

Universal Hash Families

Emin Karayel

May 26, 2024

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 $M' X I =$
 $(\forall J \subseteq I. \text{card } J \leq k \longrightarrow \text{finite } J \longrightarrow \text{indep-vars } M' X J)$

lemma (in *prob-space*) *k-wise-indep-vars-subset*:
assumes *k-wise-indep-vars* k $M' X I$
assumes $J \subseteq I$
assumes *finite* J
assumes $\text{card } J \leq k$
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 $J \subseteq I$
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* $X A =$ (
 $\text{distr } M (\text{count-space UNIV}) X = \text{uniform-measure } (\text{count-space UNIV}) A \wedge$
 $A \neq \{\} \wedge \text{finite } A \wedge \text{random-variable } (\text{count-space UNIV}) X)$

lemma (in *prob-space*) *uniform-onD*:
assumes *uniform-on* $X A$
shows $\text{prob } \{\omega \in \text{space } M. X \omega \in B\} = \text{card } (A \cap B) / \text{card } A$
proof –
have $\text{prob } \{\omega \in \text{space } M. X \omega \in B\} = \text{prob } (X \text{ -' } B \cap \text{space } M)$
by (*subst Int-commute, simp add:vimage-def Int-def*)
also have $\dots = \text{measure } (\text{distr } M (\text{count-space UNIV}) X) B$
using *assms by (subst measure-distr, auto simp:uniform-on-def)*
also have $\dots = \text{measure } (\text{uniform-measure } (\text{count-space UNIV}) A) B$
using *assms by (simp add:uniform-on-def)*
also have $\dots = \text{card } (A \cap B) / \text{card } A$
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 $D \wedge$
 $(\forall i \in D. \text{uniform-on } (X\ i)\ R))$

Note: The definition is slightly more generic than 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 $\bigwedge x. P\ x \implies f\ x = h\ (g\ x)$

shows $\{f\ x \mid x. P\ x\} = h\ \text{' } \{g\ x \mid x. P\ x\}$

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 $\bigwedge i. i \in I \implies Y\ i \in \text{measurable } (M'\ i)\ (N\ i)$

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 $F \equiv (\lambda i. (\lambda a. f - ' a \cap \text{space } M) ' A i)$

assumes *indep-F*: *indep-sets* $F I$

assumes *sets-A*: $\bigwedge i. i \in I \implies A i \subseteq \text{sets } N$

shows *prob-space.indep-sets* (*distr* $M N f$) $A I$

proof (*rule prob-space.indep-setsI*)

show $\bigwedge A' J. J \neq \{\} \implies J \subseteq I \implies \text{finite } J \implies \forall j \in J. A' j \in A j \implies$

$\text{measure } (\text{distr } M N f) (\bigcap (A' ' J)) = (\prod_{j \in J. \text{measure } (\text{distr } M N f) (A' j))$

proof –

fix $A' J$

assume $a: J \subseteq I \text{ finite } J J \neq \{\} \forall j \in J. A' j \in A j$

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

have $\bigcap (F' ' J) = f - ' (\bigcap (A' ' J)) \cap \text{space } M$

unfolding *set-eq-iff* F' -*def* **using** $a(3)$ **by** *simp*

moreover **have** $\bigcap (A' ' J) \in \text{sets } N$

by (*metis* a *sets-A* *sets.finite-INT* *subset-iff*)

ultimately **have** b :

$\text{measure } (\text{distr } M N f) (\bigcap (A' ' J)) = \text{measure } M (\bigcap (F' ' J))$

by (*metis* *assms(1)* *measure-distr*)

have $\bigwedge j. j \in J \implies F' j \in F j$

using $a(4)$ F' -*def* F -*def* **by** *blast*

hence $c: \text{measure } M (\bigcap (F' ' J)) = (\prod_{j \in J. \text{measure } M (F' j))$

by (*metis* *indep-F* *indep-setsD* $a(1,2,3)$)

have $\bigwedge j. j \in J \implies F' j = f - ' A' j \cap \text{space } M$

by (*simp* *add:F'-def*)

moreover **have** $\bigwedge j. j \in J \implies A' j \in \text{sets } N$

using $a(1,4)$ *sets-A* **by** *blast*

ultimately **have** d :

