Universal Hash Families

Emin Karayel

March 17, 2025

Abstract

A k-universal hash family is a probability space of functions, which have uniform distribution and form k-wise independent random variables.

They can often be used in place of classic (or cryptographic) hash functions and allow the rigorous analysis of the performance of randomized algorithms and data structures that rely on hash functions.

In 1981 Wegman and Carter [4] introduced a generic construction for such families with arbitrary k using polynomials over a finite field. This entry contains a formalization of them and establishes the property of k-universality.

To be useful the formalization also provides an explicit construction of finite fields using the factor ring of integers modulo a prime. Additionally, some generic results about independent families are shown that might be of independent interest.

1 Introduction and Definition

theory *Universal-Hash-Families*
 imports *HOL−Probability.Independent-Family*
begin

Universal hash families are commonly used in randomized algorithms and data structures to randomize the input of algorithms, such that probabilistic methods can be employed without requiring any assumptions about the input distribution.

If we regard a family of hash functions from a domain D to a finite range R as a uniform probability space, then the family is k-universal if:

- For each $x \in D$ the evaluation of the functions at x forms a uniformly distributed random variable on R.
- The evaluation random variables for k or fewer distinct domain elements form an independent family of random variables.

This definition closely follows the definition from Vadhan [3, §3.5.5], with the minor modification that independence is required not only for exactly k, but also for *fewer* than k distinct domain elements. The correction is due to the fact that in the corner case where D has fewer than k elements, the second part of their definition becomes void. In the formalization this helps avoid an unnecessary assumption in the theorems.

The following definition introduces the notion of k-wise independent random variables:

definition (**in** *prob-space*) *k-wise-indep-vars* **where**
  k-wise-indep-vars k M′ X I =
    (∀ *J* ⊆ *I*. *card J* ≤ *k* ⟶ *finite J* ⟶ *indep-vars M′ X J*)

lemma (**in** *prob-space*) *k-wise-indep-vars-subset*:
  assumes *k-wise-indep-vars k M′ X I*
  assumes *J* ⊆ *I*
  assumes *finite J*
  assumes *card J* ≤ *k*
  shows *indep-vars M′ X J*
  using *assms*
  by (*simp add*:*k-wise-indep-vars-def*)

lemma (**in** *prob-space*) *k-wise-indep-subset*:
  assumes *J* ⊆ *I*
  assumes *k-wise-indep-vars k M′ X′ I*
  shows *k-wise-indep-vars k M′ X′ J*
  using *assms* **unfolding** *k-wise-indep-vars-def* **by** *simp*

Similarly for a finite non-empty set A the predicate *uniform-on X A* indicates that the random variable is uniformly distributed on A:

definition (**in** *prob-space*) *uniform-on X A* = (
  distr M (*count-space UNIV*) *X* = *uniform-measure* (*count-space UNIV*) *A* ∧
  A ≠ {} ∧ *finite A* ∧ *random-variable* (*count-space UNIV*) *X*)

lemma (**in** *prob-space*) *uniform-onD*:
  assumes *uniform-on X A*
  shows *prob* {ω ∈ *space M*. *X* ω ∈ *B*} = *card* (*A* ∩ *B*) / *card A*
proof −
  have *prob* {ω ∈ *space M*. *X* ω ∈ *B*} = *prob* (*X* −‘ *B* ∩ *space M*)
    by (*subst Int-commute*, *simp add*:*vimage-def Int-def*)
  also have ... = *measure* (*distr M* (*count-space UNIV*) *X*) *B*
    using *assms* **by** (*subst measure-distr*, *auto simp*:*uniform-on-def*)
  also have ... = *measure* (*uniform-measure* (*count-space UNIV*) *A*) *B*
    using *assms* **by** (*simp add*:*uniform-on-def*)
  also have ... = *card* (*A* ∩ *B*) / *card A*
    using *assms* **by** (*subst measure-uniform-measure*, *auto simp*:*uniform-on-def*)+
  finally show *?thesis* **by** *simp*
qed

With the two previous definitions it is possible to define the k-universality condition for a family of hash functions from D to R:

definition (**in** *prob-space*) *k-universal* k X D R = (
  k-wise-indep-vars k (λ-. *count-space UNIV*) X D $\wedge$
  ($\forall$ i $\in$ D. *uniform-on* (X i) R))

Note: The definition is slightly more generic then the informal specification from above. This is because usually a family is formed by a single function with a variable seed parameter. Instead of choosing a random function from a probability space, a random seed is chosen from the probability space which parameterizes the hash function.

The following section contains some preliminary results about independent families of random variables. Section 3 introduces the Carter-Wegman hash family, which is an explicit construction of k-universal families for arbitrary k using polynomials over finite fields. The last section contains a proof that the factor ring of the integers modulo a prime ideal is a finite field, followed by an isomorphic construction of prime fields over an initial segment of the natural numbers.

end

2 Preliminary Results

theory *Universal-Hash-Families-More-Independent-Families*
  imports
    Universal-Hash-Families
    HOL−Probability.Stream-Space
    HOL−Probability.Probability-Mass-Function
begin

lemma *set-comp-image-cong*:
  assumes $\bigwedge x.\ P\ x \Longrightarrow f\ x = h\ (g\ x)$
  shows $\{f\ x |\ x.\ P\ x\} = h$ ` $\{g\ x |\ x.\ P\ x\}$
  using *assms* **by** (*auto simp*: *setcompr-eq-image*)

lemma (**in** *prob-space*) *k-wise-indep-vars-compose*:
  assumes *k-wise-indep-vars* k M' X I
  assumes $\bigwedge i.\ i \in I \Longrightarrow Y\ i \in$ *measurable* $(M'\ i)\ (N\ i)$
  shows *k-wise-indep-vars* k N $(\lambda i\ x.\ Y\ i\ (X\ i\ x))$ I
  using *indep-vars-compose2*[**where** N=N **and** X=X **and** Y=Y **and** M'=M']
assms
  by (*simp add*: *k-wise-indep-vars-def subsetD*)

lemma (**in** *prob-space*) *k-wise-indep-vars-triv*:
  assumes *indep-vars* N T I
  shows *k-wise-indep-vars* k N T I
  using *assms indep-vars-subset* **unfolding** *k-wise-indep-vars-def* **by** *auto*

The following two lemmas are of independent interest, they help infer independence of events and random variables on distributions. (Candidates for *HOL−Probability.Independent-Family*).

lemma (**in** *prob-space*) *indep-sets-distr*:
  fixes A
  assumes *random-variable* $N\ f$
  defines $F \equiv (\lambda i.\ (\lambda a.\ f\ -`\ a \cap \mathit{space}\ M)\ `\ A\ i)$
  assumes *indep-F*: *indep-sets* $F\ I$
  assumes *sets-A*: $\bigwedge i.\ i \in I \Longrightarrow A\ i \subseteq \mathit{sets}\ N$
  shows *prob-space.indep-sets* $(\mathit{distr}\ M\ N\ f)\ A\ I$
proof (*rule prob-space.indep-setsI*)
  show $\bigwedge A'\ J.\ J \neq \{\} \Longrightarrow J \subseteq I \Longrightarrow \mathit{finite}\ J \Longrightarrow \forall j{\in}J.\ A'\ j \in A\ j \Longrightarrow$
    $\mathit{measure}\ (\mathit{distr}\ M\ N\ f)\ (\bigcap\ (A'\ `\ J)) = (\prod j{\in}J.\ \mathit{measure}\ (\mathit{distr}\ M\ N\ f)\ (A'\ j))$
  proof −
    fix $A'\ J$
    assume a:$J \subseteq I$ *finite* J $J \neq \{\}$ $\forall j \in J.\ A'\ j \in A\ j$

    define F' **where** $F' = (\lambda i.\ f\ -`\ A'\ i \cap \mathit{space}\ M)$

    have $\bigcap\ (F'\ `\ J) = f\ -`\ (\bigcap\ (A'\ `\ J)) \cap \mathit{space}\ M$
      unfolding *set-eq-iff* F'*-def* **using** $a(3)$ **by** *simp*
    moreover have $\bigcap\ (A'\ `\ J) \in \mathit{sets}\ N$
      by (*metis a sets-A sets.finite-INT subset-iff*)
    ultimately have b:
      $\mathit{measure}\ (\mathit{distr}\ M\ N\ f)\ (\bigcap\ (A'\ `\ J)) = \mathit{measure}\ M\ (\bigcap\ (F'\ `\ J))$
      by (*metis assms*(*1*) *measure-distr*)

    have $\bigwedge j.\ j \in J \Longrightarrow F'\ j \in F\ j$
      using $a(4)$ F'*-def* *F-def* **by** *blast*
    hence c:$\mathit{measure}\ M\ (\bigcap\ (F'\ `\ J)) = (\prod j{\in}\ J.\ \mathit{measure}\ M\ (F'\ j))$
      by (*metis indep-F indep-setsD a*(*1*,*2*,*3*))

    have $\bigwedge j.\ j \in J \Longrightarrow F'\ j = \ f\ -`\ A'\ j\ \cap \mathit{space}\ M$
      by (*simp add*:F'*-def*)
    moreover have $\bigwedge j.\ j \in J \Longrightarrow A'\ j \in \mathit{sets}\ N$
      using $a(1,4)$ *sets-A* **by** *blast*
    ultimately have d:
      $\bigwedge j.\ j \in J \Longrightarrow \mathit{measure}\ M\ (F'\ j) = \mathit{measure}\ (\mathit{distr}\ M\ N\ f)\ (A'\ j)$
      using *assms*(*1*) *measure-distr* **by** *metis*

    show
      $\mathit{measure}\ (\mathit{distr}\ M\ N\ f)\ (\bigcap\ (A'\ `\ J)) = (\prod j{\in}J.\ \mathit{measure}\ (\mathit{distr}\ M\ N\ f)\ (A'\ j))$
      using $b\ c\ d$ **by** *auto*
  qed
  show *prob-space* $(\mathit{distr}\ M\ N\ f)$ **using** *prob-space-distr assms* **by** *blast*
  show $\bigwedge i.\ i \in I \Longrightarrow A\ i \subseteq \mathit{sets}\ (\mathit{distr}\ M\ N\ f)$ **using** *sets-A sets-distr* **by** *blast*
qed

lemma (**in** *prob-space*) *indep-vars-distr*:

```
  assumes f ∈ measurable M N
  assumes ⋀i. i ∈ I ⟹ X' i ∈ measurable N (M' i)
  assumes indep-vars M' (λi. (X' i) ∘ f) I
  shows prob-space.indep-vars (distr M N f) M' X' I
proof −
  interpret D: prob-space (distr M N f)
    using prob-space-distr[OF assms(1)] by simp

  have a: f ∈ space M → space N using assms(1) by (simp add:measurable-def)

  have D.indep-sets (λi. {X' i −` A ∩ space N |A. A ∈ sets (M' i)}) I
  proof (rule indep-sets-distr[OF assms(1)])
    have ⋀i. i ∈ I ⟹ {(X' i ∘ f) −` A ∩ space M |A. A ∈ sets (M' i)} =
      (λa. f −` a ∩ space M) ` {X' i −` A ∩ space N |A. A ∈ sets (M' i)}
      by (rule set-comp-image-cong, simp add:set-eq-iff, use a in blast)
    thus indep-sets (λi. (λa. f −` a ∩ space M) `
        {X' i −` A ∩ space N |A. A ∈ sets (M' i)}) I
      using assms(3) by (simp add:indep-vars-def2 cong:indep-sets-cong)
  next
    fix i
    assume i ∈ I
    thus {X' i −` A ∩ space N |A. A ∈ sets (M' i)} ⊆ sets N
      using assms(2) measurable-sets by blast
  qed
  thus ?thesis
    using assms by (simp add:D.indep-vars-def2)
qed

lemma range-inter: range ((∩) F) = Pow F
  unfolding image-def by auto
```

The singletons and the empty set form an intersection stable generator of a countable discrete σ-algebra:

```
lemma sigma-sets-singletons-and-empty:
  assumes countable M
  shows sigma-sets M (insert {} ((λk. {k}) ` M)) = Pow M
proof −
  have sigma-sets M ((λk. {k}) ` M) = Pow M
    using assms sigma-sets-singletons by auto
  hence Pow M ⊆ sigma-sets M (insert {} ((λk. {k}) ` M))
    by (metis sigma-sets-subseteq subset-insertI)
  moreover have (insert {} ((λk. {k}) ` M)) ⊆ Pow M by blast
  hence sigma-sets M (insert {} ((λk. {k}) ` M)) ⊆ Pow M
    by (meson sigma-algebra.sigma-sets-subset sigma-algebra-Pow)
  ultimately show ?thesis by force
qed
```

In some of the following theorems, the premise M = *measure-pmf p* is used. This allows stating theorems that hold for pmfs more concisely, for example, instead of *measure-pmf.prob p A* $\leq$ *measure-pmf.prob p B* we can just write M = *measure-pmf p* $\Longrightarrow$ *prob A* $\leq$ *prob B* in the locale *prob-space*.

lemma *prob-space-restrict-space*:
  assumes [*simp*]:M = *measure-pmf p*
  shows *prob-space* (*restrict-space M* (*set-pmf p*))
  by (*rule prob-spaceI*, *auto simp*:*emeasure-restrict-space emeasure-pmf*)

The abbreviation below is used to specify the discrete σ-algebra on *UNIV* as a measure space. It is used in places where the existing definitions, such as *indep-vars*, expect a measure space even though only a *measurable* space is really needed, i.e., in cases where the property is invariant with respect to the actual measure.

hide-const (**open**) *discrete*

abbreviation *discrete* $\equiv$ *count-space UNIV*

lemma (**in** *prob-space*) *indep-vars-restrict-space*:
  assumes [*simp*]:M = *measure-pmf p*
  assumes
    prob-space.indep-vars (*restrict-space M* (*set-pmf p*)) (λ-. *discrete*) *X I*
  shows *indep-vars* (λ-. *discrete*) *X I*
proof −
  have *a*: *id* $\in$ *restrict-space M* (*set-pmf p*) $\rightarrow_M$ *M*
    by (*simp add*:*measurable-def range-inter sets-restrict-space*)

  have *prob-space.indep-vars* (*distr* (*restrict-space M* (*set-pmf p*)) *M id*) (λ-. *discrete*) *X I*
   using *assms a prob-space-restrict-space* **by** (*auto intro*!:*prob-space.indep-vars-distr*)
  moreover have
    $\bigwedge$*A*. *emeasure* (*distr* (*restrict-space M* (*set-pmf p*)) *M id*) *A* = *emeasure M A*
    using *emeasure-distr*[*OF a*]
    by (*auto simp add*: *emeasure-restrict-space emeasure-Int-set-pmf*)
  hence *distr* (*restrict-space M p*) *M id* = *M*
    by (*auto intro*: *measure-eqI*)
  ultimately show *?thesis* **by** *simp*
qed

lemma (**in** *prob-space*) *measure-pmf-eq*:
  assumes M = *measure-pmf p*
  assumes $\bigwedge$*x*. *x* $\in$ *set-pmf p* $\Longrightarrow$ ($x \in P$) = ($x \in Q$)
  shows *prob P* = *prob Q*
  unfolding *assms*(*1*)
  by (*rule measure-eq-AE*, *rule AE-pmfI*[*OF assms*(*2*)], *auto*)

The following lemma is an intro rule for the independence of random variables defined on pmfs. In that case it is possible, to check the independence of random variables point-wise.

