) (*tb-H key*) *R*
  using *prob-tb-H-bin* **by** (*auto intro*!: *measure-pmf.uniform-onI*)

3.2 Proof of two-independence

lemma *prob-tb-H-bin1-bin2*:
  assumes $x \neq y$
  shows *measure-pmf.prob* (*tb-S n*) {*h. tb-H x h* = α ∧ *tb-H y h* = β} = *1* / *real* (*card R* ∗ *card R*)
  (**is** *?lhs* = *?rhs*)
proof −
  let *?N* = *N n*
  let *?H* = *H n*
  let *?tb-S* = *tb-S n*

  note *finite-PiE*[*simp*] *PiE-eq-empty-iff*[*simp*] *measure-pmf-single*[*simp*]

  obtain *i* **where** *i*: *x* \$ *i* ≠ *y* \$ *i* **by** (*meson assms vec-cong*)

  let *?xi* = *x* \$ *i*
  let *?yi* = *y* \$ *i*

  have *bij*:
    bij-betw (λ(*a*, *b*, *c*, *d*). *a*(*from-index i* := *b*(*?xi* := *c*, *?yi* := *d*)))
      ((*?N* − {*from-index i*} $\rightarrow_E$ *D* $\rightarrow_E$ *R*) × (*D* − {*?xi*, *?yi*} $\rightarrow_E$ *R*) × *R* × *R*)
      ?H
    using *i* **by** (*intro bij-betw-funcsetE.remove1-remove2*) *simp-all*

  let *?destructure* =
    pair-pmf (*pmf-of-set* (*?N* − {*from-index i*} $\rightarrow_E$ *D* $\rightarrow_E$ *R*))
      (*pair-pmf* (*pmf-of-set* (*D* − {*?xi*, *?yi*} $\rightarrow_E$ *R*))
        (*pmf-of-set* (*R* × *R*)))

  have *asmap*: *?tb-S* =
     map-pmf (λ(*a*, *b*, *c*, *d*). *a*(*from-index i* := *b*(*?xi* := *c*, *?yi* := *d*))) *?destructure*
    unfolding *tb-S-def*
    by (*simp add*: *bij flip*: *pmf-of-set-prod-eq map-pmf-of-set-bij-betw*
        del: *PiE-UNIV UNIV-Times-UNIV*)

  have *?lhs* =
    measure-pmf.prob ?destructure
      (*Sigma UNIV* (λ*h*.
        Sigma UNIV (λ*I*.
          {(*xi′*,*yi′*). *tb-H x* (*h*(*from-index i* := *I*(*?xi* := *xi′*, *?yi* := *yi′*))) = α ∧
                    tb-H y (*h*(*from-index i* := *I*(*?xi* := *xi′*, *?yi* := *yi′*))) = β })))
    unfolding *asmap measure-map-pmf* **by** (*auto intro*: *measure-pmf-cong*)

  also from *i* **have** . . . =

measure-pmf.prob ?destructure
(Sigma UNIV (λh.
Sigma UNIV (λI.
{(*tb-H x* (*h*(*from-index i* := *I*(*?xi* := α, *?yi* := β))),
tb-H y (*h*(*from-index i* := *I*(*?xi* := α, *?yi* := β))))}))))
apply (*clarsimp intro*!: *measure-pmf-cong simp*: *tb-H-discard tb-H-exch'*)
by (*smt* (*verit*, *ccfv-threshold*) *fun-upd-twist tb-H-discard*)

also have ... = *?rhs* **by** (*simp add*: *measure-pmf-prob-dependent-product-bound-eq'*)

finally show *?thesis* **.**
qed

3.3 Proof of 3-independence

lemma *prob-3same-conv*:
assumes *x* \$ *i* $\neq$ *y* \$ *i y* \$ *i* $\neq$ *z* \$ *i z* \$ *i* $\neq$ *x* \$ *i*
shows *measure-pmf.prob* (*tb-S n*) {*h*. *tb-H x h* = α $\wedge$ *tb-H y h* = β $\wedge$ *tb-H z h* = γ} =
1 / (*real* (*card R*) * *real* (*card R*) * *real* (*card R*)) (**is** *?lhs* = *?rhs*)
proof −
let *?N* = *N n*
let *?H* = *H n*
let *?tb-S* = *tb-S n*
let *?xi* = *x* \$ *i*
let *?yi* = *y* \$ *i*
let *?zi* = *z* \$ *i*

note *finite-PiE*[*simp*] *PiE-eq-empty-iff*[*simp*] *measure-pmf-single*[*simp*]

from *assms* **have** *bij*:
bij-betw (λ(*a*, *b*, *c*, *d*, *e*). *a*(*from-index i* := *b*(*?xi* := *c*, *?yi* := *d*, *?zi* := *e*)))
((*?N* − {*from-index i*} $\rightarrow_E$ *D* $\rightarrow_E$ *R*) $\times$ (*D* − {*?xi*, *?yi*, *?zi*} $\rightarrow_E$ *R*) $\times$ *R* $\times$ *R* $\times$ *R*)
?H
by (*fastforce intro*: *bij-betw-funcsetE.remove1-remove3*)

let *?destructure* =
(*pair-pmf* (*pmf-of-set* (*?N* − {*from-index i*} $\rightarrow_E$ *D* $\rightarrow_E$ *R*))
(*pair-pmf* (*pmf-of-set* (*D* − {*?xi*, *?yi*, *?zi*} $\rightarrow_E$ *R*))
(*pmf-of-set* (*R* $\times$ *R* $\times$ *R*))))

have *asmap*: *?tb-S* =
map-pmf (λ(*a*, *b*, *c*, *d*, *e*). *a*(*from-index i* := *b*(*?xi* := *c*, *?yi* := *d*, *?zi* := *e*))) *?destructure*
unfolding *tb-S-def*
by (*simp add*: *bij flip*: *pmf-of-set-prod-eq map-pmf-of-set-bij-betw*
del: *PiE-UNIV UNIV-Times-UNIV*)