$\bigwedge j. j \in J \implies \text{measure } M (F' j) = \text{measure } (\text{distr } M N f) (A' j)$

using *assms(1)* *measure-distr* **by** *metis*

show

$\text{measure } (\text{distr } M N f) (\bigcap (A' ' J)) = (\prod_{j \in J. \text{measure } (\text{distr } M N f) (A' j))$

using $b c d$ **by** *auto*

qed

show *prob-space* (*distr* $M N f$) **using** *prob-space-distr* *assms* **by** *blast*

show $\bigwedge i. i \in I \implies A i \subseteq \text{sets } (\text{distr } M N f)$ **using** *sets-A* *sets-distr* **by** *blast*

qed

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

```

assumes  $f \in \text{measurable } M \ N$ 
assumes  $\bigwedge i. i \in I \implies X' i \in \text{measurable } N \ (M' i)$ 
assumes  $\text{indep-vars } M' \ (\lambda i. (X' i) \circ f) \ I$ 
shows  $\text{prob-space.indep-vars } (\text{distr } M \ N \ f) \ M' \ X' \ I$ 
proof –
  interpret  $D: \text{prob-space } (\text{distr } M \ N \ f)$ 
  using  $\text{prob-space-distr}[OF \ \text{assms}(1)]$  by  $\text{simp}$ 

  have  $a: f \in \text{space } M \rightarrow \text{space } N$  using  $\text{assms}(1)$  by  $(\text{simp } \text{add:measurable-def})$ 

  have  $D.\text{indep-sets } (\lambda i. \{X' i -' A \cap \text{space } N \mid A. A \in \text{sets } (M' i)\}) \ I$ 
  proof  $(\text{rule } \text{indep-sets-distr}[OF \ \text{assms}(1)])$ 
    have  $\bigwedge i. i \in I \implies \{(X' i \circ f) -' A \cap \text{space } M \mid A. A \in \text{sets } (M' i)\} =$ 
       $(\lambda a. f -' a \cap \text{space } M) -' \{X' i -' A \cap \text{space } N \mid A. A \in \text{sets } (M' i)\}$ 
    by  $(\text{rule } \text{set-comp-image-cong}, \text{simp } \text{add:set-eq-iff}, \text{use } a \text{ in } \text{blast})$ 
    thus  $\text{indep-sets } (\lambda i. (\lambda a. f -' a \cap \text{space } M) -'$ 
       $\{X' i -' A \cap \text{space } N \mid A. A \in \text{sets } (M' i)\}) \ I$ 
    using  $\text{assms}(3)$  by  $(\text{simp } \text{add:indep-vars-def2 } \text{cong:indep-sets-cong})$ 
  next
  fix  $i$ 
  assume  $i \in I$ 
  thus  $\{X' i -' A \cap \text{space } N \mid A. A \in \text{sets } (M' i)\} \subseteq \text{sets } N$ 
  using  $\text{assms}(2)$   $\text{measurable-sets}$  by  $\text{blast}$ 
qed
thus  $?thesis$ 
using  $\text{assms}$  by  $(\text{simp } \text{add:D.indep-vars-def2})$ 
qed

```

```

lemma  $\text{range-inter: range } ((\cap) \ F) = \text{Pow } F$ 
unfolding  $\text{image-def}$  by  $\text{auto}$ 

```

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

```

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

```

In some of the following theorems, the premise $M = \text{measure-pmf } p$ is used. This allows stating theorems that hold for pmfs more concisely, for example,

instead of $\text{measure-pmf.prob } p \ A \leq \text{measure-pmf.prob } p \ B$ we can just write $M = \text{measure-pmf } p \implies \text{prob } A \leq \text{prob } B$ in the locale prob-space .

lemma *prob-space-restrict-space*:

assumes $[\text{simp}]: M = \text{measure-pmf } p$

shows $\text{prob-space } (\text{restrict-space } M \ (\text{set-pmf } p))$

by $(\text{rule } \text{prob-spaceI}, \text{auto } \text{simp}:\text{emeasure-restrict-space } \text{emeasure-pmf})$

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 $\text{discrete} \equiv \text{count-space } UNIV$