The proof relies on the fact that the support of a pmf is countable and the

σ-algebra of such a set can be generated by singletons.

lemma (**in** *prob-space*) *indep-vars-pmf*:
  assumes [*simp*]:M = *measure-pmf* p
  assumes $\bigwedge a\ J.\ J \subseteq I \Longrightarrow$ *finite* $J \Longrightarrow$
    prob $\{\omega.\ \forall i \in J.\ X\ i\ \omega = a\ i\} = (\prod i \in J.$ *prob* $\{\omega.\ X\ i\ \omega = a\ i\})$
  shows *indep-vars* (λ-. *discrete*) $X\ I$
proof –
  interpret R:*prob-space* (*restrict-space* M (*set-pmf* p))
    using *prob-space-restrict-space* **by** *auto*

  have *events-eq-pow*: R.*events* = *Pow* (*set-pmf* p)
    by (*simp add*:*sets-restrict-space range-inter*)

  define G **where** $G = (\lambda i.\ \{\{\}\} \cup (\lambda x.\ \{x\})$ ‘ ($X\ i$ ‘ *set-pmf* p))
  define F **where** $F = (\lambda i.\ \{X\ i$ –‘ $a \cap$ *set-pmf* $p|a.\ a \in G\ i\})$

  have *sigma-sets-pow*:
    $\bigwedge i.\ i \in I \Longrightarrow$ *sigma-sets* ($X\ i$ ‘ *set-pmf* p) ($G\ i$) = *Pow* ($X\ i$ ‘ *set-pmf* p)
   by (*simp add*:*G-def*, *metis countable-image countable-set-pmf sigma-sets-singletons-and-empty*)

  have *F-in-events*: $\bigwedge i.\ i \in I \Longrightarrow F\ i \subseteq$ *Pow* (*set-pmf* p)
    unfolding *F-def* **by** *blast*

  have *as-sigma-sets*:
    $\bigwedge i.\ i \in I \Longrightarrow \{u.\ \exists A.\ u = X\ i$ –‘ $A \cap$ *set-pmf* $p\}$ = *sigma-sets* (*set-pmf* p) (F
i)
  proof –
    fix i
    assume a:$i \in I$
    have $\bigwedge A.\ X\ i$ –‘ $A \cap$ *set-pmf* $p = X\ i$ –‘ ($A \cap X\ i$ ‘ *set-pmf* p) $\cap$ *set-pmf* p
      by *auto*
    hence $\{u.\ \exists A.\ u = X\ i$ –‘ $A \cap$ *set-pmf* $p\}$ =
        $\{X\ i$ –‘ $A \cap$ *set-pmf* $p\ |A.\ A \subseteq X\ i$ ‘ *set-pmf* $p\}$
      by (*metis* (*no-types*, *opaque-lifting*) *inf-le2*)
    also have
      ... = $\{X\ i$ –‘ $A \cap$ *set-pmf* $p\ |A.\ A \in$ *sigma-sets* ($X\ i$ ‘ *set-pmf* p) ($G\ i$)$\}$
      using a **by** (*simp add*:*sigma-sets-pow*)
    also have ... = *sigma-sets* (*set-pmf* p) $\{X\ i$ –‘ $a \cap$ *set-pmf* $p\ |a.\ a \in G\ i\}$
      by (*subst sigma-sets-vimage-commute*[*symmetric*], *auto*)
    also have ... = *sigma-sets* (*set-pmf* p) ($F\ i$)
      by (*simp add*:*F-def*)
    finally show
      $\{u.\ \exists A.\ u = X\ i$ –‘ $A \cap$ *set-pmf* $p\}$ = *sigma-sets* (*set-pmf* p) ($F\ i$)
      by *simp*
  qed

  have *F-Int-stable*: $\bigwedge i.\ i \in I \Longrightarrow$ *Int-stable* ($F\ i$)
  proof (*rule Int-stableI*)
    fix $i\ a\ b$

```
  assume i ∈ I  a ∈ F i  b ∈ F i
  thus a ∩ b ∈ (F i)
    unfolding F-def G-def by (cases a ∩ b = {}, auto)
qed

have F-indep-sets:R.indep-sets F I
proof (rule R.indep-setsI)
  fix i
  assume i ∈ I
  show F i ⊆ R.events
    unfolding F-def events-eq-pow by blast
next
  fix A
  fix J
  assume a:J ⊆ I J ≠ {} finite J ∀ j∈J. A j ∈ F j
  have b: ⋀j. j ∈ J ⟹ A j ⊆ set-pmf p
    by (metis PowD a(1,4) subsetD F-in-events)
  obtain x where x-def:⋀j. j ∈ J  ⟹ A j = X j −‘ x j ∩ set-pmf p ∧ x j ∈ G j
    using a by (simp add:Pi-def F-def, metis)

  show R.prob (⋂ (A ‘ J)) = (∏j∈J. R.prob (A j))
  proof (cases ∃ j ∈ J. A j = {})
    case True
    hence ⋂ (A ‘ J) = {} by blast
    then show ?thesis
      using a True by (simp, metis measure-empty)
  next
    case False
    then have ⋀j. j ∈ J ⟹ x j ≠ {} using x-def by auto
    hence ⋀j. j ∈ J ⟹ x j ∈ (λx. {x}) ‘ X j ‘ set-pmf p
      using x-def by (simp add:G-def)
    then obtain y where y-def: ⋀j. j ∈ J ⟹ x j = {y j}
      by (simp add:image-def, metis)

    have ⋂ (A ‘ J) ⊆ set-pmf p using b a(2) by blast
    hence R.prob (⋂ (A ‘ J)) = prob (⋂ j ∈ J. A j)
      by (simp add: measure-restrict-space)
    also have ... = prob ({ω. ∀ j ∈ J. X j ω = y j})
      using a x-def y-def apply (simp add:vimage-def measure-Int-set-pmf)
      by (rule arg-cong2 [where f=measure], auto)
    also have ... = (∏ j∈ J. prob (A j))
      using x-def y-def a assms(2)
      by (simp add:vimage-def measure-Int-set-pmf)
    also have ... = (∏j∈J. R.prob (A j))
      using b by (simp add: measure-restrict-space cong:prod.cong)
    finally show ?thesis by blast
  qed
qed
```

have *R.indep-sets* ($\lambda i.$ *sigma-sets* (*set-pmf p*) (F i)) I
using *R.indep-sets-sigma*[*simplified*] *F-Int-stable F-indep-sets*
by (*auto simp*:*space-restrict-space*)

hence *R.indep-sets* ($\lambda i.$ $\{u.\ \exists A.\ u = X\ i\ -`\ A \cap$ *set-pmf p*$\}$) I
by (*simp add*: *as-sigma-sets cong*:*R.indep-sets-cong*)

hence *R.indep-vars* (λ-. *discrete*) X I
unfolding *R.indep-vars-def2*
by (*simp add*:*measurable-def sets-restrict-space range-inter*)

thus *?thesis*
using *indep-vars-restrict-space*[*OF assms*(*1*)] **by** *simp*
qed

lemma (**in** *prob-space*) *split-indep-events*:
assumes M = *measure-pmf p*
assumes *indep-vars* ($\lambda i.$ *discrete*) X' I
assumes $K \subseteq I$ *finite* K
shows *prob* $\{\omega.\ \forall x \in K.\ P\ x\ (X'\ x\ \omega)\} = (\prod x \in K.$ *prob* $\{\omega.\ P\ x\ (X'\ x\ \omega)\})$
proof −
have [*simp*]: *space M* = *UNIV* *events* = *UNIV* *prob UNIV* = *1*
by (*simp add*:*assms*(*1*))+

have *indep-vars* (λ-. *discrete*) X' K
using *assms*(*2*,*3*) *indep-vars-subset* **by** *blast*
hence *indep-events* ($\lambda x.$ $\{\omega \in$ *space M*. $P\ x\ (X'\ x\ \omega)\}$) K
using *indep-eventsI-indep-vars* **by** *force*
hence *a*:*indep-events* ($\lambda x.$ $\{\omega.\ P\ x\ (X'\ x\ \omega)\}$) K
by *simp*

have *prob* $\{\omega.\ \forall x \in K.\ P\ x\ (X'\ x\ \omega)\}$ = *prob* $(\bigcap x \in K.\ \{\omega.\ P\ x\ (X'\ x\ \omega)\})$
by (*simp add*: *measure-pmf-eq*[*OF assms*(*1*)])
also have ... = $(\prod x \in K.$ *prob* $\{\omega.\ P\ x\ (X'\ x\ \omega)\})$
using *a assms*(*4*) **by** (*cases* $K = \{\}$, *auto simp*: *indep-events-def*)
finally show *?thesis* **by** *simp*
qed

lemma *pmf-of-set-eq-uniform*:
assumes *finite A* $A \neq \{\}$
shows *measure-pmf* (*pmf-of-set A*) = *uniform-measure discrete A*
proof −
have *a*:*real* (*card A*) > *0* **using** *assms*
by (*simp add*: *card-gt-0-iff*)

have *b*:
$\bigwedge Y.$ *emeasure* (*pmf-of-set A*) Y = *emeasure* (*uniform-measure discrete A*) Y
using *assms a*
by (*simp add*: *emeasure-pmf-of-set divide-ennreal ennreal-of-nat-eq-real-of-nat*)

```
  show ?thesis
    by (rule measure-eqI, auto simp add: b)
qed

lemma (in prob-space) uniform-onI:
  assumes M = measure-pmf p
  assumes finite A A ≠ {}
  assumes ⋀a. prob {ω. X ω = a} = indicator A a / card A
  shows uniform-on X A
proof -
  have a:⋀a. measure-pmf.prob p {x. X x = a} = indicator A a / card A
    using assms(1,4) by simp

  have b:map-pmf X p = pmf-of-set A
    by (rule pmf-eqI, simp add:assms pmf-map vimage-def a)

  have distr M discrete X = map-pmf X p
    by (simp add: map-pmf-rep-eq assms(1))
  also have ... = measure-pmf (pmf-of-set A)
    using b by simp
  also have ... = uniform-measure discrete A
    by (rule pmf-of-set-eq-uniform[OF assms(2,3)])
  finally have distr M discrete X = uniform-measure discrete A
    by simp
  moreover have random-variable discrete X
    by (simp add: assms(1))
  ultimately show ?thesis using assms(2,3)
    by (simp add: uniform-on-def)
qed

end
```

3 Carter-Wegman Hash Family

```
theory Carter-Wegman-Hash-Family
  imports
    Interpolation-Polynomials-HOL-Algebra.Interpolation-Polynomial-Cardinalities
    Universal-Hash-Families-More-Independent-Families
begin
```

The Carter-Wegman hash family is a generic method to obtain k-universal hash families for arbitrary k. (There are faster solutions, such as tabulation hashing, which are limited to a specific k. See for example [2].)

The construction was described by Wegman and Carter [4], it is a hash family between the elements of a finite field and works by choosing randomly a polynomial over the field with degree less than k. The hash function is the evaluation of a such a polynomial.

Using the property that the fraction of polynomials interpolating a given set of $s \leq k$ points is *1 / real (card (carrier R))*^{s}, which is shown in [1], it is possible to obtain both that the hash functions are k-wise independent and uniformly distributed.

In the following two locales are introduced, the main reason for both is to make the statements of the theorems and proofs more concise. The first locale *poly-hash-family* fixes a finite ring R and the probability space of the polynomials of degree less than k. Because the ring is not a field, the family is not yet k-universal, but it is still possible to state a few results such as the fact that the range of the hash function is a subset of the carrier of the ring.

The second locale *carter-wegman-hash-family* is an extension of the former with the assumption that R is a field with which the k-universality follows.