have *?lhs* =
measure-pmf.prob ?destructure
(*Sigma UNIV* (λh.
Sigma UNIV (λI.
$\{(xi',yi',zi')$.
tb-H x (*h*(*from-index i* := *I*(*?xi* := *xi'*, *?yi* := *yi'*, *?zi* := *zi'*))) = $\alpha \wedge$
tb-H y (*h*(*from-index i* := *I*(*?xi* := *xi'*, *?yi* := *yi'*, *?zi* := *zi'*))) = $\beta \wedge$
tb-H z (*h*(*from-index i* := *I*(*?xi* := *xi'*, *?yi* := *yi'*, *?zi* := *zi'*))) = $\gamma\}$)))
unfolding *asmap measure-map-pmf* **by** (*auto intro*: *measure-pmf-cong*)

also from *assms* **have** ... =
measure-pmf.prob ?destructure
(*Sigma UNIV* (λh.
Sigma UNIV (λI.
$\{(xi',yi',zi')$.
tb-H x (*h*(*from-index i* := *I*(*?xi* := *xi'*))) = $\alpha \wedge$
tb-H y (*h*(*from-index i* := *I*(*?yi* := *yi'*))) = $\beta \wedge$
tb-H z (*h*(*from-index i* := *I*(*?zi* := *zi'*))) = $\gamma\}$)))
unfolding *asmap measure-map-pmf*
apply (*clarsimp intro*!: *measure-pmf-cong simp*: *tb-H-discard*)
by (*smt* (*verit*, *best*) *fun-upd-twist tb-H-discard*)

also have ... =
(*measure-pmf.prob ?destructure*
(*Sigma UNIV* (λh.
Sigma UNIV (λI.
{*let xi'* = *tb-H x* (*h*(*from-index i* := *I*(*?xi* := α)));
yi' = *tb-H y* (*h*(*from-index i* := *I*(*?yi* := β)));
zi' = *tb-H z* (*h*(*from-index i* := *I*(*?zi* := γ)))
in (*xi'*,*yi'*,*zi'*)}))))
by (*auto simp add*: *tb-H-exch'* *intro*: *measure-pmf-cong conj-cong*)

also have ... = *?rhs* **by** (*simp add*: *measure-pmf-prob-dependent-product-bound-eq'*)

finally show *?thesis* **.**
qed

lemma *prob-2same-conv*:
assumes *x* \$ *i* $\neq$ *y* \$ *i z* \$ *j* $\neq$ *x* \$ *j i* $\neq$ *j*
and *z* \$ *i* = *x* \$ *i*
shows *measure-pmf.prob* (*tb-S n*) {*h*. *tb-H x h* = $\alpha \wedge$ *tb-H y h* = $\beta \wedge$ *tb-H z h*
= γ} =
1 / (*real* (*card R*) * *real* (*card R*) * *real* (*card R*)) (**is** *?lhs* = *?rhs*)
proof −
let *?N* = *N n*
let *?H* = *H n*
let *?tb-S* = *tb-S n*

let *?xi* = *x* \$ *i*

let *?yi* = *y* $ *i*
let *?zi* = *z* $ *i*
let *?xj* = *x* $ *j*
let *?yj* = *y* $ *j*
let *?zj* = *z* $ *j*

note *finite-PiE*[*simp*] *PiE-eq-empty-iff*[*simp*] *measure-pmf-single*[*simp*]

from *assms* **have** *bij*:
bij-betw
(λ(*a*, *b*, *c*, *d*, *e*, *f*). *a*(*from-index i* := *b*(*?xi* := *d*, *?yi* := *e*), *from-index j* := *c*(*?zj* := *f*)))
((*?N* − {*from-index i*, *from-index j*} $\rightarrow_E$ *D* $\rightarrow_E$ *R*) ×
(*D* − {*?xi*, *?yi*} $\rightarrow_E$ *R*) ×
(*D* − {*?zj*} $\rightarrow_E$ *R*) ×
R × *R* × *R*)
?H
by (*fastforce intro*: *bij-betw-funcsetE.remove2-remove2′remove1*)

let *?destructure* =
(*pair-pmf* (*pmf-of-set* (*?N* − {*from-index i*, *from-index j*} $\rightarrow_E$ *D* $\rightarrow_E$ *R*))
(*pair-pmf* (*pmf-of-set* (*D* − {*?xi*, *?yi*} $\rightarrow_E$ *R*))
(*pair-pmf* (*pmf-of-set* (*D* − {*?zj*} $\rightarrow_E$ *R*))
(*pmf-of-set* (*R* × *R* × *R*)))))

have *asmap*: *?tb-S* =
map-pmf
(λ(*a*, *b*, *c*, *d*, *e*, *f*). *a*(*from-index i* := *b*(*?xi* := *d*, *?yi* := *e*), *from-index j* := *c*(*?zj* := *f*)))
?destructure
unfolding *tb-S-def*
by (*simp add*: *bij flip*: *pmf-of-set-prod-eq map-pmf-of-set-bij-betw*
del: *PiE-UNIV UNIV-Times-UNIV*)

have *?lhs* =
(*measure-pmf.prob ?destructure*
(*Sigma UNIV* (λ*h*.
Sigma UNIV (λ*I*.
Sigma UNIV (λ*J*.
{(*xi′*,*yi′*,*zj′*).
tb-H x (*h*(*from-index i* := *I*(*?xi* := *xi′*, *?yi* := *yi′*), *from-index j* := *J*(*?zj* := *zj′*))) = α ∧
tb-H y (*h*(*from-index i* := *I*(*?xi* := *xi′*, *?yi* := *yi′*), *from-index j* := *J*(*?zj* := *zj′*))) = β ∧
tb-H z (*h*(*from-index i* := *I*(*?xi* := *xi′*, *?yi* := *yi′*), *from-index j* := *J*(*?zj* := *zj′*))) = γ
})))))
unfolding *asmap measure-map-pmf* **by** (*auto intro*: *measure-pmf-cong*)