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

assumes $[\text{simp}]: M = \text{measure-pmf } p$

assumes

$\text{prob-space.indep-vars } (\text{restrict-space } M \ (\text{set-pmf } p)) \ (\lambda-. \text{discrete}) \ X \ I$

shows $\text{indep-vars } (\lambda-. \text{discrete}) \ X \ I$

proof –

have $a: \text{id} \in \text{restrict-space } M \ (\text{set-pmf } p) \rightarrow_M M$

by $(\text{simp } \text{add}:\text{measurable-def } \text{range-inter } \text{sets-restrict-space})$

have $\text{prob-space.indep-vars } (\text{distr } (\text{restrict-space } M \ (\text{set-pmf } p)) \ M \ \text{id}) \ (\lambda-. \text{discrete}) \ X \ I$

using $\text{assms } a \ \text{prob-space-restrict-space}$ **by** $(\text{auto } \text{intro}!: \text{prob-space.indep-vars-distr})$

moreover have

$\bigwedge A. \ \text{emeasure } (\text{distr } (\text{restrict-space } M \ (\text{set-pmf } p)) \ M \ \text{id}) \ A = \text{emeasure } M \ A$

using $\text{emeasure-distr}[OF \ a]$

by $(\text{auto } \text{simp } \text{add}:\text{emeasure-restrict-space } \text{emeasure-Int-set-pmf})$

hence $\text{distr } (\text{restrict-space } M \ p) \ M \ \text{id} = M$

by $(\text{auto } \text{intro}:\text{measure-eqI})$

ultimately show *?thesis* **by** *simp*

qed

lemma (in prob-space) *measure-pmf-eq*:

assumes $M = \text{measure-pmf } p$

assumes $\bigwedge x. x \in \text{set-pmf } p \implies (x \in P) = (x \in Q)$

shows $\text{prob } P = \text{prob } Q$

unfolding $\text{assms}(1)$

by $(\text{rule } \text{measure-eq-AE}, \text{rule } \text{AE-pmfI}[OF \ \text{assms}(2)], \text{auto})$

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*]: $M = \text{measure-pmf } p$

assumes $\bigwedge a J. J \subseteq I \implies \text{finite } J \implies$

$\text{prob } \{\omega. \forall i \in J. X \ i \ \omega = a \ i\} = (\prod i \in J. \text{prob } \{\omega. X \ i \ \omega = a \ i\})$

shows *indep-vars* ($\lambda \cdot$. *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*: $R.\text{events} = \text{Pow } (\text{set-pmf } p)$

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** $F = (\lambda i. \{X \ i \ -' a \ \cap \ \text{set-pmf } p \mid a. a \in G \ i\})$

have *sigma-sets-pow*:

$\bigwedge i. i \in I \implies \text{sigma-sets } (X \ i \ -' \ \text{set-pmf } p) (G \ i) = \text{Pow } (X \ i \ -' \ \text{set-pmf } p)$

by (*simp add:G-def, metis countable-image countable-set-pmf sigma-sets-singletons-and-empty*)

have *F-in-events*: $\bigwedge i. i \in I \implies F \ i \subseteq \text{Pow } (\text{set-pmf } p)$

unfolding *F-def* **by** *blast*

have *as-sigma-sets*:

$\bigwedge i. i \in I \implies \{u. \exists A. u = X \ i \ -' A \ \cap \ \text{set-pmf } p\} = \text{sigma-sets } (\text{set-pmf } p) (F \ i)$

proof –

fix i

assume $a: i \in I$

have $\bigwedge A. X \ i \ -' A \ \cap \ \text{set-pmf } p = X \ i \ -' (A \ \cap \ X \ i \ -' \ \text{set-pmf } p) \ \cap \ \text{set-pmf } p$

by *auto*

hence $\{u. \exists A. u = X \ i \ -' A \ \cap \ \text{set-pmf } p\} =$

$\{X \ i \ -' A \ \cap \ \text{set-pmf } p \mid A. A \subseteq X \ i \ -' \ \text{set-pmf } p\}$

by (*metis (no-types, opaque-lifting) inf-le2*)

also have

$\dots = \{X \ i \ -' A \ \cap \ \text{set-pmf } p \mid A. A \in \text{sigma-sets } (X \ i \ -' \ \text{set-pmf } p) (G \ i)\}$

using a **by** (*simp add:sigma-sets-pow*)

also have $\dots = \text{sigma-sets } (\text{set-pmf } p) \{X \ i \ -' a \ \cap \ \text{set-pmf } p \mid a. a \in G \ i\}$

by (*subst sigma-sets-vimage-commute[symmetric], auto*)