The reason for using two separate locales is to support use cases, where the ring is only probably a field. For example if it is the set of integers modulo an approximate prime, in such a situation a subset of the properties of an algorithm using approximate primes would need to be verified even if R is only a ring.

definition (**in** *ring*) *hash x ω = eval ω x*

locale *poly-hash-family = ring +*
  fixes *k :: nat*
  assumes *finite-carrier*[*simp*]: *finite* (*carrier R*)
  assumes *k-ge-0*: *k > 0*
begin

definition *space* **where** *space = bounded-degree-polynomials R k*
definition *M* **where** *M = measure-pmf* (*pmf-of-set space*)

lemma *finite-space*[*simp*]:*finite space*
  unfolding *space-def* **using** *fin-degree-bounded finite-carrier* **by** *simp*

lemma *non-empty-bounded-degree-polynomials*[*simp*]:*space ≠ {}*
  unfolding *space-def* **using** *non-empty-bounded-degree-polynomials* **by** *simp*

This is to add *carrier-not-empty* to the simp set in the context of *poly-hash-family*:

lemma *non-empty-carrier*[*simp*]: *carrier R ≠ {}*
  by (*simp add*:*carrier-not-empty*)

sublocale *prob-space M*
  by (*simp add*:*M-def prob-space-measure-pmf*)

lemma *hash-range*[*simp*]:
  assumes *ω ∈ space*
  assumes *x ∈ carrier R*
  shows *hash x ω ∈ carrier R*

using *assms* **unfolding** *hash-def space-def bounded-degree-polynomials-def*
by (*simp*, *metis eval-in-carrier polynomial-incl univ-poly-carrier*)

lemma *hash-range-2*:
assumes $\omega \in$ *space*
shows (λx. *hash x* ω) ‘ *carrier R* $\subseteq$ *carrier R*
using *hash-range assms* **by** *auto*

lemma *integrable-M*[*simp*]:
fixes *f* :: *'a list* $\Rightarrow$ *'c*::{*banach*, *second-countable-topology*}
shows *integrable M f*
unfolding *M-def*
by (*rule integrable-measure-pmf-finite*, *simp*)

end

locale *carter-wegman-hash-family* = *poly-hash-family* +
assumes *field-R*: *field R*
begin
sublocale *field*
using *field-R* **by** *simp*

abbreviation *field-size* $\equiv$ *card* (*carrier R*)

lemma *poly-cards*:
assumes *K* $\subseteq$ *carrier R*
assumes *card K* $\leq$ *k*
assumes *y* ‘ *K* $\subseteq$ (*carrier R*)
shows
card {$\omega \in$ *space*. ($\forall k \in K$. *eval* ω *k* = *y k*)} = *field-size*^(*k*−*card K*)
unfolding *space-def*
using *interpolating-polynomials-card*[**where** *n*=*k*−*card K* **and** *K*=*K*] *assms*
using *finite-carrier finite-subset* **by** *fastforce*

lemma *poly-cards-single*:
assumes *x* $\in$ *carrier R*
assumes *y* $\in$ *carrier R*
shows *card* {$\omega \in$ *space*. *eval* ω *x* = *y*} = *field-size*^(*k*−*1*)
using *poly-cards*[**where** *K*={*x*} **and** *y*=λ-. *y*, *simplified*] *assms k-ge-0* **by** *simp*

lemma *hash-prob*:
assumes *K* $\subseteq$ *carrier R*
assumes *card K* $\leq$ *k*
assumes *y* ‘ *K* $\subseteq$ *carrier R*
shows
prob {ω. ($\forall x \in K$. *hash x* ω = *y x*)} = *1*/(*real field-size*)^*card K*
proof −
have **0** $\in$ *carrier R* **by** *simp*

hence *a*:*field-size* > *0*
using *finite-carrier card-gt-0-iff* **by** *blast*

have *b*:*real* (*card* {ω ∈ *space*. ∀ *x*∈*K*. *eval* ω *x* = *y x*}) / *real* (*card space*) =
1 / *real field-size* ^ *card K*
using *a assms*(*2*)
apply (*simp add*: *frac-eq-eq poly-cards*[*OF assms*(*1*,*2*,*3*)] *power-add*[*symmetric*])
by (*simp add*:*space-def bounded-degree-polynomials-card*)

show *?thesis*
unfolding *M-def*
by (*simp add*:*hash-def measure-pmf-of-set Int-def b*)
qed

lemma *prob-single*:
assumes *x* ∈ *carrier R y* ∈ *carrier R*
shows *prob* {ω. *hash x* ω = *y*} = *1*/(*real field-size*)
using *hash-prob*[**where** *K*={*x*}] *assms finite-carrier k-ge-0* **by** *simp*

lemma *prob-range*:
assumes [*simp*]:*x* ∈ *carrier R*
shows *prob* {ω. *hash x* ω ∈ *A*} = *card* (*A* ∩ *carrier R*) / *field-size*
proof −
have *prob* {ω. *hash x* ω ∈ *A*} = *prob* ($\bigcup$ *a* ∈ *A* ∩ *carrier R*. {ω. *hash x* ω = *a*})
by (*rule measure-pmf-eq*, *auto simp*:*M-def*)
also have ... = ($\sum$ *a* ∈ (*A* ∩ *carrier R*). *prob* {ω. *hash x* ω = *a*})
by (*rule measure-finite-Union*, *auto simp*:*M-def disjoint-family-on-def*)
also have ... = ($\sum$ *a* ∈ (*A* ∩ *carrier R*). *1*/(*real field-size*))
by (*rule sum.cong*, *auto simp*:*prob-single*)
also have ... = *card* (*A* ∩ *carrier R*) / *field-size*
by *simp*
finally show *?thesis* **by** *simp*
qed

lemma *indep*:
assumes *J* ⊆ *carrier R*
assumes *card J* ≤ *k*
shows *indep-vars* (λ-. *discrete*) *hash J*
proof −
have 0 ∈ *carrier R* **by** *simp*
hence *card-R-ge-0*:*field-size* > *0*
using *card-gt-0-iff finite-carrier* **by** *blast*

have *fin-J*: *finite J*
using *finite-carrier assms*(*1*) *finite-subset* **by** *blast*

show *?thesis*
proof (*rule indep-vars-pmf*[*OF M-def*])
fix *a*

```
fix J'
assume a: J' ⊆ J finite J'
have card-J': card J' ≤ k
  by (metis card-mono order-trans a(1) assms(2) fin-J)
have J'-in-carr: J' ⊆ carrier R by (metis order-trans a(1) assms(1))

show prob {ω. ∀x∈J'. hash x ω = a x} = (∏x∈J'. prob {ω. hash x ω = a x})
proof (cases a ` J' ⊆ carrier R)
  case True
  have a-carr: ⋀x. x ∈ J' ⟹ a x ∈ carrier R  using True by force
  have prob {ω. ∀x∈J'. hash x ω = a x} =
    real (card {ω ∈ space. ∀x∈J'. eval ω x = a x}) / real (card space)
    by (simp add:M-def measure-pmf-of-set Int-def hash-def)
  also have ... = real (field-size ^ (k − card J')) / real (card space)
    using True by (simp add: poly-cards[OF J'-in-carr card-J'])
  also have
    ... = real field-size ^ (k − card J') / real field-size ^ k
    by (simp add:space-def bounded-degree-polynomials-card)
  also have
    ... = real field-size ^ ((k − 1) * card J') / real field-size ^ (k * card J')
    using card-J' by (simp add:power-add[symmetric] power-mult[symmetric]
        diff-mult-distrib frac-eq-eq add.commute)
  also have
    ... = (real field-size ^ (k − 1)) ^ card J' / (real field-size ^ k) ^ card J'
    by (simp add:power-add power-mult)
  also have
    ... = (∏x∈J'. real (card {ω ∈ space. eval ω x = a x}) / real (card space))
    using a-carr poly-cards-single[OF subsetD[OF J'-in-carr]]
    by (simp add:space-def bounded-degree-polynomials-card power-divide)
  also have ... = (∏x∈J'. prob {ω. hash x ω = a x})
    by (simp add:measure-pmf-of-set M-def Int-def hash-def)
  finally show ?thesis by simp
next
  case False
  then obtain j where j-def: j ∈ J' a j ∉ carrier R by blast
  have {ω ∈ space. hash j ω = a j} ⊆ {ω ∈ space. hash j ω ∉ carrier R}
    by (rule subsetI, simp add:j-def)
  also have ... ⊆ {} using j-def(1) J'-in-carr hash-range by blast
  finally have b:{ω ∈ space. hash j ω = a j} = {} by simp
  hence real (card ({ω ∈ space. hash j ω = a j})) = 0 by simp
  hence (∏x∈J'. real (card {ω ∈ space. hash x ω = a x})) = 0
    using a(2) prod-zero[OF a(2)] j-def(1) by auto
  moreover have
    {ω ∈ space. ∀x∈J'. hash x ω = a x} ⊆ {ω ∈ space. hash j ω = a j}
    using j-def by blast
  hence {ω ∈ space. ∀x∈J'. hash x ω = a x} = {} using b by blast
  ultimately show ?thesis
    by (simp add:measure-pmf-of-set M-def Int-def prod-dividef)
qed
```

```
  qed
qed

lemma k-wise-indep:
  k-wise-indep-vars k (λ-. discrete) hash (carrier R)
  unfolding k-wise-indep-vars-def using indep by simp

lemma inj-if-degree-1:
  assumes ω ∈ space
  assumes degree ω = 1
  shows inj-on (λx. hash x ω) (carrier R)
  using assms eval-inj-if-degree-1
  by (simp add:M-def space-def bounded-degree-polynomials-def hash-def)

lemma uniform:
  assumes i ∈ carrier R
  shows uniform-on (hash i) (carrier R)
proof −
  have a:
    ⋀a. prob {ω. hash i ω ∈ {a}} = indicat-real (carrier R) a / real field-size
    by (subst prob-range[OF assms], simp add:indicator-def)
  show ?thesis
    by (rule uniform-onI, use a M-def in auto)
qed
```

This the main result of this section - the Carter-Wegman hash family is k-universal.

```
theorem k-universal:
  k-universal k hash (carrier R) (carrier R)
  using uniform k-wise-indep by (simp add:k-universal-def)

end

lemma poly-hash-familyI:
  assumes ring R
  assumes finite (carrier R)
  assumes 0 < k
  shows poly-hash-family R k
  using assms
  by (simp add:poly-hash-family-def poly-hash-family-axioms-def)

lemma carter-wegman-hash-familyI:
  assumes field F
  assumes finite (carrier F)
  assumes 0 < k
  shows carter-wegman-hash-family F k
  using assms field.is-ring[OF assms(1)] poly-hash-familyI
 by (simp add:carter-wegman-hash-family-def carter-wegman-hash-family-axioms-def)
```

lemma *hash-k-wise-indep*:
  assumes *field F* ∧ *finite* (*carrier F*)
  assumes *1* ≤ *n*
  shows
    prob-space.k-wise-indep-vars (*pmf-of-set* (*bounded-degree-polynomials F n*)) *n*
    (λ-. *pmf-of-set* (*carrier F*)) (*ring.hash F*) (*carrier F*)
proof −
  interpret *carter-wegman-hash-family F n*
    using *assms carter-wegman-hash-familyI* **by** *force*
  have *k-wise-indep-vars n* (λ-. *pmf-of-set* (*carrier F*)) *hash* (*carrier F*)
    by (*rule k-wise-indep-vars-compose*[*OF k-wise-indep*], *simp*)
  thus *?thesis*
    by (*simp add*:*M-def space-def*)
qed

lemma *hash-prob-single*:
  assumes *field F* ∧ *finite* (*carrier F*)
  assumes *x* ∈ *carrier F*
  assumes *1* ≤ *n*
  assumes *y* ∈ *carrier F*
  shows
    $\mathcal{P}$(ω *in pmf-of-set* (*bounded-degree-polynomials F n*). *ring.hash F x* ω = *y*)
      = *1*/(*real* (*card* (*carrier F*)))
proof −
  interpret *carter-wegman-hash-family F n*
    using *assms carter-wegman-hash-familyI* **by** *force*
  show *?thesis*
    using *prob-single*[*OF assms*(*2*,*4*)] **by** (*simp add*:*M-def space-def*)
qed

end

4 Indexed Products of Probability Mass Functions

theory *Universal-Hash-Families-More-Product-PMF*
  imports
    Concentration-Inequalities.Concentration-Inequalities-Preliminary
    Finite-Fields.Finite-Fields-More-Bijections
    Universal-Hash-Families-More-Independent-Families
begin

hide-const (**open**) *Isolated.discrete*

This section introduces a restricted version of *Pi-pmf* where the default value is *undefined* and contains some additional results about that case in addition to *HOL−Probability.Product-PMF*

abbreviation *prod-pmf* **where** *prod-pmf I M* ≡ *Pi-pmf I undefined M*

lemma *measure-pmf-cong*:
  assumes $\bigwedge x.\ x \in \textit{set-pmf}\ p \Longrightarrow x \in P \longleftrightarrow x \in Q$
  shows *measure* (*measure-pmf p*) P = *measure* (*measure-pmf p*) Q
  using *assms*
  by (*intro finite-measure.finite-measure-eq-AE AE-pmfI*) *auto*

lemma *pmf-mono*:
  assumes $\bigwedge x.\ x \in \textit{set-pmf}\ p \Longrightarrow x \in P \Longrightarrow x \in Q$
  shows *measure* (*measure-pmf p*) $P \le$ *measure* (*measure-pmf p*) Q
proof −
  have *measure* (*measure-pmf p*) P = *measure* (*measure-pmf p*) ($P \cap$ (*set-pmf p*))
    by (*intro measure-pmf-cong*) *auto*
  also have ... $\le$ *measure* (*measure-pmf p*) Q
    using *assms* **by** (*intro finite-measure.finite-measure-mono*) *auto*
  finally show *?thesis* **by** *simp*
qed

lemma *pmf-add*:
  assumes $\bigwedge x.\ x \in P \Longrightarrow x \in \textit{set-pmf}\ p \Longrightarrow x \in Q \lor x \in R$
  shows *measure p* $P \le$ *measure p* Q + *measure p* R
proof −
  have *measure p* $P \le$ *measure p* ($Q \cup R$)
    using *assms* **by** (*intro pmf-mono*) *blast*
  also have ... $\le$ *measure p* Q + *measure p* R
    by (*rule measure-subadditive*, *auto*)
  finally show *?thesis* **by** *simp*
qed

lemma *pmf-prod-pmf*:
  assumes *finite I*
  shows *pmf* (*prod-pmf I M*) x = (*if* $x \in$ *extensional I then* $\prod i \in I.$ (*pmf* (*M i*)) (*x i*) *else 0*)
  by (*simp add*: *pmf-Pi*[*OF assms*(*1*)] *extensional-def*)

lemma *PiE-defaut-undefined-eq*: *PiE-dflt I undefined M* = *PiE I M*
  by (*simp add*:*PiE-dflt-def PiE-def extensional-def Pi-def set-eq-iff*) *blast*

lemma *set-prod-pmf*:
  assumes *finite I*
  shows *set-pmf* (*prod-pmf I M*) = *PiE I* (*set-pmf* $\circ$ *M*)
  by (*simp add*:*set-Pi-pmf*[*OF assms*] *PiE-defaut-undefined-eq*)