also from *assms* **have** . . . =
(*measure-pmf.prob ?destructure*
(*Sigma UNIV* (λh.
Sigma UNIV (λI.
Sigma UNIV (λJ.
$\{(xi', yi', zj')$.
tb-H x (*h*(*from-index j* := *J*, *from-index i* := *I*(*?xi* := *xi*$'$))) = α $\wedge$
tb-H y (*h*(*from-index j* := *J*(*?zj* := *zj*$'$), *from-index i* := *I*(*?yi* := *yi*$'$))) = β $\wedge$
tb-H z (*h*(*from-index i* := *I*(*?xi* := *xi*$'$), *from-index j* := *J*(*?zj* := *zj*$'$))) = γ
$\}$)))))
unfolding *asmap measure-map-pmf*
apply (*clarsimp intro*!: *measure-pmf-cong dest*!: *from-index-neq simp*: *tb-H-discard*)
by (*smt* (*verit*, *best*) *fun-upd-twist tb-H-discard*)

also have . . . =
measure-pmf.prob ?destructure
(*Sigma UNIV* (λh.
Sigma UNIV (λI.
Sigma UNIV (λJ.
$\{$*let*
xi$'$ = *tb-H x* (*h*(*from-index j* := *J*, *from-index i* := *I*(*?xi* := α)));
zj$'$ = *tb-H z* (*h*(*from-index i* := *I*(*?xi* := *xi*$'$), *from-index j* := *J*(*?zj* := γ)));
yi$'$ = *tb-H y* (*h*(*from-index j* := *J*(*?zj* := *zj*$'$), *from-index i* := *I*(*?yi* := β)))
in (*xi*$'$,*yi*$'$,*zj*$'$)$\}$)))))
by (*auto simp add*: *Let-unfold tb-H-exch*$'$ *intro*: *measure-pmf-cong*)

also have . . . = *?rhs* **by** (*simp add*: *measure-pmf-prob-dependent-product-bound-eq*$'$)

finally show *?thesis* **.**
qed

lemma *prob-tb-H-bin1-bin2-bin3*:
assumes $x \neq y$ $y \neq z$ $z \neq x$
shows *measure-pmf.prob* (*tb-S n*) $\{h$. *tb-H x h* = α $\wedge$ *tb-H y h* = β $\wedge$ *tb-H z h* = $\gamma\}$ =
1 / (*real* (*card R*) $*$ *real* (*card R*) $*$ *real* (*card R*)) (**is** *?lhs* = *?rhs*)
proof −
obtain *i* **where** *i*: *x* \$ *i* $\neq$ *y* \$ *i* **using** *assms*(*1*) *vec-cong* **by** *blast*

let *?xi* = *x* \$ *i*
let *?yi* = *y* \$ *i*
let *?zi* = *z* \$ *i*

consider *?zi* $\neq$ *?xi* $\wedge$ *?zi* $\neq$ *?yi* | *?zi* = *?xi* | *?zi* = *?yi* **by** *blast*
then show *?thesis*
proof *cases*
case *2*
moreover from *assms vec-cong* **obtain** *j* **where** *z* \$ *j* $\neq$ *x* \$ *j* **by** *blast*
ultimately show *?thesis* **using** *i assms* **by** (*auto intro*: *prob-2same-conv*)

next
  case *3*
  moreover obtain *j* **where** $z \$ j \neq y \$ j$ **using** *assms*(*2*) *vec-cong* **by** *blast*
  ultimately show *?thesis*
    using *i assms*
    by (*subst prob-2same-conv*[**where** $y = x$, *symmetric*]) (*auto simp*: *ac-simps*)
qed (*auto intro*!: *prob-3same-conv simp*: *assms i*)
qed

3.4 Proof of 3-universal

lemma *three-indep-vars*:
  shows *prob-space.k-wise-indep-vars* (*tb-S n*) *3* (λ-. *count-space R*) *tb-H* D^n
proof (*rule prob-space.k-wise-indep-varsI*)
  show *prob-space* (*measure-pmf* (*tb-S n*)) **using** *measure-pmf.prob-space-axioms*
by *blast*
  show *measure-pmf* (*tb-S n*) = *measure-pmf* (*tb-S n*) **by** (*rule refl*)
next
  fix *J* :: (*'fragment*, *'n*) *vec set* **assume** *card J* $\leq$ *3 finite J*
  fix *f*
  consider (*0*) *card J* = *0*
      | (*1*) *card J* = *1*
      | (*2*) *card J* = *2*
      | (*3*) *card J* = *3* **using** ‹*card J* $\leq$ *3*› **by** *linarith*
  then show *measure-pmf.prob* (*tb-S n*) {*h*. $\forall$ *key*$\in$*J*. *tb-H key h* = *f key*} =
    ($\prod$ *key*$\in$*J*. *measure-pmf.prob* (*tb-S n*) {*h*. *tb-H key h* = *f key*})
    (**is** *?lhs* = *?rhs*)
  proof *cases*
    case *0*
    then have *J* = {} **using** ‹*finite J*› *card-0-eq* **by** *blast*
    then have *?lhs* = *1* **by** *simp*
    also have ... = *?rhs* **unfolding** ‹*J* = {}› *prod.empty* **..**
    finally show *?thesis* **.**
  next
    case *1*
    then have $\exists$ *key*. *J* = {*key*} **by** (*meson card-1-singletonE*)
    then obtain *key* **where** [*simp*]: *J* = {*key*} **by** *blast*
    — Since there is only one value inhabiting J, we can erase the quantifier
    then have *?lhs* = *measure-pmf.prob* (*tb-S n*) {*h*. *tb-H key h* = *f key*} **by** *simp*
    also have ... = *?rhs* **by** *simp*
    finally show *?thesis* **.**
  next
    case *2*
    then have $\exists$ *k1 k2*. *J* = {*k1*,*k2*} $\wedge$ *k1* $\neq$ *k2* **by** (*meson card-2-iff*)
    then obtain *k1 k2* **where** [*simp*]: *J* = {*k1*,*k2*} *k1* $\neq$ *k2* **by** *blast+*

    have *?lhs* = *measure-pmf.prob* (*tb-S n*) {*h*. *tb-H k1 h* = *f k1* $\wedge$ *tb-H k2 h* = *f*
k2} **by** *simp*
    also have ... = *1* / (*real* (*card R*) $*$ *real* (*card R*)) **using** *prob-tb-H-bin1-bin2*