also have $\dots = \text{sigma-sets } (\text{set-pmf } p) (F \ i)$

by (*simp add:F-def*)

finally show

$\{u. \exists A. u = X \ i \ -' A \ \cap \ \text{set-pmf } p\} = \text{sigma-sets } (\text{set-pmf } p) (F \ i)$

by *simp*

qed

have *F-Int-stable*: $\bigwedge i. i \in I \implies \text{Int-stable } (F \ i)$

proof (*rule Int-stableI*)

fix $i \ a \ b$

```

assume  $i \in I$   $a \in F i$   $b \in F i$ 
thus  $a \cap b \in (F i)$ 
  unfolding  $F\text{-def}$   $G\text{-def}$  by (cases  $a \cap b = \{\}$ , auto)
qed

have  $F\text{-indep-sets}:R.\text{indep-sets}$   $F I$ 
proof (rule  $R.\text{indep-sets}I$ )
  fix  $i$ 
  assume  $i \in I$ 
  show  $F i \subseteq R.\text{events}$ 
    unfolding  $F\text{-def}$  events-eq-pow by blast
next
  fix  $A$ 
  fix  $J$ 
  assume  $a:J \subseteq I$   $J \neq \{\}$  finite  $J$   $\forall j \in J. A j \in F j$ 
  have  $b: \bigwedge j. j \in J \implies A j \subseteq \text{set-pmf } p$ 
    by (metis  $\text{PowD}$   $a(1,4)$  subsetD  $F\text{-in-events}$ )
  obtain  $x$  where  $x\text{-def}:\bigwedge j. j \in J \implies A j = X j$   $x j \cap \text{set-pmf } p \wedge x j \in G j$ 
    using  $a$  by (simp add:Pi-def  $F\text{-def}$ , metis)

  show  $R.\text{prob} (\bigcap (A \text{ ' } J)) = (\prod_{j \in J} R.\text{prob} (A j))$ 
  proof (cases  $\exists j \in J. A j = \{\}$ )
    case True
      hence  $\bigcap (A \text{ ' } J) = \{\}$  by blast
      then show ?thesis
        using  $a$  True by (simp, metis measure-empty)
    next
      case False
        then have  $\bigwedge j. j \in J \implies x j \neq \{\}$  using  $x\text{-def}$  by auto
        hence  $\bigwedge j. j \in J \implies x j \in (\lambda x. \{x\}) \text{ ' } X j$   $\text{ ' } \text{set-pmf } p$ 
          using  $x\text{-def}$  by (simp add:G-def)
        then obtain  $y$  where  $y\text{-def}:\bigwedge j. j \in J \implies x j = \{y j\}$ 
          by (simp add:image-def, metis)

        have  $\bigcap (A \text{ ' } J) \subseteq \text{set-pmf } p$  using  $b$   $a(2)$  by blast
        hence  $R.\text{prob} (\bigcap (A \text{ ' } J)) = \text{prob} (\bigcap_{j \in J} A j)$ 
          by (simp add:measure-restrict-space)
        also have  $\dots = \text{prob} (\{\omega. \forall j \in J. X j \omega = y j\})$ 
          using  $a$   $x\text{-def}$   $y\text{-def}$  apply (simp add:vimage-def measure-Int-set-pmf)
          by (rule arg-cong2 [where  $f = \text{measure}$ ], auto)
        also have  $\dots = (\prod_{j \in J} \text{prob} (A j))$ 
          using  $x\text{-def}$   $y\text{-def}$   $a$  assms(2)
          by (simp add:vimage-def measure-Int-set-pmf)
        also have  $\dots = (\prod_{j \in J} R.\text{prob} (A j))$ 
          using  $b$  by (simp add:measure-restrict-space cong:prod.cong)
        finally show ?thesis by blast
  qed
qed

```

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

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

hence $R.indep\text{-}vars$ $(\lambda -. discrete)\ X\ I$
unfolding $R.indep\text{-}vars\text{-}def2$
by $(simp\ add:measurable\text{-}def\ sets\text{-}restrict\text{-}space\ range\text{-}inter)$