A more general version of *measure-Pi-pmf-Pi*.

lemma *prob-prod-pmf'*:
  assumes *finite I*
  assumes $J \subseteq I$
  shows *measure* (*measure-pmf* (*Pi-pmf I d M*)) (*Pi J A*) = ($\prod i \in J.$ *measure* (*M i*) (*A i*))
proof −

have *a*:*Pi J A* = *Pi I* (*λi. if i* ∈ *J then A i else UNIV*)
using *assms* **by** (*simp add*:*Pi-def set-eq-iff*, *blast*)
show *?thesis*
using *assms*
by (*simp add*:*if-distrib a measure-Pi-pmf-Pi*[*OF assms*(*1*)] *prod.If-cases*[*OF assms*(*1*)] *Int-absorb1*)
qed

lemma *prob-prod-pmf-slice*:
assumes *finite I*
assumes *i* ∈ *I*
shows *measure* (*measure-pmf* (*prod-pmf I M*)) {*ω*. *P* (*ω i*)} = *measure* (*M i*) {*ω*. *P ω*}
using *prob-prod-pmf'*[*OF assms*(*1*), **where** *J*={*i*} **and** *M*=*M* **and** *A*=*λ-*. *Collect P*]
by (*simp add*:*assms Pi-def*)

definition *restrict-dfl* **where** *restrict-dfl f A d* = (*λx. if x* ∈ *A then f x else d*)

lemma *pi-pmf-decompose*:
assumes *finite I*
shows *Pi-pmf I d M* = *map-pmf* (*λω. restrict-dfl* (*λi. ω* (*f i*) *i*) *I d*) (*Pi-pmf* (*f* ‘ *I*) (*λ-*. *d*) (*λj. Pi-pmf* (*f* −‘ {*j*} ∩ *I*) *d M*))
proof −
have *fin-F-I*:*finite* (*f* ‘ *I*) **using** *assms* **by** *blast*

have *finite I* ⟹ *?thesis*
using *fin-F-I*
proof (*induction f* ‘ *I arbitrary*: *I rule*:*finite-induct*)
case *empty*
then show *?case* **by** (*simp add*:*restrict-dfl-def*)
next
case (*insert x F*)
have *a*: (*f* −‘ {*x*} ∩ *I*) ∪ (*f* −‘ *F* ∩ *I*) = *I*
using *insert*(*4*) **by** *blast*
have *b*: *F* = *f* ‘ (*f* −‘ *F* ∩ *I*) **using** *insert*(*2*,*4*)
by (*auto simp add*:*set-eq-iff image-def vimage-def*)
have *c*: *finite* (*f* −‘ *F* ∩ *I*) **using** *insert* **by** *blast*
have *d*: ⋀*j. j* ∈ *F* ⟹ (*f* −‘ {*j*} ∩ (*f* −‘ *F* ∩ *I*)) = (*f* −‘ {*j*} ∩ *I*)
using *insert*(*4*) **by** *blast*

have *Pi-pmf I d M* = *Pi-pmf* ((*f* −‘ {*x*} ∩ *I*) ∪ (*f* −‘ *F* ∩ *I*)) *d M*
by (*simp add*:*a*)
also have ... = *map-pmf* (*λ*(*g*, *h*) *i. if i* ∈ *f* −‘ {*x*} ∩ *I then g i else h i*)
(*pair-pmf* (*Pi-pmf* (*f* −‘ {*x*} ∩ *I*) *d M*) (*Pi-pmf* (*f* −‘ *F* ∩ *I*) *d M*))
using *insert* **by** (*subst Pi-pmf-union*) *auto*
also have ... = *map-pmf* (*λ*(*g*,*h*) *i. if f i* = *x* ∧ *i* ∈ *I then g i else if f i* ∈ *F* ∧ *i* ∈ *I then h* (*f i*) *i else d*)
(*pair-pmf* (*Pi-pmf* (*f* −‘ {*x*} ∩ *I*) *d M*) (*Pi-pmf F* (*λ-*. *d*) (*λj. Pi-pmf* (*f* −‘

$\{j\} \cap (f -` F \cap I))\ d\ M)))$
by (*simp add*:*insert*(*3*)[*OF b c*] *map-pmf-comp case-prod-beta′ apsnd-def map-prod-def*
pair-map-pmf2 b[*symmetric*] *restrict-dfl-def*) (*metis fst-conv snd-conv*)
also have ... = *map-pmf* (λ(*g*,*h*) *i*. *if i* ∈ *I then* (*h*(*x* := *g*)) (*f i*) *i else d*)
(*pair-pmf* (*Pi-pmf* (*f* −‘ {*x*} ∩ *I*) *d M*) (*Pi-pmf F* (λ-. *d*) (λ*j*. *Pi-pmf* (*f* −‘ {*j*} ∩ *I*) *d M*)))
using *insert*(*4*) *d*
by (*intro arg-cong2*[**where** *f*=*map-pmf*] *ext*) (*auto simp add*:*case-prod-beta′ cong*:*Pi-pmf-cong*)
also have ... = *map-pmf* (λω *i*. *if i* ∈ *I then* ω (*f i*) *i else d*) (*Pi-pmf* (*insert x F*) (λ-. *d*) (λ*j*. *Pi-pmf* (*f* −‘ {*j*} ∩ *I*) *d M*))
by (*simp add*:*Pi-pmf-insert*[*OF insert*(*1*,*2*)] *map-pmf-comp case-prod-beta′*)
finally show *?case* **by** (*simp add*:*insert*(*4*) *restrict-dfl-def*)
qed
thus *?thesis* **using** *assms* **by** *blast*
qed

lemma *restrict-dfl-iter*: *restrict-dfl* (*restrict-dfl f I d*) *J d* = *restrict-dfl f* (*I* ∩ *J*) *d*
by (*rule ext*, *simp add*:*restrict-dfl-def*)

lemma *indep-vars-restrict′*:
assumes *finite I*
shows *prob-space.indep-vars* (*Pi-pmf I d M*) (λ-. *discrete*) (λ*i* ω. *restrict-dfl* ω (*f* −‘ {*i*} ∩ *I*) *d*) (*f* ‘ *I*)
proof −
let *?Q* = (*Pi-pmf* (*f* ‘ *I*) (λ-. *d*) (λ*i*. *Pi-pmf* (*I* ∩ *f* −‘ {*i*}) *d M*))
have *a*:*prob-space.indep-vars ?Q* (λ-. *discrete*) (λ*i* ω. ω *i*) (*f* ‘ *I*)
using *assms* **by** (*intro indep-vars-Pi-pmf*, *blast*)
have *b*: *AE x in measure-pmf ?Q*. ∀ *i*∈*f* ‘ *I*. *x i* = *restrict-dfl* (λ*i*. *x* (*f i*) *i*) (*I* ∩ *f* −‘ {*i*}) *d*
using *assms*
by (*auto simp add*:*PiE-dflt-def restrict-dfl-def AE-measure-pmf-iff set-Pi-pmf comp-def Int-commute*)
have *prob-space.indep-vars ?Q* (λ-. *discrete*) (λ*i x*. *restrict-dfl* (λ*i*. *x* (*f i*) *i*) (*I* ∩ *f* −‘ {*i*}) *d*) (*f* ‘ *I*)
by (*rule prob-space.indep-vars-cong-AE*[*OF prob-space-measure-pmf b a*], *simp*)
thus *?thesis*
using *prob-space-measure-pmf*
by (*auto intro*!:*prob-space.indep-vars-distr simp*:*pi-pmf-decompose*[*OF assms*, **where** *f*=*f*]
map-pmf-rep-eq comp-def restrict-dfl-iter Int-commute)
qed

lemma *indep-vars-restrict-intro′*:
assumes *finite I*
assumes $\bigwedge i\ \omega.\ i \in J \Longrightarrow X'\ i\ \omega = X'\ i$ (*restrict-dfl* ω (*f* −‘ {*i*} ∩ *I*) *d*)
assumes *J* = *f* ‘ *I*

```
  assumes ⋀ω i. i ∈ J ⟹ X' i ω ∈ space (M' i)
  shows prob-space.indep-vars (measure-pmf (Pi-pmf I d p)) M' (λi ω. X' i ω) J
proof −
  define M where M ≡ measure-pmf (Pi-pmf I d p)
  interpret prob-space M
    using M-def prob-space-measure-pmf by blast
  have indep-vars (λ-. discrete) (λi x. restrict-dfl x (f −` {i} ∩ I) d) (f ` I)
    unfolding M-def by (rule indep-vars-restrict'[OF assms(1)])
  hence indep-vars M' (λi ω. X' i (restrict-dfl ω ( f −` {i} ∩ I) d)) (f ` I)
    using assms(4)
   by (intro indep-vars-compose2[where Y=X' and N=M' and M'=λ-. discrete])
(auto simp:assms(3))
  hence indep-vars M' (λi ω. X' i ω) (f ` I)
    using assms(2)[symmetric]
    by (simp add:assms(3) cong:indep-vars-cong)
  thus ?thesis
    unfolding M-def using assms(3) by simp
qed

lemma
  fixes f :: 'b ⇒ ('c :: {second-countable-topology,banach,real-normed-field})
  assumes finite I
  assumes i ∈ I
  assumes integrable (measure-pmf (M i)) f
  shows  integrable-Pi-pmf-slice: integrable (Pi-pmf I d M) (λx. f (x i))
  and expectation-Pi-pmf-slice: integral^L (Pi-pmf I d M) (λx. f (x i)) = integral^L
(M i) f
proof −
   have a:distr (Pi-pmf I d M) (M i) (λω. ω i) = distr (Pi-pmf I d M) discrete
(λω. ω i)
    by (rule distr-cong, auto)

  have b: measure-pmf.random-variable (M i) borel f
    using assms(3) by simp

  have c:integrable (distr (Pi-pmf I d M) (M i) (λω. ω i)) f
    using assms(1,2,3)
    by (subst a, subst map-pmf-rep-eq[symmetric], subst Pi-pmf-component, auto)

  show integrable (Pi-pmf I d M) (λx. f (x i))
    by (rule integrable-distr[where f=f and M'=M i])  (auto intro: c)

   have integral^L (Pi-pmf I d M) (λx. f (x i)) = integral^L (distr (Pi-pmf I d M)
(M i) (λω. ω i)) f
    using b by (intro integral-distr[symmetric], auto)
  also have ... =  integral^L (map-pmf (λω. ω i) (Pi-pmf I d M)) f
    by (subst a, subst map-pmf-rep-eq[symmetric], simp)
  also have ... =  integral^L (M i) f
    using assms(1,2) by (simp add: Pi-pmf-component)
```

finally show *integral*^{L} (*Pi-pmf I d M*) (λx. *f* (*x i*)) = *integral*^{L} (*M i*) *f* **by** *simp*
qed

This is an improved version of *expectation-prod-Pi-pmf*. It works for general normed fields instead of non-negative real functions .

lemma *expectation-prod-Pi-pmf*:
fixes *f* :: $'a \Rightarrow 'b \Rightarrow ('c$ :: {*second-countable-topology*,*banach*,*real-normed-field*})
assumes *finite I*
assumes $\bigwedge i.\ i \in I \Longrightarrow$ *integrable* (*measure-pmf* (*M i*)) (*f i*)
shows *integral*^{L} (*Pi-pmf I d M*) (λx. ($\prod i \in I$. *f i* (*x i*))) = ($\prod$ $i \in I$. *integral*^{L} (*M i*) (*f i*))
proof –
have *a*: *prob-space.indep-vars* (*measure-pmf* (*Pi-pmf I d M*)) (λ-. *borel*) ($\lambda i\ \omega$. *f i* (ω *i*)) *I*
by (*intro prob-space.indep-vars-compose2*[**where** *Y*=*f* **and** *M'*=λ-. *discrete*]
prob-space-measure-pmf indep-vars-Pi-pmf assms(*1*)) *auto*
have *integral*^{L} (*Pi-pmf I d M*) (λx. ($\prod i \in I$. *f i* (*x i*))) = ($\prod$ $i \in I$. *integral*^{L} (*Pi-pmf I d M*) (λx. *f i* (*x i*)))
by (*intro prob-space.indep-vars-lebesgue-integral prob-space-measure-pmf assms*(*1*,*2*)
a integrable-Pi-pmf-slice) *auto*
also have ... = ($\prod$ $i \in I$. *integral*^{L} (*M i*) (*f i*))
by (*intro prod.cong expectation-Pi-pmf-slice assms*(*1*,*2*)) *auto*
finally show *?thesis* **by** *simp*
qed

lemma *variance-prod-pmf-slice*:
fixes *f* :: $'a \Rightarrow$ *real*
assumes $i \in I$ *finite I*
assumes *integrable* (*measure-pmf* (*M i*)) ($\lambda\omega$. *f* ω^*2*)
shows *prob-space.variance* (*Pi-pmf I d M*) ($\lambda\omega$. *f* (ω *i*)) = *prob-space.variance* (*M i*) *f*
proof –
have *a*:*integrable* (*measure-pmf* (*M i*)) *f*
using *assms*(*3*) *measure-pmf.square-integrable-imp-integrable* **by** *auto*
have *b*: *integrable* (*measure-pmf* (*Pi-pmf I d M*)) (λx. (*f* (*x i*))^2)
by (*rule integrable-Pi-pmf-slice*[*OF assms*(*2*) *assms*(*1*)], *metis assms*(*3*))
have *c*: *integrable* (*measure-pmf* (*Pi-pmf I d M*)) (λx. (*f* (*x i*)))
by (*rule integrable-Pi-pmf-slice*[*OF assms*(*2*) *assms*(*1*)], *metis a*)