by *simp*
also have ... = ($\prod$ *key*∈*J*. *indicat-real R* (*f key*) / *real* (*card R*)) **by** *simp*
also have ... = *?rhs* **using** *prob-tb-H-bin* **by** *fastforce*
finally show *?thesis* **.**
next
case *3*
then have ∃ *k1 k2 k3*. *J* = {*k1*,*k2*,*k3*} ∧ *k1* ≠ *k2* ∧ *k2* ≠ *k3* ∧ *k3* ≠ *k1* **by** (*metis card-3-iff*)
then obtain *k1 k2 k3* **where** [*simp*]: *J* = {*k1*,*k2*,*k3*} *k1* ≠ *k2 k2* ≠ *k3 k3* ≠ *k1 k1* ≠ *k3*
by *blast+*

have *?lhs* = *measure-pmf.prob* (*tb-S n*)
{*h*. *tb-H k1 h* = *f k1* ∧ *tb-H k2 h* = *f k2* ∧ *tb-H k3 h* = *f k3*} **by** *simp*
also have ... = *1* / (*real* (*card R*) ∗ *real* (*card R*) ∗ *real* (*card R*))
using *prob-tb-H-bin1-bin2-bin3* **by** *simp*
also have ... = ($\prod$ *key*∈*J*. *indicat-real R* (*f key*) / *real* (*card R*)) **by** *simp*
also have ... = *?rhs* **using** *prob-tb-H-bin* **by** *fastforce*
finally show *?thesis* **.**
qed
qed

theorem *three-universal*:
shows *prob-space.k-universal* (*tb-S n*) *3 tb-H* D^n *R*
using *prob-space.k-universal-def measure-pmf.prob-space-axioms three-indep-vars uniform* **by** *blast*

3.5 Proof of non-4-universal

lemma *prob-4-conv*:
fixes *u v* :: *'fragment*
assumes *W* ⊕ *X* ⊕ *Y* ⊕ *Z* = *0* ⋀*h*. *P* (*h*(*0* := *undefined*, *Suc 0* := *undefined*)) = *P h*
CARD(*'n*) > *Suc 0 u* ≠ *v*
shows *measure-pmf.prob* (*tb-S n*) {*h*::*nat* ⇒ *'fragment* ⇒ (*'result*, *'q*) *vec*.
h 0 u ⊕ *h* (*Suc 0*) *u* ⊕ *P h* = *W* ∧
h 0 u ⊕ *h* (*Suc 0*) *v* ⊕ *P h* = *X* ∧
h 0 v ⊕ *h* (*Suc 0*) *u* ⊕ *P h* = *Y* ∧
h 0 v ⊕ *h* (*Suc 0*) *v* ⊕ *P h* = *Z* }
= *1* / (*real* (*card R*) ∗ *real* (*card R*) ∗ *real* (*card R*)) (**is** *?lhs* = *?rhs*)
proof −
let *?N* = *N n*
let *?H* = *H n*
let *?tb-S* = *tb-S n*

note *assms*(*1*,*2*) [*simp*]
note *finite-PiE*[*simp*] *PiE-eq-empty-iff*[*simp*] *measure-pmf-single*[*simp*]

define *W′ X′ Y′ Z′* **where**

$W' \equiv \lambda h.\ W \oplus P\ h$
$X' \equiv \lambda h.\ X \oplus P\ h$
$Y' \equiv \lambda h.\ Y \oplus P\ h$
$Z' \equiv \lambda h.\ Z \oplus P\ h$

have *xor-sum*: $W'\ h \oplus X'\ h \oplus Y'\ h \oplus Z'\ h = 0$ **for** *h*
  by (*simp add*: *W′-X′-Y′-Z′-def commute left-commute*)

have *bij*:
  bij-betw
    $(\lambda(a,\ b,\ c,\ d,\ e,\ f).\ a(Suc\ 0 := b(u := d,\ v := e),\ 0 := c(v := f)))$
    $((?N - \{Suc\ 0,\ 0\} \rightarrow_E D \rightarrow_E R) \times$
      $(D - \{u,\ v\} \rightarrow_E R) \times$
      $(D - \{v\} \rightarrow_E R) \times$
      $R \times R \times R)$
    ?H
  using *assms* **by** (*intro bij-betw-funcsetE.remove2-remove2′remove1*) *simp-all*

let *?destructure* =
  (*pair-pmf* (*pmf-of-set* $(?N - \{Suc\ 0,\ 0\} \rightarrow_E D \rightarrow_E R)$)
    (*pair-pmf* (*pmf-of-set* $(D - \{u,\ v\} \rightarrow_E R)$)
      (*pair-pmf* (*pmf-of-set* $(D - \{v\} \rightarrow_E R)$)
        (*pmf-of-set* $(R \times R \times R)$))))

have *asmap*: *?tb-S* =
  map-pmf
    $(\lambda(a,\ b,\ c,\ d,\ e,\ f).\ a(1 := b(u := d,\ v := e),\ 0 := c(v := f)))$
    ?destructure
  unfolding *tb-S-def*
  by (*simp add*: *bij flip*: *pmf-of-set-prod-eq map-pmf-of-set-bij-betw*
      del: *PiE-UNIV UNIV-Times-UNIV*)

have *?lhs* =
  (*measure-pmf.prob ?destructure*
    (*Sigma UNIV* (λ*h*.
      Sigma UNIV (λ*I1*.
        Sigma UNIV (λ*I0*.
          $\{(u1,v1,v0).$
            $I0\ u \oplus u1 \oplus P\ h = W\ \wedge$
            $I0\ u \oplus v1 \oplus P\ h = X\ \wedge$
            $v0\ \ \oplus u1 \oplus P\ h = Y\ \wedge$
            $v0\ \ \oplus v1 \oplus P\ h = Z\})))))$
  unfolding *asmap measure-map-pmf*
  apply (*clarsimp intro*!: *measure-pmf-cong simp add*: *assms*(4))
  by (*smt* (*z3*) *array-rules*(5) *assms*(2) *fun-upd-twist*)