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

lemma $(in\ prob\text{-}space)$ $split\text{-}indep\text{-}events$:
assumes $M = measure\text{-}pmf\ p$
assumes $indep\text{-}vars$ $(\lambda i. discrete)\ X'\ I$
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 $indep\text{-}vars$ $(\lambda -. discrete)\ X'\ K$
using $assms(2,3)$ $indep\text{-}vars\text{-}subset$ **by** $blast$
hence $indep\text{-}events$ $(\lambda x. \{\omega \in space\ M. P\ x\ (X'\ x\ \omega)\})\ K$
using $indep\text{-}eventsI\text{-}indep\text{-}vars$ **by** $force$
hence $a:indep\text{-}events$ $(\lambda x. \{\omega. P\ x\ (X'\ x\ \omega)\})\ K$
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 $(simp\ add: measure\text{-}pmf\text{-}eq[OF\ assms(1)])$
also have $\dots = (\prod x \in K. prob\ \{\omega. P\ x\ (X'\ x\ \omega)\})$
using $a\ assms(4)$ **by** $(cases\ K = \{\}, auto\ simp: indep\text{-}events\text{-}def)$
finally show $?thesis$ **by** $simp$
qed

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

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

```

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

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

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

  have  $\text{distr } M \ \text{discrete } X = \text{map-pmf } X \ p$ 
    by (simp add:  $\text{map-pmf-rep-eq } \text{assms}(1)$ )
  also have  $\dots = \text{measure-pmf } (\text{pmf-of-set } A)$ 
    using  $b$  by simp
  also have  $\dots = \text{uniform-measure } \text{discrete } A$ 
    by (rule  $\text{pmf-of-set-eq-uniform}[OF \ \text{assms}(2,3)]$ )
  finally have  $\text{distr } M \ \text{discrete } X = \text{uniform-measure } \text{discrete } A$ 
    by simp
  moreover have  $\text{random-variable } \text{discrete } X$ 
    by (simp add:  $\text{assms}(1)$ )
  ultimately show ?thesis using  $\text{assms}(2,3)$ 
    by (simp add:  $\text{uniform-on-def}$ )
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 $1 / \text{real}(\text{card}(\text{carrier } R))^s$, 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* $x \ \omega = \text{eval } \omega \ x$

locale *poly-hash-family* = *ring* +
fixes $k :: \text{nat}$
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** $M = \text{measure-pmf}(\text{pmf-of-set } \text{space})$

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* $\neq \{\}$
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* $R \neq \{\}$
by (*simp* *add:carrier-not-empty*)

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

lemma *hash-range*[*simp*]:
assumes $\omega \in \text{space}$
assumes $x \in \text{carrier } R$
shows *hash* $x \ \omega \in \text{carrier } R$

```

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  $\omega \in \text{space}$ 
  shows  $(\lambda x. \text{hash } x \ \omega) \text{ 'carrier } R \subseteq \text{carrier } R$ 
  using hash-range assms by auto

lemma integrable-M[simp]:
  fixes  $f :: 'a \text{ list} \Rightarrow 'c::\{\text{banach, second-countable-topology}\}$ 
  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  $\equiv \text{card } (\text{carrier } R)$ 