have *measure-pmf.expectation* (*Pi-pmf I d M*) (λx. (*f* (*x i*))^2) – (*measure-pmf.expectation* (*Pi-pmf I d M*) (λx. *f* (*x i*)))^2 =
measure-pmf.expectation (*M i*) (λx. (*f x*)^2) – (*measure-pmf.expectation* (*M i*) *f*)^2
using *assms a b c* **by** ((*subst expectation-Pi-pmf-slice*[*OF assms*(*2*,*1*)])*?*, *simp*)+

thus *?thesis*
using *assms a b c* **by** (*simp add*: *measure-pmf.variance-eq*)
qed

lemma *Pi-pmf-bind-return*:
  assumes *finite I*
  shows *Pi-pmf I d* ($\lambda i.$ *M i* $\gg=$ ($\lambda x.$ *return-pmf* (*f i x*))) = *Pi-pmf I d' M* $\gg=$ ($\lambda x.$ *return-pmf* ($\lambda i.$ *if i* $\in$ *I then f i* (*x i*) *else d*))
  using *assms* **by** (*simp add*: *Pi-pmf-bind*[**where** *d'=d'*])

lemma *pmf-of-set-prod-eq*:
  assumes $A \neq \{\}$ *finite A*
  assumes $B \neq \{\}$ *finite B*
  shows *pmf-of-set* ($A \times B$) = *pair-pmf* (*pmf-of-set A*) (*pmf-of-set B*)
proof −
  have *indicat-real* ($A \times B$) (i, j) = *indicat-real A i* $*$ *indicat-real B j* **for** *i j*
    by (*cases i* $\in$ *A*; *cases j* $\in$ *B*) *auto*
  hence *pmf* (*pmf-of-set* ($A \times B$)) (*i*,*j*) = *pmf* (*pair-pmf* (*pmf-of-set A*) (*pmf-of-set B*)) (*i*,*j*)
    for *i j* **using** *assms* **by** (*simp add*:*pmf-pair*)
  thus *?thesis*
    by (*intro pmf-eqI*) *auto*
qed

lemma *split-pmf-mod-div'*:
  assumes *a* > (*0*::*nat*)
  assumes *b* > *0*
  shows *map-pmf* ($\lambda x.$ (*x mod a*, *x div a*)) (*pmf-of-set* {..<*a* $*$ *b*}) = *pmf-of-set* ({..<*a*} $\times$ {..<*b*})
  using *assms* **by** (*intro map-pmf-of-set-bij-betw bij-betw-prod finite-lessThan*)
    (*simp add*: *lessThan-empty-iff*)

lemma *split-pmf-mod-div*:
  assumes *a* > (*0*::*nat*)
  assumes *b* > *0*
  shows *map-pmf* ($\lambda x.$ (*x mod a*, *x div a*)) (*pmf-of-set* {..<*a* $*$ *b*}) =
    pair-pmf (*pmf-of-set* {..<*a*}) (*pmf-of-set* {..<*b*})
  using *assms* **by** (*auto intro*!: *pmf-of-set-prod-eq simp add*:*split-pmf-mod-div'*)

end

5 Pseudorandom Objects

theory *Pseudorandom-Objects*
  imports *Universal-Hash-Families-More-Product-PMF*
begin

This section introduces a combinator library for pseudorandom objects [3]. These can be thought of as PRNGs but with rigorous mathematical properties, which can be used to in algorithms to reduce their randomness usage.

Such an object represents a non-empty multiset, with an efficient mechanism to sample from it. They have a natural interpretation as a probability space (each element is selected with a probability proportional to its occurrence count in the multiset).

The following section will introduce a construction of k-independent hash families as a pseudorandom object. The AFP entry `Expander_Graphs` then follows up with expander walks as pseudorandom objects.

record *'a pseudorandom-object* =
pro-last :: *nat*
pro-select :: *nat* ⇒ *'a*

definition *pro-size* **where** *pro-size S* = *pro-last S* + *1*
definition *sample-pro* **where** *sample-pro S* = *map-pmf* (*pro-select S*) (*pmf-of-set* {*0*..*pro-last S*})

declare [[*coercion sample-pro*]]

abbreviation *pro-set* **where** *pro-set S* ≡ *set-pmf* (*sample-pro S*)

lemma *sample-pro-alt*: *sample-pro S* = *map-pmf* (*pro-select S*) (*pmf-of-set* {..<*pro-size S*})
unfolding *pro-size-def sample-pro-def*
using *Suc-eq-plus1 atLeast0AtMost lessThan-Suc-atMost* **by** *presburger*

lemma *pro-size-gt-0*: *pro-size S* > *0*
unfolding *pro-size-def* **by** *auto*

lemma *set-sample-pro*: *pro-set S* = *pro-select S* ' {..<*pro-size S*}
using *pro-size-gt-0* **unfolding** *sample-pro-alt set-map-pmf*
by (*subst set-pmf-of-set*) *auto*

lemma *set-pmf-of-set-sample-size*[*simp*]:
set-pmf (*pmf-of-set* {..<*pro-size S*}) = {..<*pro-size S*}
using *pro-size-gt-0* **by** (*intro set-pmf-of-set*) *auto*

lemma *pro-select-in-set*: *pro-select S* (*x mod pro-size S*) ∈ *pro-set S*
unfolding *set-sample-pro* **by** (*intro imageI*) (*simp add*:*pro-size-gt-0*)

lemma *finite-pro-set*: *finite* (*pro-set S*)
unfolding *set-sample-pro* **by** (*intro finite-imageI*) *auto*

lemma *integrable-sample-pro*[*simp*]:
fixes *f* :: *'a* ⇒ *'c*::{*banach*, *second-countable-topology*}
shows *integrable* (*measure-pmf* (*sample-pro S*)) *f*
by (*intro integrable-measure-pmf-finite finite-pro-set*)

definition *list-pro* :: *'a list* ⇒ *'a pseudorandom-object* **where**
list-pro ls = (| *pro-last* = *length ls* − *1*, *pro-select* = (!) *ls* |)

lemma *list-pro*:
  assumes $xs \neq []$
  shows *sample-pro* (*list-pro xs*) = *pmf-of-multiset* (*mset xs*) (**is** *?L = ?R*)
proof –
  have *?L = map-pmf* ((!) *xs*) (*pmf-of-set* {..<*length xs*})
    using *assms* **unfolding** *list-pro-def sample-pro-alt pro-size-def* **by** *simp*
  also have ... = *pmf-of-multiset* (*image-mset* ((!) *xs*) (*mset-set* {..<*length xs*}))
    using *assms* **by** (*subst map-pmf-of-set*) *auto*
  also have ... = *?R*
    by (*metis map-nth mset-map mset-set-upto-eq-mset-upto*)
  finally show *?thesis* **by** *simp*
qed

lemma *list-pro-2*:
  assumes $xs \neq []$ *distinct xs*
  shows *sample-pro* (*list-pro xs*) = *pmf-of-set* (*set xs*) (**is** *?L = ?R*)
proof –
  have *?L = map-pmf* ((!) *xs*) (*pmf-of-set* {..<*length xs*})
    using *assms* **unfolding** *list-pro-def sample-pro-alt pro-size-def* **by** *simp*
  also have ... = *pmf-of-set* ((!) *xs* ‘ {..<*length xs*})
    using *assms nth-eq-iff-index-eq* **by** (*intro map-pmf-of-set-inj inj-onI*) *auto*
  also have ... = *?R*
   by (*intro arg-cong*[**where** *f=pmf-of-set*]) (*metis atLeast-upt list.set-map map-nth*)
  finally show *?thesis* **by** *simp*
qed

lemma *list-pro-size*:
  assumes $xs \neq []$
  shows *pro-size* (*list-pro xs*) = *length xs*
  using *assms* **unfolding** *pro-size-def list-pro-def* **by** *auto*

lemma *list-pro-set*:
  assumes $xs \neq []$
  shows *pro-set* (*list-pro xs*) = *set xs*
proof –
  have (!) *xs* ‘ {..<*length xs*} = *set xs* **by** (*metis atLeast-upt list.set-map map-nth*)
 thus *?thesis* **unfolding** *set-sample-pro list-pro-size*[*OF assms*] **by** (*simp add*:*list-pro-def*)
qed

definition *nat-pro* :: *nat* ⇒ *nat pseudorandom-object* **where**
  nat-pro n = (| *pro-last* = *n*−*1*, *pro-select* = *id* |)

lemma *nat-pro-size*:
  assumes *n* > *0*
  shows*pro-size* (*nat-pro n*) = *n*
  using *assms* **unfolding** *nat-pro-def pro-size-def* **by** *auto*

lemma *nat-pro*:
  assumes $n > 0$
  shows *sample-pro* (*nat-pro n*) = *pmf-of-set* {..<*n*}
  unfolding *sample-pro-alt nat-pro-size*[*OF assms*] **by** (*simp add*:*nat-pro-def*)

lemma *nat-pro-set*:
  assumes $n > 0$
  shows *pro-set* (*nat-pro n*) = {..<*n*}
  using *assms* **unfolding** *nat-pro*[*OF assms*] **by** (*simp add*: *lessThan-empty-iff*)

fun *count-zeros* :: *nat* ⇒ *nat* ⇒ *nat* **where**
  count-zeros 0 k = 0 |
  count-zeros (*Suc n*) *k* = (*if odd k then 0 else 1* + *count-zeros n* (*k div 2*))

lemma *count-zeros-iff*: *j* ≤ *n* ⟹ *count-zeros n k* ≥ *j* ⟷ *2^j dvd k*
proof (*induction j arbitrary*: *n k*)
  case *0*
  then show *?case* **by** *simp*
next
  case (*Suc j*)
  then obtain *n'* **where** *n-def*: *n* = *Suc n'* **using** *Suc-le-D* **by** *presburger*
  show *?case* **using** *Suc* **unfolding** *n-def* **by** *auto*
qed

lemma *count-zeros-max*:
  count-zeros n k ≤ *n*
  by (*induction n arbitrary*: *k*) *auto*

definition *geom-pro* :: *nat* ⇒ *nat pseudorandom-object* **where**
  geom-pro n = (| *pro-last* = *2^n* − *1*, *pro-select* = *count-zeros n* |)

lemma *geom-pro-size*: *pro-size* (*geom-pro n*) = *2^n*
  unfolding *geom-pro-def pro-size-def* **by** *simp*

lemma *geom-pro-range*: *pro-set* (*geom-pro n*) ⊆ {..*n*}
  using *count-zeros-max* **unfolding** *sample-pro-alt* **unfolding** *geom-pro-def* **by** *auto*

lemma *geom-pro-prob*:
  measure (*sample-pro* (*geom-pro n*)) {ω. ω ≥ *j*} = *of-bool* (*j* ≤ *n*) / *2^j* (**is** *?L* = *?R*)
proof (*cases j* ≤ *n*)
  case *True*
  have *a*:{..<(*2^n*)::*nat*} ≠ {}
    by (*simp add*: *lessThan-empty-iff*)
  have *b*:*finite* {..<(*2^n*)::*nat*} **by** *simp*

```
    define f :: nat ⇒ nat where f = (λx. x ∗ 2^j)
    have d:inj-on f {..<2^(n−j)} unfolding f-def by (intro inj-onI) simp

    have e:2^j > (0::nat) by simp

    have y ∈ f ‘ {..< 2^(n−j)} ⟷ y ∈ {x. x < 2^n ∧ 2^j dvd x} for y :: nat
    proof −
      have y ∈ f ‘ {..< 2^(n−j)} ⟷ (∃ x. x < 2 ^ (n − j) ∧ y = 2 ^ j ∗ x)
        unfolding f-def by auto
      also have ... ⟷ (∃ x. 2^j ∗ x < 2^j ∗ 2 ^ (n−j) ∧ y = 2 ^ j ∗ x)
        using e by simp
      also have ... ⟷ (∃ x. 2^j ∗ x < 2^n ∧ y = 2 ^ j ∗ x)
        using True by (subst power-add[symmetric]) simp
      also have ... ⟷ (∃ x. y < 2^n ∧ y = x ∗ 2 ^ j)
        by (metis Groups.mult-ac(2))
      also have ... ⟷ y ∈ {x. x < 2^n ∧ 2^j dvd x} by auto
      finally show ?thesis by simp
    qed

    hence c:f ‘ {..< 2^(n−j)} = {x. x < 2^n ∧ 2^j dvd x} by auto

    have ?L = measure (pmf-of-set {..<2^n}) {ω. count-zeros n ω ≥ j}
      unfolding sample-pro-alt geom-pro-size by (simp add:geom-pro-def)
    also have ... = real (card {x::nat. x < 2^n ∧ 2^j dvd x}) / 2^n
      by (simp add: measure-pmf-of-set[OF a b] count-zeros-iff[OF True])
       (simp add:lessThan-def Collect-conj-eq)
    also have ... = real (card (f ‘ {..<2^(n−j)})) / 2^n
      by (simp add:c)
    also have ... = real (card ({..<(2^(n−j)::nat)})) / 2^n
      by (simp add: card-image[OF d])
    also have ... = ?R
      using True by (simp add:frac-eq-eq power-add[symmetric])
    finally show ?thesis by simp
  next
    case False
    have set-pmf (sample-pro (geom-pro n)) ⊆ {..n}
      using geom-pro-range by simp
    hence ?L = measure (sample-pro (geom-pro n)) {}
      using False by (intro measure-pmf-cong) auto
    also have ... = ?R
      using False by simp
    finally show ?thesis
      by simp
  qed

  lemma geom-pro-prob-single:
    measure (sample-pro (geom-pro n)) {j} ≤ 1 / 2^j (is ?L ≤ ?R)
  proof −
```