also have ... =
  measure-pmf.prob ?destructure

```
    (Sigma UNIV (λh.
      Sigma UNIV (λI1.
        Sigma UNIV (λI0.
          {(u1,v1,v0).
            I0 u ⊕ u1 = W' h ∧
            I0 u ⊕ v1 = X' h ∧
            v0   ⊕ u1 = Y' h ∧
            v0   ⊕ v1 = Z' h }))))
   apply (clarsimp intro!: measure-pmf-cong)
   unfolding W'-X'-Y'-Z'-def
   by (smt (verit, best) assoc right-neutral self-inv)

  also have ... =
   measure-pmf.prob ?destructure
    (Sigma UNIV (λh.
      Sigma UNIV (λI1.
        Sigma UNIV (λI0.
          {(I0 u ⊕ W' h,
            I0 u ⊕ X' h,
            I0 u ⊕ W' h ⊕ Y' h)}))))
   apply (clarsimp intro!: measure-pmf-cong)
   by (smt (z3) xor-sum assoc abel-group-xor-class.eq-iff)

  also have ... = ?rhs
   apply (subst measure-pmf-prob-dependent-product-bound-eq'[where r = ?rhs])+
   by simp-all

  finally show ?thesis .
qed

lemma not-four-indep:
  assumes CARD('n) > 1 — if n = 1, then tabulation hashing is 4-independent
         card D ≥ 2 — if D =  or D = x, then it is impossible to obtain 4 distinct
keys
         card R > 1 — tabulation hashing is 4-independent otherwise
  shows ¬ prob-space.k-wise-indep-vars (tb-S n) 4 (λ-. count-space R) tb-H D^n
proof −
  let ?tb-S = tb-S n
  fix f :: ('fragment, 'n) vec ⇒ ('result, 'q) vec

  obtain u v :: 'fragment where pq [simp]: u ≠ v by (meson assms(2) UNIV-I
card-2-iff' ex-card)
  define u-n where u-n ≡ replicate (CARD('n) − 2) u

  let ?w = u # u # u-n
  let ?x = u # v # u-n
  let ?y = v # u # u-n
  let ?z = v # v # u-n
```

define *w x y z* :: (*'fragment*, *'n*) *vec* **where** *wxyz*:
w = *vec-of-list ?w x* = *vec-of-list ?x y* = *vec-of-list ?y z* = *vec-of-list ?z*
define *J* **where** [*simp*]: *J* = {*w*,*x*,*y*,*z*}

have *N*: *a* # *b* # *u-n* ∈ {*xs. length xs* = *CARD*(*'n*)} **for** *a b*
unfolding *u-n-def* **using** *assms*(*1*) **by** *simp*

have *distinct* [*simp*]: *w* ≠ *x w* ≠ *y w* ≠ *z x* ≠ *y x* ≠ *z y* ≠ *z*
using *N* **by** (*simp-all add*: *wxyz vec-of-list-inject*)

have [*simp*]: *card J* = *4* **by** *simp*

have [*simp*]:
w \$ *to-index 0* = *u w* \$ *to-index* (*Suc 0*) = *u*
x \$ *to-index 0* = *u x* \$ *to-index* (*Suc 0*) = *v*
y \$ *to-index 0* = *v y* \$ *to-index* (*Suc 0*) = *u*
z \$ *to-index 0* = *v z* \$ *to-index* (*Suc 0*) = *v*
using *N* **by** (*simp-all add*: *wxyz nth-vec.rep-eq vec-of-list-inverse to-from-index*)

have *rw*:
xor-fold (*map* (*λi. h i* ((*a* # *b* # *u-n*) ! *from-index* (*to-index i* :: *'n*))) [*0*..<*CARD*(*'n*)])
=
h 0 a ⊕
h (*Suc 0*) *b* ⊕
xor-fold (*map* (*λi. h i u*) [*Suc* (*Suc 0*)..<*CARD*(*'n*)])
(**is** *xor-fold ?a* = -) **for** *a b h*
proof −
have *?a* = *h 0 a* # *h* (*Suc 0*) *b* # *map* (*λi. h i u*) [*Suc* (*Suc 0*) ..< *CARD*(*'n*)]
using *assms*(*1*) **by** (*auto simp*: *upt-conv-Cons to-from-index u-n-def*)
then show *?thesis* **by** *simp*
qed

have ¬ *prob-space.indep-vars ?tb-S* (*λ-. count-space UNIV*) *tb-H J*
proof −
have *measure-pmf.prob ?tb-S* {*h.* ∀ *key*∈*J. tb-H key h* = *f key*}
≠ *1*/(*card R* ∗ *card R* ∗ *card R* ∗ *card R*) (**is** *?lhs* ≠ . . .)
proof (*cases f w* ⊕ *f x* ⊕ *f y* ⊕ *f z* = *0*)
case *True*

In this case, we want to show that the probability is $\frac{1}{R^3}$ and not $\frac{1}{R^4}$

let *?a* = *λh. xor-fold* (*map* (*λi. h i u*) [*Suc* (*Suc 0*)..<*CARD*(*'n*)])

have *?lhs* = *measure-pmf.prob ?tb-S* {*h.*
h 0 u ⊕ *h* (*Suc 0*) *u* ⊕ *?a h* = *f w* ∧
h 0 u ⊕ *h* (*Suc 0*) *v* ⊕ *?a h* = *f x* ∧

h 0 v ⊕ *h* (*Suc 0*) *u* ⊕ *?a h* = *f y* ∧
h 0 v ⊕ *h* (*Suc 0*) *v* ⊕ *?a h* = *f z*}
using *N* **by** (*simp add*: *tb-H-def wxyz nth-vec.rep-eq vec-of-list-inverse rw to-from-index*)

also have ... = *1* / *real* (*card R* ∗ *card R* ∗ *card R*)
using *assms*(*1*) *True* **by** (*auto intro*!: *prob-4-conv arg-cong*[**where** *f* = *xor-fold*])