have *?L* = *measure* (*sample-pro* (*geom-pro n*)) ({*j*..}−{*j*+*1*..})
by (*intro measure-pmf-cong*) *auto*
also have ... = *measure* (*sample-pro* (*geom-pro n*)) {*j*..} − *measure* (*sample-pro* (*geom-pro n*)) {*j*+*1*..}
by (*intro measure-Diff*) *auto*
also have ... = *measure* (*sample-pro* (*geom-pro n*)) {ω. ω ≥ *j*}−*measure* (*sample-pro* (*geom-pro n*)) {ω. ω ≥ (*j*+*1*)}
by (*intro arg-cong2*[**where** *f*=(−)] *measure-pmf-cong*) *auto*
also have ... = *of-bool* (*j* ≤ *n*) ∗ *1* / *2* ^ *j* − *of-bool* (*j* + *1* ≤ *n*) / *2* ^ (*j* + *1*)
unfolding *geom-pro-prob* **by** *simp*
also have ... ≤ *1*/*2*^*j* − *0*
by (*intro diff-mono*) *auto*
also have ... = *?R* **by** *simp*
finally show *?thesis* **by** *simp*
qed

definition *prod-pro* ::
'a pseudorandom-object ⇒ *'b pseudorandom-object* ⇒ (*'a* × *'b*) *pseudorandom-object*
where
prod-pro P Q =
(| *pro-last* = *pro-size P* ∗ *pro-size Q* − *1*,
pro-select = (λ*k*. (*pro-select P* (*k mod pro-size P*), *pro-select Q* (*k div pro-size P*))) |)

lemma *prod-pro-size*:
pro-size (*prod-pro P Q*) = *pro-size P* ∗ *pro-size Q*
unfolding *prod-pro-def* **by** (*subst pro-size-def*) (*simp add:pro-size-gt-0*)

lemma *prod-pro*:
sample-pro (*prod-pro P Q*) = *pair-pmf* (*sample-pro P*) (*sample-pro Q*) (**is** *?L* = *?R*)
proof −
let *?p* = *pro-size P*
let *?q* = *pro-size Q*
have *?L* = *map-pmf* (λ*k*. (*pro-select P* (*k mod ?p*),*pro-select Q* (*k div ?p*))) (*pmf-of-set*{..<*?p*∗*?q*})
unfolding *sample-pro-alt prod-pro-size* **by** (*simp add:prod-pro-def*)
also have ... = *map-pmf* (*map-prod* (*pro-select P*) (*pro-select Q*))
(*map-pmf* (λ*k*. (*k mod ?p*, *k div ?p*)) (*pmf-of-set*{..<*?p*∗*?q*}))
unfolding *map-pmf-comp* **by** *simp*
also have ... = *?R*
unfolding *split-pmf-mod-div*[*OF pro-size-gt-0 pro-size-gt-0*] *sample-pro-alt map-prod-def map-pair*
by *simp*
finally show *?thesis* **by** *simp*
qed

lemma *prod-pro-set*:
pro-set (*prod-pro* *P* *Q*) = *pro-set* *P* × *pro-set* *Q*
unfolding *prod-pro set-pair-pmf* **by** *simp*

end

6 K-Independent Hash Families as Pseudorandom Objects

theory *Pseudorandom-Objects-Hash-Families*
imports
Pseudorandom-Objects
Finite-Fields.Find-Irreducible-Poly
Carter-Wegman-Hash-Family
Universal-Hash-Families-More-Product-PMF
begin

hide-const (**open**) *Numeral-Type.mod-ring*
hide-const (**open**) *Divisibility.prime*
hide-const (**open**) *Isolated.discrete*

definition *hash-space′* ::
(′*a*,′*b*) *idx-ring-enum-scheme* ⇒ *nat* ⇒ (′*c*,′*d*) *pseudorandom-object-scheme*
⇒ (*nat* ⇒ ′*c*) *pseudorandom-object*
where *hash-space′ R k S* = (
⦇
pro-last = *idx-size R* ^*k*−*1*,
pro-select = (λ*x i*.
pro-select S
(*idx-enum-inv R* (*poly-eval R* (*poly-enum R k x*) (*idx-enum R i*)) *mod pro-size*
S))
⦈)

lemma *hash-prob-single′*:
assumes *field F finite* (*carrier F*)
assumes *x* ∈ *carrier F*
assumes *1* ≤ *n*
shows *measure* (*pmf-of-set* (*bounded-degree-polynomials F n*)) {ω. *ring.hash F x*
ω = *y*} =
of-bool (*y*∈ *carrier F*)/(*real* (*card* (*carrier F*))) (**is** *?L* = *?R*)
proof (*cases y* ∈ *carrier F*)
case *True*
have *?L* = 𝒫(ω *in pmf-of-set* (*bounded-degree-polynomials F n*). *ring.hash F x* ω
= *y*) **by** *simp*
also have ... = *1* / (*real* (*card* (*carrier F*))) **by** (*intro hash-prob-single assms*
conjI True)
also have ... = *?R* **using** *True* **by** *simp*
finally show *?thesis* **by** *simp*

next
  case *False*
  interpret *field F* **using** *assms* **by** *simp*
  have *fin-carr*: *finite* (*carrier F*) **using** *assms* **by** *simp*
  note *S* = *non-empty-bounded-degree-polynomials fin-degree-bounded*[*OF fin-carr*]
  let *?S* = *bounded-degree-polynomials F n*

  have *hash x f* ≠ *y* **if** *f* ∈ *?S* **for** *f*
  proof −
    have *hash x f* ∈ *carrier F*
      using *that* **unfolding** *hash-def bounded-degree-polynomials-def*
      by (*intro eval-in-carrier assms*) (*simp add*: *polynomial-incl univ-poly-carrier*)
    thus *?thesis* **using** *False* **by** *auto*
  qed
  hence *?L* = *measure* (*pmf-of-set* (*bounded-degree-polynomials F n*)) {}
    using *S* **by** (*intro measure-eq-AE AE-pmfI*) *simp-all*
  also have ... = *?R* **using** *False* **by** *simp*
  finally show *?thesis* **by** *simp*
qed

lemma *hash-k-wise-indep′*:
  assumes *field F* ∧ *finite* (*carrier F*)
  assumes *1* ≤ *n*
  shows *prob-space.k-wise-indep-vars* (*pmf-of-set* (*bounded-degree-polynomials F n*)) *n*
    (λ-. *discrete*) (*ring.hash F*) (*carrier F*)
  by (*intro prob-space.k-wise-indep-vars-compose*[*OF - hash-k-wise-indep*[*OF assms*]]
      prob-space-measure-pmf) *auto*

lemma *hash-space′*:
  fixes *R* :: (′*a*,′*b*) *idx-ring-enum-scheme*
  assumes $enum_C$ *R* $field_C$ *R*
  assumes *pro-size S dvd order* (*ring-of R*)
  assumes *I* ⊆ {..<*order* (*ring-of R*)} *card I* ≤ *k*
  shows *map-pmf* (λ*f*. (λ*i*∈*I*. *f i*)) (*sample-pro* (*hash-space′ R k S*)) = *prod-pmf I* (λ-. *sample-pro S*)
    (**is** *?L* = *?R*)
proof (*cases I* = {})
  case *False*
  let *?b* = *idx-size R*
  let *?s* = *pro-size S*
  let *?t* = *?b div ?s*
  let *?g* = λ*x i*. *poly-eval R* (*poly-enum R k x*) (*idx-enum R i*)
  let *?f* = λ*x*. *pro-select S* (*idx-enum-inv R x mod ?s*)
  let *?R-pmf* = *pmf-of-set* (*carrier* (*ring-of R*))
  let *?S* = {*xs* ∈ *carrier* (*poly-ring* (*ring-of R*)). *length xs* ≤ *k*}
  let *?T* = *pmf-of-set* (*bounded-degree-polynomials* (*ring-of R*) *k*)

  interpret *field ring-of R* **using** *assms*(*2*) **unfolding** $field_C$*-def* **by** *auto*

have *ring-c*: $ring_C$ *R* **using** *field-c-imp-ring assms*(*2*) **by** *auto*
note *enum-c* = *enum-cD*[*OF assms*(*1*)]

have *fin-carr*: *finite* (*carrier* (*ring-of R*)) **using** *enum-c* **by** *simp*

have $0 <$ *card I* **using** *False assms*(*4*) *card-gt-0-iff finite-nat-iff-bounded* **by** *blast*
also have ... $\leq k$ **using** *assms*(*5*) **by** *simp*
finally have *k-gt-0*: $k > 0$ **by** *simp*
have *b-gt-0*: $?b > 0$ **unfolding** *enum-c*(*2*) **using** *fin-carr order-gt-0-iff-finite* **by** *blast*
hence *t-gt-0*: $?t > 0$ **using** *enum-c*(*2*) *assms*(*3*) *dvd-div-gt0* **by** *simp*
have *b-k-gt-0*: $?b \mathbin{\hat{}} k > 0$ **using** *b-gt-0* **by** *simp*

have *fin-I*: *finite I* **using** *assms*(*4*) *finite-subset* **by** *auto*

have *inj*: *inj-on* (*idx-enum R*) *I*
using *assms*(*4*) **unfolding** *enum-c*(*2*)
by (*intro inj-on-subset*[*OF bij-betw-imp-inj-on*[*OF enum-c*(*3*)]])
have *card* (*idx-enum R ‘ I*) $\leq k$
using *assms*(*5*) **unfolding** *card-image*[*OF inj*] **by** *auto*

hence *prob-space.indep-vars ?T* (λ-. *discrete*) *hash* (*idx-enum R ‘ I*)
using *assms*(*4*) *k-gt-0 fin-I bij-betw-apply*[*OF enum-c*(*3*)] *enum-c*(*2*)
by (*intro prob-space.k-wise-indep-vars-subset*[*OF - hash-k-wise-indep'*]
prob-space-measure-pmf conjI fin-carr field-axioms) *auto*
hence *prob-space.indep-vars ?T* ((λ-. *discrete*) $\circ$ *idx-enum R*) ($\lambda x\ \omega$. *eval* ω (*idx-enum R x*)) *I*
using *inj* **unfolding** *hash-def*
by (*intro prob-space.indep-vars-reindex prob-space-measure-pmf*) *auto*
hence *indep*: *prob-space.indep-vars ?T* (λ-. *discrete*) ($\lambda x\ \omega$. *eval* ω (*idx-enum R x*)) *I*
by (*simp add*:*comp-def*)

have *0*: *pmf* (*map-pmf* (λx. $\lambda i{\in}I$. *eval x* (*idx-enum R i*)) *?T*) ω = *pmf* (*prod-pmf I* (λ-. *?R-pmf*)) ω
(**is** *?L1* = *?R1*) **for** ω
proof (*cases* $\omega \in$ *extensional I*)
case *True*
have *?L1* = *measure ?T* {*x*. ($\lambda i{\in}I$. *eval x* (*idx-enum R i*)) = ω}
by (*simp add*:*pmf-map vimage-def*)
also have ... = *measure ?T* {*x*. ($\forall\, i{\in}I$. *eval x* (*idx-enum R i*) = ω *i*)}
using *True* **unfolding** *restrict-def extensional-def*
by (*intro arg-cong2*[**where** *f*=*measure*] *refl Collect-cong*) *auto*
also have ... = ($\prod i{\in}I$. *measure ?T* {*x*. *eval x* (*idx-enum R i*) = ω *i*})
by (*intro prob-space.split-indep-events*[**where** *I*=*I* **and** *p*=*?T*] *prob-space-measure-pmf fin-I refl prob-space.indep-vars-compose2*[*OF - indep*]) *auto*
also have ... = ($\prod i{\in}I$. *measure ?T* {*x*. *hash* (*idx-enum R i*) *x* = ω *i*})
unfolding *hash-def* **by** *simp*

also have ... = ($\prod i{\in}I.$ *of-bool*(ω $i \in$ *carrier* (*ring-of R*))/*real* (*card* (*carrier* (*ring-of R*))))
using *k-gt-0 assms*(*4*) **by** (*intro prod.cong refl hash-prob-single'* *bij-betw-apply*[*OF enum-c*(*3*)] *fin-carr field-axioms*) (*auto simp*:*enum-c*)
also have ... = ($\prod i{\in}I.$ *pmf* (*pmf-of-set* (*carrier* (*ring-of R*))) (ω i))
using *fin-carr carrier-not-empty* **by** (*simp add*:*indicator-def*)
also have ... = *?R1*
using *True* **unfolding** *pmf-prod-pmf*[*OF fin-I*] **by** *simp*
finally show *?thesis* **by** *simp*
next
case *False*
have *?L1* = *0* **using** *False* **unfolding** *pmf-eq-0-set-pmf set-map-pmf* **by** *auto*
moreover have *?R1* = *0*
using *False* **unfolding** *pmf-eq-0-set-pmf set-prod-pmf*[*OF fin-I*] *PiE-def* **by** *simp*
ultimately show *?thesis* **by** *simp*
qed

have *map-pmf* ($\lambda x.$ $\lambda i{\in}I.$ *?g x i*) (*pmf-of-set* {..<*?b*^*k*}) =
map-pmf ($\lambda x.$ $\lambda i{\in}I.$ *poly-eval R x* (*idx-enum R i*)) (*map-pmf* (*poly-enum R k*) (*pmf-of-set* {..<*?b*^*k*}))
by (*simp add*:*map-pmf-comp*)
also have ... = *map-pmf* ($\lambda x.$ $\lambda i{\in}I.$ *poly-eval R x* (*idx-enum R i*)) (*pmf-of-set* *?S*)
using *b-k-gt-0* **by** (*intro arg-cong2*[**where** *f*=*map-pmf*] *refl map-pmf-of-set-bij-betw* *bij-betw-poly-enum assms*(*1*,*2*) *field-c-imp-ring*) *blast+*
also have ... = *map-pmf* ($\lambda x.$ $\lambda i{\in}I.$ *poly-eval R x* (*idx-enum R i*)) *?T*
using *k-gt-0* **unfolding** *bounded-degree-polynomials-def*
by (*intro map-pmf-cong refl arg-cong*[**where** *f*=*pmf-of-set*] *restrict-ext ring-c*) *auto*
also have ... = *map-pmf* ($\lambda x.$ $\lambda i{\in}I.$ *eval x* (*idx-enum R i*)) *?T*
using *non-empty-bounded-degree-polynomials fin-degree-bounded*[*OF fin-carr*] *assms*(*4*)
by (*intro map-pmf-cong poly-eval refl restrict-ext ring-c bij-betw-apply*[*OF enum-c*(*3*)])
(*auto simp add*:*bounded-degree-polynomials-def ring-of-poly*[*OF ring-c*] *enum-c*(*2*))
also have ... = *prod-pmf I* (λ-. *?R-pmf*) (**is** *?L1* = *?R1*)
by (*intro pmf-eqI 0*)
finally have *0*: *map-pmf* ($\lambda x.$ $\lambda i{\in}I.$ *?g x i*) (*pmf-of-set* {..<*?b*^*k*}) = *prod-pmf I* (λ-. *?R-pmf*)
by *simp*

have *1*: *map-pmf* ($\lambda x.$ *x mod ?s*) (*pmf-of-set* {..<*?b*}) = *pmf-of-set* {..<*?s*} (**is** *?L1*=*?R1*)
proof –
have *?L1* = *map-pmf fst* (*map-pmf* ($\lambda x.$ (*x mod ?s*, *x div ?s*)) (*pmf-of-set* {..<*?s***?t*}))
using *assms*(*3*) **by** (*simp add*:*map-pmf-comp enum-c*(*2*))
also have ... = *map-pmf fst* (*pmf-of-set* ({..<*?s*} $\times$ {..<*?t*}))

using *pro-size-gt-0 t-gt-0 lessThan-empty-iff finite-lessThan*
by (*intro arg-cong2*[**where** *f*=*map-pmf*] *refl map-pmf-of-set-bij-betw bij-betw-prod*) *force*+
also have ... = *map-pmf fst* (*pair-pmf* (*pmf-of-set* {..<*?s*}) (*pmf-of-set* {..<*?t*}))
using *pro-size-gt-0 t-gt-0* **by**(*intro arg-cong2*[**where** *f*=*map-pmf*] *pmf-of-set-prod-eq refl*) *auto*
also have ... = *pmf-of-set* {..<*?s*} **using** *map-fst-pair-pmf* **by** *blast*
finally show *?thesis* **by** *simp*
qed

have *map-pmf ?f ?R-pmf* = *map-pmf* (λ*x*. *pro-select S* (*x mod ?s*)) (*map-pmf* (*idx-enum-inv R*) *?R-pmf*)
by (*simp add*:*map-pmf-comp*)
also have ... = *map-pmf* (λ*x*. *pro-select S* (*x mod ?s*)) (*pmf-of-set* {..<*?b*})
using *enum-cD*(*1*,*2*,*4*)[*OF assms*(*1*)] *carrier-not-empty*
by (*intro arg-cong2*[**where** *f*=*map-pmf*] *refl map-pmf-of-set-bij-betw*) *auto*
also have ... = *map-pmf* (*pro-select S*) (*map-pmf* (λ*x*. *x mod ?s*) (*pmf-of-set* {..<*?b*}))
by (*simp add*:*map-pmf-comp*)
also have ... = *sample-pro S* **unfolding** *sample-pro-alt 1* **by** *simp*
finally have *2*:*map-pmf ?f ?R-pmf* = *sample-pro S* **by** *simp*

have *?L* = *map-pmf* (λ*x*. λ*i*∈*I*. *?f* (*?g x i*)) (*pmf-of-set* {..<*?b*^*k*})
using *b-k-gt-0* **unfolding** *sample-pro-alt hash-space′-def pro-size-def*
by (*simp add*: *map-pmf-comp del*:*poly-eval.simps*)
also have ... = *map-pmf* (λ*f*. λ*i*∈*I*. *?f* (*f i*)) (*map-pmf* (λ*x*. λ*i*∈*I*. *?g x i*) (*pmf-of-set* {..<*?b*^*k*}))
unfolding *map-pmf-comp* **by** (*intro arg-cong2*[**where** *f*=*map-pmf*] *refl restrict-ext ext*) *simp*
also have ... = *prod-pmf I* (λ-. *map-pmf ?f* (*pmf-of-set* (*carrier* (*ring-of R*)))) **unfolding** *0*
by (*simp add*:*map-pmf-def Pi-pmf-bind-return*[*OF fin-I*, **where** *d′*=*undefined*] *restrict-def*)
also have ... = *?R* **unfolding** *2* **by** *simp*
finally show *?thesis* **by** *simp*
next
case *True*
have *?L* = *map-pmf* (λ*f i*. *undefined*) (*sample-pro* (*hash-space′ R k S*))
using *True* **by** (*intro map-pmf-cong refl*) *auto*
also have ... = *return-pmf* (λ*f*. *undefined*) **unfolding** *map-pmf-const* **by** *simp*
also have ... = *?R* **using** *True* **by** *simp*
finally show *?L* = *?R* **by** *simp*
qed

lemma *hash-space′-range*:
pro-select (*hash-space′ R k S*) *i j* ∈ *pro-set S*
unfolding *hash-space′-def* **by** (*simp add*: *pro-select-in-set*)

definition *hash-pro* ::

nat ⇒ *nat* ⇒ (*'a*,*'b*) *pseudorandom-object-scheme* ⇒ (*nat* ⇒ *'a*) *pseudorandom-object*
where *hash-pro k d S* = (
let (*p*,*j*) = *split-power* (*pro-size S*);
l = *max j* (*floorlog p* (*d*−*1*))
in hash-space' (*GF* (*p*^*l*)) *k S*)

definition *hash-pro-spmf* ::
nat ⇒ *nat* ⇒ (*'a*,*'b*) *pseudorandom-object-scheme* ⇒ (*nat* ⇒ *'a*) *pseudorandom-object spmf*
where *hash-pro-spmf k d S* =
do {
let (*p*,*j*) = *split-power* (*pro-size S*);
let l = *max j* (*floorlog p* (*d*−*1*));
$R \leftarrow GF_R$ (*p*^*l*);
return-spmf (*hash-space' R k S*)
}

definition *hash-pro-pmf* ::
nat ⇒ *nat* ⇒ (*'a*,*'b*) *pseudorandom-object-scheme* ⇒ (*nat* ⇒ *'a*) *pseudorandom-object pmf*
where *hash-pro-pmf k d S* = *map-pmf the* (*hash-pro-spmf k d S*)

syntax
-FLIPBIND :: (*'a* ⇒ *'b*) ⇒ *'c* ⇒ *'b* (**infixr** ‹=<<› *54*)

syntax-consts
-FLIPBIND == *Monad-Syntax.bind*

translations
-FLIPBIND f g => *g* ≫= *f*

context
fixes *S*
fixes *d* :: *nat*
fixes *k* :: *nat*
assumes *size-prime-power*: *is-prime-power* (*pro-size S*)
begin

private definition *p* **where** *p* = *fst* (*split-power* (*pro-size S*))
private definition *j* **where** *j* = *snd* (*split-power* (*pro-size S*))
private definition *l* **where** *l* = *max j* (*floorlog p* (*d*−*1*))

private lemma *split-power*: (*p*,*j*) = *split-power* (*pro-size S*)
using *p-def j-def* **by** *auto*

private lemma *hash-sample-space-alt*: *hash-pro k d S* = *hash-space'* (*GF* (*p*^*l*)) *k S*
unfolding *hash-pro-def split-power*[*symmetric*] **by** (*simp add*:*j-def l-def Let-def*)

private lemma *p-prime* : *prime p* **and** *j-gt-0*: *j > 0*
proof −
  obtain *q r* **where** *0*:*pro-size S = q^r* **and** *q-prime*: *prime q* **and** *r-gt-0*: *r > 0*
    using *size-prime-power is-prime-power-def* **by** *blast*

  have (*p*,*j*) = *split-power* (*q^r*) **unfolding** *split-power 0* **by** *simp*
  also have ... = (*q*,*r*) **by** (*intro split-power-prime q-prime r-gt-0*)
  finally have (*p*,*j*) = (*q*,*r*) **by** *simp*
  thus *prime p j > 0* **using** *q-prime r-gt-0* **by** *auto*
qed

private lemma *l-gt-0*: *l > 0*
  unfolding *l-def* **using** *j-gt-0* **by** *simp*

private lemma *prime-power*: *is-prime-power* (*p^l*)
  using *p-prime l-gt-0* **unfolding** *is-prime-power-def* **by** *auto*

lemma *hash-in-hash-pro-spmf*: *hash-pro k d S* ∈ *set-spmf* (*hash-pro-spmf k d S*)
  using *GF-in-GF-R*[*OF prime-power*]
  unfolding *hash-pro-def hash-pro-spmf-def split-power*[*symmetric*] *l-def* **by** (*auto simp add*:*set-bind-spmf*)

lemma *lossless-hash-pro-spmf*: *lossless-spmf* (*hash-pro-spmf k d S*)
proof −
  have *lossless-spmf* (GF_R (*p^l*)) **by** (*intro galois-field-random-1 prime-power*)
  thus *?thesis* **unfolding** *hash-pro-spmf-def split-power*[*symmetric*] *l-def* **by** *simp*
qed

lemma *hashp-eq-hash-pro-spmf*: *set-pmf* (*hash-pro-pmf k d S*) = *set-spmf* (*hash-pro-spmf k d S*)
  unfolding *hash-pro-pmf-def* **using** *lossless-imp-spmf-of-pmf*[*OF lossless-hash-pro-spmf*]
  by (*metis set-spmf-spmf-of-pmf*)

lemma *hashp-in-hash-pro-spmf*:
  assumes *x* ∈ *set-pmf* (*hash-pro-pmf k d S*)
  shows *x* ∈ *set-spmf* (*hash-pro-spmf k d S*)
  using *hashp-eq-hash-pro-spmf assms* **by** *auto*

lemma *hash-pro-in-hash-pro-pmf*: *hash-pro k d S* ∈ *set-pmf* (*hash-pro-pmf k d S*)
  unfolding *hashp-eq-hash-pro-spmf* **by** (*intro hash-in-hash-pro-spmf*)

lemma *hash-pro-spmf-distr*:
  assumes *s* ∈ *set-spmf* (*hash-pro-spmf k d S*)
  assumes *I* ⊆ {..<*d*} *card I* ≤ *k*
  shows *map-pmf* (λ*f*. (λ*i*∈*I*. *f i*)) (*sample-pro s*) = *prod-pmf I* (λ-. *sample-pro S*)
proof −
  have (*d*−*1*) < *p^floorlog p* (*d*−*1*)

using *floorlog-leD prime-gt-1-nat*[*OF p-prime*] **by** *simp*
hence $d \leq p\hat{}floorlog\ p\ (d-1)$ **by** (*cases d*) *auto*
also have ... $\leq p\hat{}l$
using *prime-gt-0-nat*[*OF p-prime*] **unfolding** *l-def* **by** (*intro power-increasing*) *auto*
finally have *0*: $d \leq p\hat{}l$ **by** *simp*

obtain *R* **where** *R-in*: $R \in$ *set-spmf* (GF_R ($p\hat{}l$)) **and** *s-def*: $s =$ *hash-space'* *R k S*
using *assms*(*1*) **unfolding** *hash-pro-spmf-def split-power*[*symmetric*] *l-def*
by (*auto simp add*:*set-bind-spmf*)
have *1*: *order* (*ring-of R*) $= p\ \hat{}\ l$
using *galois-field-random-1*(*1*)[*OF prime-power R-in*] **by** *auto*
have $I \subseteq \{..<d\}$ **using** *assms* **by** *auto*
also have ... $\subseteq \{..<$*order* (*ring-of R*)$\}$ **using** *0* **unfolding** *1* **by** *auto*
finally have $I \subseteq \{..<$*order* (*ring-of R*)$\}$ **by** *simp*
moreover have $j \leq l$ **unfolding** *l-def* **by** *auto*
hence *pro-size S dvd order* (*ring-of R*)
unfolding *1 split-power-result*[*OF split-power*] **by** (*intro le-imp-power-dvd*)
ultimately show *?thesis*
using *galois-field-random-1*(*1*)[*OF prime-power R-in*] *assms*(*3*)
unfolding *s-def* **by** (*intro hash-space'*) *simp-all*
qed

lemma *hash-pro-spmf-component*:
assumes $s \in$ *set-spmf* (*hash-pro-spmf k d S*)
assumes $i < d\ k > 0$
shows *map-pmf* ($\lambda f.\ f\ i$) (*sample-pro s*) $=$ *sample-pro S* (**is** *?L* = *?R*)
proof −
have *?L* = *map-pmf* ($\lambda f.\ f\ i$) (*map-pmf* ($\lambda f.$ ($\lambda i \in \{i\}.\ f\ i$)) (*sample-pro s*))
using *assms*(*1*) **unfolding** *map-pmf-comp* **by** (*intro map-pmf-cong refl*) *auto*
also have ... = *map-pmf* ($\lambda f.\ f\ i$) (*prod-pmf* $\{i\}$ (λ-. *sample-pro S*))
using *assms* **by** (*subst hash-pro-spmf-distr*[*OF assms*(*1*)]) *auto*
also have ... = *?R* **by** (*subst Pi-pmf-component*) *auto*
finally show *?thesis* **by** *simp*
qed