finally show *?thesis* **using** *assms*(*3*) **by** *simp*
next
case *False*

have *False* **if** *tb-H w h* = *f w tb-H x h* = *f x tb-H y h* = *f y tb-H z h* = *f z* **for** *h*
proof −
have *tb-H w h* ⊕ *tb-H x h* ⊕ *tb-H y h* ⊕ *tb-H z h* = *0*
using *N* **by** (*simp add*: *tb-H-def wxyz nth-vec.rep-eq vec-of-list-inverse rw to-from-index ac-simps*)
then show *False* **using** *that False* **by** *simp*
qed
then have *?lhs* = *0* **by** (*auto simp*: *measure-pmf-zero-iff*)
then show *?thesis* **by** *simp*
qed

moreover have ... = (∏ *key*∈*J*. *measure-pmf.prob ?tb-S* {*h*. *tb-H key h* = *f key*})
using *prob-tb-H-bin* **by** *fastforce*

ultimately show *?thesis*
by (*intro measure-pmf.indep-vars-pmf-contrapos*[**where** *P* = λ *x y*. *y* = *f x*]) *simp-all*
qed

then show *?thesis* **by** (*force simp*: *prob-space.k-wise-indep-vars-def prob-space-measure-pmf*)
qed

theorem *not-four-universal*:
assumes *CARD*(*'n*) > *1* — if n = 1, then tabulation hashing is 4-independent
card D ≥ *2* — if D = or D = x, then it is impossible to obtain 4 distinct keys
card R > *1* — tabulation hashing is 4-independent otherwise
shows ¬ *prob-space.k-universal* (*tb-S n*) *4* *tb-H* D^n *R*
using *not-four-indep*[*OF assms*]
by (*simp add*: *prob-space.k-universal-def measure-pmf.prob-space-axioms*)

end

4 Appendix

4.1 Utility

context *prob-space*
begin

context
  fixes *K D k p*
  assumes *assms'*: $K \subseteq D$ *card* $K \leq k$ *finite* K M = *measure-pmf p*
begin

lemma *k-wise-indep-vars-probD*:
  assumes *k-wise-indep-vars k* (λ-. *count-space UNIV*) *X D*
  shows *prob* {ω. ∀ x∈K. P x (X x ω)} = (∏ x∈K. *prob* {ω. P x (X x ω)})
  using *assms' assms* **unfolding** *k-wise-indep-vars-def* **by** (*subst split-indep-events*[*of* - *X K*]) *auto*

lemma
  assumes *k-universal k X D R*
  shows
    k-universal-probD:
      ⋀ P. *prob* {ω. ∀ x∈K. P x (X x ω)} = (∏ x∈K. *prob* {ω. P x (X x ω)})
      (**is** *PROP ?thesis-0*) **and**
    k-universal-probD':
      prob {ω. ∀ x∈K. X x ω = P x} = (∏ x∈K. *indicat-real R* (P x) / *real* (*card* R))
      (**is** *?thesis-1*)
proof −
  from *assms k-wise-indep-vars-probD* **show** *PROP ?thesis-0*
    unfolding *k-universal-def* **by** *fastforce*
  from *this*[*of* λ x y. y = P x] *uniform-onD*[**where** A = R **and** B = {P -}] *assms' assms*
  show *?thesis-1*
    unfolding *k-universal-def*
    by (*auto intro*!: *prod.cong split*: *split-indicator simp*: *prob-space-measure-pmf*)
qed

end
end

locale *simple-tabulation-hashing'* = *simple-tabulation-hashing* +
  assumes *r*: *r* = *TYPE*(*'result*::{*finite*,*abel-group-xor*})
  assumes *q*: *q* = *TYPE*(*'q*::*index1*)
  assumes *d*: *d* = *TYPE*(*'fragment*::*finite*)
begin

end

4.2 Alternate proof of non-4-universal

4.2.1 Preliminaries

lemma (**in** *prob-space*) *finite-if-k-universal*:
  assumes *k-universal k X D R* $D \neq \{\}$
  shows *finite R*
  using *assms* **by** (*auto simp add*: *k-universal-def uniform-on-def*)

lemma *xor-sum-uniform*:
  fixes α :: *'a* :: {*finite*,*abel-group-xor*}
  defines $R \equiv UNIV$
  assumes [*simp*]: *finite J* $J \neq \{\}$
  shows *measure-pmf.prob* (*pmf-of-set* ($J \rightarrow_E R$)) $\{f.\ (\bigoplus x{\in}J.\ f\ x) = \alpha\} = 1\ /$ *real CARD('a)*
  (**is** *?lhs* = *?rhs*)
proof −
  note *finite-PiE*[*simp*] *PiE-eq-empty-iff*[*simp*] *measure-pmf-single*[*simp*]

  obtain *a* **where** *a*: $a \in J$ **and** *bij*: *bij-betw* ($\lambda(f,\ v).\ f(a := v)$) $((J - \{a\} \rightarrow_E R) \times R)\ (J \rightarrow_E R)$
    using *bij-betw-funcsetE.bij-PiE-remove-point*[*of - J R*] *assms*(*3*) **by** *blast*

  have [*simp*]: $R \neq \{\}$ **using** *assms*(*1*) **by** *blast*

  have [*simp*]: *set-pmf* (*pmf-of-set* ($J \rightarrow_E R$)) = $J \rightarrow_E R$
        set-pmf (*pmf-of-set* ($J - \{a\} \rightarrow_E R$)) = $J - \{a\} \rightarrow_E R$
        set-pmf (*pmf-of-set R*) = *R*
    by (*auto intro*!: *set-pmf-of-set*)

  let *?destructure* =
    (*pair-pmf* (*pmf-of-set* ($J - \{a\} \rightarrow_E R$))
      (*pmf-of-set R*))

  have *asmap*: *pmf-of-set* ($J \rightarrow_E R$) = *map-pmf* ($\lambda(f,\ v).\ f(a := v)$) *?destructure*
    using *assms bij* **by** (*simp flip*: *pmf-of-set-prod-eq map-pmf-of-set-bij-betw*
                del: *PiE-UNIV UNIV-Times-UNIV*)