lemma *hash-pro-spmf-indep*:
assumes $s \in$ *set-spmf* (*hash-pro-spmf k d S*)
assumes $I \subseteq \{..<d\}$ *card* $I \leq k$
shows *prob-space.indep-vars* (*sample-pro s*) (λ-. *discrete*) ($\lambda i\ \omega.\ \omega\ i$) *I*
proof (*rule measure-pmf.indep-vars-pmf*[*OF refl*])
fix *x J*
assume *a*:$J \subseteq I$
have *0*:$J \subseteq \{..<d\}$ **using** *a assms*(*2*) **by** *auto*
have *card* $J \leq$ *card* *I* **using** *finite-subset*[*OF assms*(*2*)] **by** (*intro card-mono a*) *auto*
also have ... $\leq k$ **using** *assms*(*3*) **by** *simp*
finally have *1*: *card* $J \leq k$ **by** *simp*

let *?s* = *sample-pro s*

have *2*: *0* < *k* **if** $x \in J$ **for** *x*
proof −
have *0* < *card J* **using** *0 that card-gt-0-iff finite-nat-iff-bounded* **by** *auto*
also have ... ≤ *k* **using** *1* **by** *simp*
finally show *?thesis* **by** *simp*
qed

have *measure ?s* {ω. ∀ j∈J. ω j = x j} = *measure* (*map-pmf* (λω. λj∈J. ω j)*?s*) {ω. ∀ j∈J. ω j = x j}
by *auto*
also have ... = *measure* (*prod-pmf J* (λ-. *sample-pro S*)) (*Pi J* (λj. {x j}))
unfolding *hash-pro-spmf-distr*[*OF assms*(*1*) *0 1*] **by** (*intro arg-cong2*[**where** *f*=*measure*]) (*auto simp*:*Pi-def*)
also have ... = (∏ j∈J. *measure* (*sample-pro S*) {x j})
using *finite-subset*[*OF a*] *finite-subset*[*OF assms*(*2*)] **by** (*intro measure-Pi-pmf-Pi*) *auto*
also have ... = (∏ j∈J. *measure* (*map-pmf* (λω. ω j) *?s*) {x j})
using *0 1 2* **by** (*intro prod.cong arg-cong2*[**where** *f*=*measure*] *refl*
arg-cong[**where** *f*=*measure-pmf*] *hash-pro-spmf-component*[*OF assms*(*1*), *symmetric*]) *auto*
also have ... = (∏ j∈J. *measure ?s* {ω. ω j = x j}) **by** (*simp add*:*vimage-def*)
finally show *measure ?s* {ω. ∀ j∈J. ω j = x j} = (∏ j∈J. *measure-pmf.prob ?s* {ω. ω j = x j})
by *simp*
qed

lemma *hash-pro-spmf-k-indep*:
assumes $s \in$ *set-spmf* (*hash-pro-spmf k d S*)
shows *prob-space.k-wise-indep-vars* (*sample-pro s*) *k* (λ-. *discrete*) (λi ω. ω i) {..<d}
using *hash-pro-spmf-indep*[*OF assms*]
unfolding *prob-space.k-wise-indep-vars-def*[*OF prob-space-measure-pmf*] **by** *auto*

private lemma *hash-pro-spmf-size-aux*:
assumes $s \in$ *set-spmf* (*hash-pro-spmf k d S*)
shows *pro-size s* = (p^l)^k (**is** *?L* = *?R*)
proof −
obtain *R* **where** *R-in*: $R \in$ *set-spmf* (GF_R (p^l)) **and** *s-def*: *s* = *hash-space′ R k S*
using *assms*(*1*) **unfolding** *hash-pro-spmf-def split-power*[*symmetric*] *l-def*
by (*auto simp add*:*set-bind-spmf*)
have *1*: *order* (*ring-of R*) = *p* ^ *l* **and** *ec*: $enum_C$ *R*
using *galois-field-random-1*(*1*)[*OF prime-power R-in*] **by** *auto*

have *?L* = *idx-size R* ^ *k* − *1* + *1*
unfolding *s-def pro-size-def hash-space′-def* **by** *simp*

also have ... = $((p\hat{\ }l)\hat{\ }k - 1) + 1$
using *1 enum-cD(2)[OF ec]* **by** *simp*
also have ... = $(p\hat{\ }l)\hat{\ }k$ **using** *prime-gt-0-nat[OF p-prime]* **by** *simp*
finally show *?thesis* **by** *simp*
qed

lemma *floorlog-alt-def*:
floorlog b a = (*if* $1 < b$ *then nat* $\lceil$*log* (*real b*) (*real a*+1)$\rceil$ *else 0*)
proof (*cases* $a > 0 \wedge 1 < b$)
case *True*
have *1*:*log* (*real b*) (*real a* + 1) > 0 **using** *True* **by** (*subst zero-less-log-cancel-iff*) *auto*

have *a* < *real a* + *1* **by** *simp*
also have ... = *b powr* (*log b* (*real a* + *1*)) **using** *True* **by** *simp*
also have ... ≤ *b powr* ($\lceil$*log b* (*real a* + *1*)$\rceil$)
using *True* **by** (*intro iffD2[OF powr-le-cancel-iff]*) *auto*
also have ... = *b powr* (*real* (*nat* $\lceil$*log b* (*real a* + *1*)$\rceil$))
using *1* **by** (*intro arg-cong2[***where** *f=(powr)] refl*) *linarith*
also have ... = *b* ^ *nat* $\lceil$*log* (*real b*) (*real a* + *1*)$\rceil$ **using** *True* **by** (*subst powr-realpow*) *auto*
finally have *a* < *b* ^ *nat* $\lceil$*log* (*real b*) (*real a* + *1*)$\rceil$ **by** *simp*
hence *0*:*floorlog b a* ≤ *nat* $\lceil$*log* (*real b*) (*real a*+*1*)$\rceil$ **using** *True* **by** (*intro floorlog-leI*) *auto*

have *b* ^(*nat* $\lceil$*log b* (*real a* + *1*)$\rceil$ − *1*) = *b powr* (*real* (*nat* $\lceil$*log b* (*real a* + *1*)$\rceil$ − *1*))
using *True* **by** (*subst powr-realpow*) *auto*
also have ... = *b powr* ($\lceil$*log b* (*real a* + *1*)$\rceil$ − *1*)
using *1* **by** (*intro arg-cong2[***where** *f=(powr)] refl*) *linarith*
also have ... < *b powr* (*log b* (*real a* + *1*)) **using** *True* **by** (*intro powr-less-mono*) *linarith+*
also have ... = *real* (*a* + *1*) **using** *True* **by** *simp*
finally have *b* ^(*nat* $\lceil$*log* (*real b*) (*real a* + *1*)$\rceil$ − *1*) < *a* + *1* **by** *linarith*
hence *b* ^(*nat* $\lceil$*log* (*real b*) (*real a* + *1*)$\rceil$ − *1*) ≤ *a* **by** *simp*
hence *floorlog b a* ≥ *nat* $\lceil$*log* (*real b*) (*real a*+*1*)$\rceil$ **using** *True* **by** (*intro floorlog-geI*) *auto*
hence *floorlog b a* = *nat* $\lceil$*log* (*real b*) (*real a*+*1*)$\rceil$ **using** *0* **by** *linarith*
also have ... = (*if 1* < *b then nat* $\lceil$*log* (*real b*) (*real a*+*1*)$\rceil$ *else 0*) **using** *True* **by** *simp*
finally show *?thesis* **by** *simp*
next
case *False*
hence *a-eq-0*: *a* = *0* ∨ ¬(*1* < *b*) **by** *simp*
thus *?thesis* **unfolding** *floorlog-def* **by** *auto*
qed

lemma *hash-pro-spmf-size*:
assumes *s* ∈ *set-spmf* (*hash-pro-spmf k d S*)

assumes $(p',j') = split\text{-}power\ (pro\text{-}size\ S)$
shows $pro\text{-}size\ s = (p'\hat{\ }(max\ j'\ (floorlog\ p'\ (d-1))))\hat{\ }k$
unfolding *hash-pro-spmf-size-aux*[*OF assms*(*1*)] *l-def p-def j-def* **using** *assms*(*2*)
by (*metis fst-conv snd-conv*)

lemma *hash-pro-spmf-size'*:
assumes $s \in set\text{-}spmf\ (hash\text{-}pro\text{-}spmf\ k\ d\ S)\ d > 0$
assumes $(p',j') = split\text{-}power\ (pro\text{-}size\ S)$
shows $pro\text{-}size\ s = (p'\hat{\ }(k*max\ j'\ (nat\ \lceil log\ p'\ d\rceil)))$
proof −
have $pro\text{-}size\ s = (p\hat{\ }(max\ j\ (floorlog\ p\ (d-1))))\hat{\ }k$
unfolding *hash-pro-spmf-size-aux*[*OF assms*(*1*)] *l-def* **by** *simp*
also have ... $= (p\hat{\ }(max\ j\ (nat\ \lceil log\ p\ (real\ (d-1)+1)\rceil)))\hat{\ }k$
using *prime-gt-1-nat*[*OF p-prime*] **by** (*simp add*:*floorlog-alt-def*)
also have ... $= (p\hat{\ }(max\ j\ (nat\ \lceil log\ p\ d\rceil)))\hat{\ }k$ **using** *assms*(*2*) **by** (*subst of-nat-diff*)
auto
also have ... $= p\hat{\ }(k*max\ j\ (nat\ \lceil log\ p\ d\rceil))$ **by** (*simp add*:*ac-simps power-mult*[*symmetric*])
also have ... $= p'\hat{\ }(k*max\ j'\ (nat\ \lceil log\ p'\ d\rceil))$
using *assms*(*3*) *p-def j-def* **by** (*metis fst-conv snd-conv*)
finally show *?thesis* **by** *simp*
qed

lemma *hash-pro-spmf-size-prime-power*:
assumes $s \in set\text{-}spmf\ (hash\text{-}pro\text{-}spmf\ k\ d\ S)$
assumes $k > 0$
shows *is-prime-power* (*pro-size s*)
unfolding *hash-pro-spmf-size-aux*[*OF assms*(*1*)] *power-mult*[*symmetric*] *is-prime-power-def*
using *p-prime mult-pos-pos*[*OF l-gt-0 assms*(*2*)] **by** *blast*

lemma *hash-pro-smpf-range*:
assumes $s \in set\text{-}spmf\ (hash\text{-}pro\text{-}spmf\ k\ d\ S)$
shows $pro\text{-}select\ s\ i\ q \in pro\text{-}set\ S$
proof −
obtain *R* **where** *R-in*: $R \in set\text{-}spmf\ (GF_R\ (p\hat{\ }l))$ **and** *s-def*: $s = hash\text{-}space'\ R$
$k\ S$
using *assms*(*1*) **unfolding** *hash-pro-spmf-def split-power*[*symmetric*] *l-def*
by (*auto simp add*:*set-bind-spmf*)
thus *?thesis*
unfolding *s-def* **using** *hash-space'-range* **by** *auto*
qed

lemmas *hash-pro-size'* = *hash-pro-spmf-size'*[*OF hash-in-hash-pro-spmf*]
lemmas *hash-pro-size* = *hash-pro-spmf-size*[*OF hash-in-hash-pro-spmf*]
lemmas *hash-pro-size-prime-power* = *hash-pro-spmf-size-prime-power*[*OF hash-in-hash-pro-spmf*]
lemmas *hash-pro-distr* = *hash-pro-spmf-distr*[*OF hash-in-hash-pro-spmf*]
lemmas *hash-pro-component* = *hash-pro-spmf-component*[*OF hash-in-hash-pro-spmf*]
lemmas *hash-pro-indep* = *hash-pro-spmf-indep*[*OF hash-in-hash-pro-spmf*]
lemmas *hash-pro-k-indep* = *hash-pro-spmf-k-indep*[*OF hash-in-hash-pro-spmf*]
lemmas *hash-pro-range* = *hash-pro-smpf-range*[*OF hash-in-hash-pro-spmf*]

lemmas *hash-pro-pmf-size′* = *hash-pro-spmf-size′*[*OF hashp-in-hash-pro-spmf*]
lemmas *hash-pro-pmf-size* = *hash-pro-spmf-size*[*OF hashp-in-hash-pro-spmf*]
lemmas *hash-pro-pmf-size-prime-power* = *hash-pro-spmf-size-prime-power*[*OF hashp-in-hash-pro-spmf*]
lemmas *hash-pro-pmf-distr* = *hash-pro-spmf-distr*[*OF hashp-in-hash-pro-spmf*]
lemmas *hash-pro-pmf-component* = *hash-pro-spmf-component*[*OF hashp-in-hash-pro-spmf*]
lemmas *hash-pro-pmf-indep* = *hash-pro-spmf-indep*[*OF hashp-in-hash-pro-spmf*]
lemmas *hash-pro-pmf-k-indep* = *hash-pro-spmf-k-indep*[*OF hashp-in-hash-pro-spmf*]
lemmas *hash-pro-pmf-range* = *hash-pro-smpf-range*[*OF hashp-in-hash-pro-spmf*]

end

open-bundle *pseudorandom-object-syntax*
begin
notation *hash-pro* (‹$\mathcal{H}$›)
notation *hash-pro-spmf* (‹$\mathcal{H}_S$›)
notation *hash-pro-pmf* (‹$\mathcal{H}_P$›)
notation *list-pro* (‹$\mathcal{L}$›)
notation *nat-pro* (‹$\mathcal{N}$›)
notation *geom-pro* (‹$\mathcal{G}$›)
notation *prod-pro* (**infixr** ‹$\times_P$› *65*)
end

end

References

[1] E. Karayel. Interpolation polynomials (in hol-algebra). *Archive of Formal Proofs*, Jan. 2022. https://isa-afp.org/entries/Interpolation_Polynomials_HOL_Algebra.html, Formal proof development.

[2] M. Thorup and Y. Zhang. Tabulation based 5-universal hashing and linear probing. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, ALENEX '10, pages 62–76, USA, 2010. Society for Industrial and Applied Mathematics.

[3] S. P. Vadhan. Pseudorandomness. *Foundations and Trends®in Theoretical Computer Science*, 7(1-3):1–336, 2012.

[4] M. N. Wegman and J. L. Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22(3):265–279, 1981.