  have *?lhs* = *measure-pmf.prob* (*pmf-of-set* ($J \rightarrow_E R$)) $\{f{\in}J \rightarrow_E R.\ (\bigoplus x{\in}J.\ f\ x) = \alpha\}$
    apply (*subst measure-Int-set-pmf*[*symmetric*])
    by (*auto intro*!: *arg-cong*[**where** *?f* = *measure-pmf.prob* -])

  also have ... =
    measure-pmf.prob ?destructure
      (*Sigma* ($J - \{a\} \rightarrow_E R$) ($\lambda f.\ \{x.\ x \oplus (\bigoplus x{\in}J - \{a\}.\ f\ x) = \alpha\}$))
    using *PiE-fun-upd a* **by** (*auto simp add*: *asmap xor-sum.remove*[**where** *?x* =

a, *OF assms*(*2*) *a*]
intro!: *measure-pmf-cong*)

also have ... =
measure-pmf.prob ?destructure
(*Sigma* (*J* − {*a*} $\to_E$ *R*) (λ*f*. {α ⊕ (⨁*x*∈*J*− {*a*}. *f x*)}))
by (*fastforce intro*: *measure-pmf-cong simp add*: *ac-simps*)

also have ... = *?rhs*
by (*simp add*: *assms*(*1*) *measure-pmf-prob-dependent-product-bound-eq′ measure-pmf-of-set*)

finally show *?lhs* = *?rhs* **.**
qed

lemma *k-universal-conv-pmf-of-set*:
defines *R* ≡ *UNIV*
assumes *prob-space.k-universal* (*measure-pmf p*) *k X D R*
and *J* ⊆ *D card J* ≤ *k finite J D* ≠ {}
shows *map-pmf* (λω. λ*x*∈*J*. *X x* ω) *p* = *pmf-of-set* (*J* $\to_E$ *R*)
proof (*intro pmf-eqI*)
fix *P*
from *assms*(*2*) **have** *fR*: *finite R* **by** (*simp add*: *assms*(*6*) *measure-pmf.finite-if-k-universal*)

{
assume *P*: *P* ∈ *J* $\to_E$ *R*

have *pmf* (*map-pmf* (λω. λ*x*∈*J*. *X x* ω) *p*) *P* = *measure-pmf.prob p* {ω. (λ*x*∈*J*. *X x* ω) = *P*}
by (*simp add*: *pmf-map vimage-def*)

also have ... = *measure-pmf.prob p* {ω. ∀ *x*∈*J*. *X x* ω = *P x*}
using *P* **by** (*fastforce intro*: *arg-cong*[**where** *?f* = *measure-pmf.prob* -])

also have ... = (∏*x*∈*J*. *indicat-real R* (*P x*) / *real* (*card R*))
using *assms* **by** (*simp add*: *measure-pmf.k-universal-probD′*)

also have ... = (*1* / *real* (*card R*)) ^ *card J*
apply (*subst prod.cong*[**where** *?B*= *J* **and** *?h* = λ-. *1* / *real* (*card R*)]; *simp*)
by (*metis P indicator-simps*(*1*) *PiE-E*)

also have ... = *indicat-real* (*J* $\to_E$ *R*) *P* / *real* (*card* (*J* $\to_E$ *R*))
by (*simp add*: *P assms*(*5*) *card-funcsetE power-one-over*)

also have ... = *pmf* (*pmf-of-set* (*J* $\to_E$ *R*)) *P*
using *P* **by** (*subst pmf-of-set*) (*auto simp add*: *assms*(*5*) *fR finite-PiE*)

finally have *pmf* (*map-pmf* ($\lambda\omega$. $\lambda x{\in}J$. *X x* ω) *p*) *P* = *pmf* (*pmf-of-set* (*J* $\rightarrow_E$ *R*)) *P* **.**
}

moreover {
assume *P*: *P* $\notin$ *J* $\rightarrow_E$ *R*

have *pmf* (*map-pmf* ($\lambda\omega$. $\lambda x{\in}J$. *X x* ω) *p*) *P* = *measure-pmf.prob p* {ω. ($\lambda x{\in}J$. *X x* ω) = *P*}
by (*simp add*: *pmf-map vimage-def*)

also have ... = *0*
proof (*subst measure-pmf-zero-iff*)
have {ω. ($\lambda x{\in}J$. *X x* ω) = *P*} = {}
using *P assms*(*1*)
by (*auto simp add*: *measure-pmf.prob-space-axioms prob-space.k-universal-def*)

then show *set-pmf p* $\cap$ {ω. ($\lambda x{\in}J$. *X x* ω) = *P*} = {} **by** *blast*
qed

also have ... = *pmf* (*pmf-of-set* (*J* $\rightarrow_E$ *R*)) *P*
apply (*rule sym*)
apply (*subst pmf-eq-0-set-pmf*)
apply (*subst set-pmf-of-set*)
subgoal by (*simp add*: *PiE-eq-empty-iff assms*(*1*))
subgoal by (*simp add*: *assms*(*5*) *fR finite-PiE*)
using *P* **.**

finally have *pmf* (*map-pmf* ($\lambda\omega$. $\lambda x{\in}J$. *X x* ω) *p*) *P* = *pmf* (*pmf-of-set* (*J* $\rightarrow_E$ *R*)) *P* **.**
}

ultimately show *pmf* (*map-pmf* ($\lambda\omega$. $\lambda x{\in}J$. *X x* ω) *p*) *P* = *pmf* (*pmf-of-set* (*J* $\rightarrow_E$ *R*)) *P* **by** *blast*
qed

lemma *k-universal-imp-xor-sum-uniform*:
fixes α :: *'a* :: {*finite*,*abel-group-xor*}
defines *R* $\equiv$ *UNIV*
assumes *prob-space.k-universal* (*measure-pmf p*) *k X D R*
and *J* $\subseteq$ *D card J* $\leq$ *k finite J J* $\neq$ {}
shows *measure-pmf.prob p* {ω. ($\bigoplus x{\in}J$. *X x* ω) = α} = *1* / *real* (*card R*)
(**is** *?lhs* = *?rhs*)
proof −
have *?lhs* = *measure-pmf.prob* (*map-pmf* ($\lambda\omega$. $\lambda x{\in}J$. *X x* ω) *p*) {*f*. ($\bigoplus x{\in}J$. *f x*) = α} **by** *simp*
also have ... = *measure-pmf.prob* (*pmf-of-set* (*J* $\rightarrow_E$ *R*)) {*f*. ($\bigoplus x{\in}J$. *f x*) = α}
using *assms* **by** (*intro arg-cong2*[**where** *?f* = *measure-pmf.prob*])

(*auto intro*!: *k-universal-conv-pmf-of-set*)
also have . . . = *?rhs* **using** *assms* **by** (*auto intro*!: *xor-sum-uniform*)
finally show *?lhs* = *?rhs* **.**
qed

4.2.2 Proofs

context *simple-tabulation-hashing* **begin**

lemma *k-wise-indep-vars-imp-xor-sum-uniform*:
assumes *prob-space.k-wise-indep-vars* (*tb-S n*) *k* (λ-. *count-space R*) *tb-H* D^n
$J \neq \{\}$ *card* $J \leq k$
shows *measure-pmf.prob* (*tb-S n*) {*h*. ($\bigoplus x \in J$. *tb-H x h*) = α} = *1* / *real* (*card R*)
using *assms*
by (*auto*
intro: *k-universal-imp-xor-sum-uniform*[**where** *?k* = *k* **and** *?D* = D^n]
simp: *prob-space.k-universal-def prob-space-measure-pmf uniform*)

lemma *not-k-wise-indep-vars-by-xor-sum*:
assumes *measure-pmf.prob* (*tb-S n*) {*h*. ($\bigoplus x \in J$. *tb-H x h*) = α} ≠ *1* / *real* (*card R*)
$J \neq \{\}$ *card* $J \leq k$
shows ¬ *prob-space.k-wise-indep-vars* (*tb-S n*) *k* (λ-. *count-space R*) *tb-H* D^n
using *assms*(*1*) **apply** (*rule contrapos-nn*)
using *assms* **by** (*simp add*: *k-wise-indep-vars-imp-xor-sum-uniform*)

lemma *not-four-indep′*:
assumes *CARD*(′*n*) > *1* — if n = 1, then tabulation hashing is 4-independent
card D ≥ *2* — if D = or D = x, then it is impossible to obtain 4 distinct keys
card R > *1* — tabulation hashing is 4-independent otherwise
shows ¬ *prob-space.k-wise-indep-vars* (*tb-S n*) *4* (λ-. *count-space R*) *tb-H* D^n
proof −
obtain *u v* :: ′*fragment* **where** *pq* [*simp*]: *u* ≠ *v* **by** (*meson assms*(*2*) *UNIV-I card-2-iff′ ex-card*)
define *u-n* **where** *u-n* ≡ *replicate* (*CARD*(′*n*) − *2*) *u*
let *?w* = *u* # *u* # *u-n*
let *?x* = *u* # *v* # *u-n*
let *?y* = *v* # *u* # *u-n*
let *?z* = *v* # *v* # *u-n*

define *w x y z* :: (′*fragment*, ′*n*) *vec* **where** *wxyz*:
w = *vec-of-list ?w x* = *vec-of-list ?x y* = *vec-of-list ?y z* = *vec-of-list ?z*
define *J* **where** [*simp*]: *J* = {*w*,*x*,*y*,*z*}

have *N*: *a* # *b* # *u-n* ∈ {*xs*. *length xs* = *CARD*(′*n*)} **for** *a b*
using *assms*(*1*) **by** (*simp add*: *u-n-def*)

have [*simp*]: $w \neq x$ $w \neq y$ $w \neq z$ $x \neq y$ $x \neq z$ $y \neq z$
  using *N* **by** (*simp-all add*: *vec-of-list-inject wxyz*)

have [*simp*]: *card* $J = 4$ **by** *simp*

have *rw*:
xor-fold (*map* ($\lambda i.$ *h i* ((*a* # *b* # *u-n*) ! *from-index* (*to-index i* :: '*n*))) [*0*..<*CARD*('*n*)])
=
  h 0 a ⊕
  h (*Suc 0*) *b* ⊕
  xor-fold (*map* ($\lambda i.$ *h i u*) [*Suc* (*Suc 0*)..<*CARD*('*n*)])
 (**is** *xor-fold ?a* = -) **for** *a b h*
proof −
  have *?a* = *h 0 a* # *h* (*Suc 0*) *b* # *map* ($\lambda i.$ *h i u*) [*Suc* (*Suc 0*) ..< *CARD*('*n*)]
   using *assms*(*1*) **by** (*auto simp*: *upt-conv-Cons to-from-index u-n-def*)
  then show *?thesis* **by** *simp*
qed

from *N* **have** *measure-pmf.prob* (*tb-S n*) {*h*. ($\bigoplus x \in J.$ *tb-H x h*) = *0*} = *1*
  apply (*simp add*: *tb-H-def nth-vec.rep-eq del*: *eq-iff*)
  by (*simp add*: *wxyz nth-vec.rep-eq vec-of-list-inverse rw ac-simps*)

then show *?thesis*
  using *assms*(*3*) **by** (*intro not-k-wise-indep-vars-by-xor-sum*[*of J 0*]) *simp-all*
qed
end
end

References

[1] D. Mareek. NTIN066 Data Structures 1. https://ufal.mff.cuni.cz/courses/ntin066, Apr 2023. Solutions 8 is relevant for simple tabulation hashing.

[2] M. Patrascu and M. Thorup. The power of simple tabulation hashing. In *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing*, STOC '11, pages 1–10, New York, NY, USA, 2011. Association for Computing Machinery. numpages: 10, keywords: tabulation hashing, minwise independence, linear probing, independence, cuckoo hashing, concentration bounds, location: San Jose, California, USA.

[3] Y. K. Tan and J. Yang. Approximate model counting. *Archive of Formal Proofs*, March 2024. https://isa-afp.org/entries/Approximate_Model_Counting.html, Formal proof development.

[4] Wikipedia contributors. Tabulation hashing. https://en.wikipedia.org/wiki/Tabulation_hashing#Universality, Sep 2024. Accessed: 2026-03-28.