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

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

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

```

hence $a:\text{field-size} > 0$
using *finite-carrier card-gt-0-iff* **by** *blast*

have $b:\text{real} (\text{card} \{\omega \in \text{space}. \forall x \in K. \text{eval } \omega \ x = y \ x\}) / \text{real} (\text{card} \ \text{space}) =$
 $1 / \text{real} \ \text{field-size} \wedge \text{card} \ K$
using $a \ \text{assms}(2)$
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 $x \in \text{carrier } R \ y \in \text{carrier } R$
shows $\text{prob} \ \{\omega. \text{hash } x \ \omega = y\} = 1 / (\text{real} \ \text{field-size})$
using *hash-prob[where $K=\{x\}$] assms finite-carrier k-ge-0* **by** *simp*

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

qed

lemma *indep*:
assumes $J \subseteq \text{carrier } R$
assumes $\text{card } J \leq k$
shows *indep-vars* ($\lambda\cdot$. *discrete*) *hash* J
proof –
have $0 \in \text{carrier } R$ **by** *simp*
hence $\text{card-}R\text{-ge-0:field-size} > 0$
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 $\omega \in$ *space*
assumes *degree* $\omega = 1$
shows *inj-on* (λx . *hash* $x \omega$) (*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 $i \in$ *carrier* R
shows *uniform-on* (*hash* i) (*carrier* R)
proof –
have a :
 $\wedge a$. *prob* $\{\omega$. *hash* $i \omega \in \{a\}\} =$ *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* \wedge *finite (carrier F)*
assumes $1 \leq n$
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* \wedge *finite (carrier F)*
assumes $x \in \text{carrier } F$
assumes $1 \leq n$
assumes $y \in \text{carrier } F$
shows
 $\mathcal{P}(\omega \text{ in } \text{pmf-of-set (bounded-degree-polynomials F n)}. \text{ring.hash F } x \ \omega = y)$
 $= 1 / (\text{real (card (carrier F))})$
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* \equiv *Pi-pmf I undefined M*

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

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

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

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

lemma *PiE-default-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 $\text{set-pmf } (\text{prod-pmf } I M) = \text{PiE } I (\text{set-pmf } \circ M)$
by (*simp add: set-Pi-pmf[OF assms] PiE-default-undefined-eq*)

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

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

have $a: \text{Pi } J \ A = \text{Pi } I \ (\lambda i. \text{if } i \in J \text{ then } A \ i \text{ else } \text{UNIV})$
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 $i \in I$
shows $\text{measure } (\text{measure-pmf } (\text{prod-pmf } I \ M)) \ \{\omega. P \ (\omega \ i)\} = \text{measure } (M \ i)$
 $\{\omega. P \ \omega\}$
using *prob-prod-pmf'[OF assms(1), where J={i} and M=M and A= $\lambda \cdot$. Collect P]*
by (*simp add:assms Pi-def*)

definition *restrict-dfl* **where** $\text{restrict-dfl } f \ A \ d = (\lambda x. \text{if } x \in A \text{ then } f \ x \text{ else } d)$

lemma *pi-pmf-decompose*:
assumes *finite I*
shows $\text{Pi-pmf } I \ d \ M = \text{map-pmf } (\lambda \omega. \text{restrict-dfl } (\lambda i. \omega \ (f \ i) \ i) \ I \ d) \ (\text{Pi-pmf } (f \ ' \ I) \ (\lambda \cdot. d) \ (\lambda j. \text{Pi-pmf } (f \ -' \ \{j\} \cap I) \ d \ M))$
proof –
have *fin-F-I:finite (f ' I)* **using** *assms* **by** *blast*

have *finite I \implies ?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 $a: (f \ -' \ \{x\} \cap I) \cup (f \ -' \ F \cap I) = I$
using *insert(4)* **by** *blast*
have $b: F = f \ ' \ (f \ -' \ F \cap I)$ **using** *insert(2,4)*
by (*auto simp add:set-eq-iff image-def vimage-def*)
have $c: \text{finite } (f \ -' \ F \cap I)$ **using** *insert* **by** *blast*
have $d: \bigwedge j. j \in F \implies (f \ -' \ \{j\} \cap (f \ -' \ F \cap I)) = (f \ -' \ \{j\} \cap I)$
using *insert(4)* **by** *blast*

have $\text{Pi-pmf } I \ d \ M = \text{Pi-pmf } ((f \ -' \ \{x\} \cap I) \cup (f \ -' \ F \cap I)) \ d \ M$
by (*simp add:a*)
also have $\dots = \text{map-pmf } (\lambda(g, h) \ i. \text{if } i \in f \ -' \ \{x\} \cap I \text{ then } g \ i \text{ else } h \ i)$
 $(\text{pair-pmf } (\text{Pi-pmf } (f \ -' \ \{x\} \cap I) \ d \ M) \ (\text{Pi-pmf } (f \ -' \ F \cap I) \ d \ M))$
using *insert* **by** (*subst Pi-pmf-union*) *auto*
also have $\dots = \text{map-pmf } (\lambda(g, h) \ i. \text{if } f \ i = x \wedge i \in I \text{ then } g \ i \text{ else if } f \ i \in F \wedge i \in I \text{ then } h \ (f \ i) \ i \text{ else } d)$
 $(\text{pair-pmf } (\text{Pi-pmf } (f \ -' \ \{x\} \cap I) \ d \ M) \ (\text{Pi-pmf } F \ (\lambda \cdot. d) \ (\lambda j. \text{Pi-pmf } (f \ -'$

$\{j\} \cap (f - ' F \cap I)) d M))$
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* ($\lambda(g,h) i. \text{if } i \in I \text{ then } (h(x := g)) (f i) i \text{ else } d$)
(pair-pmf (Pi-pmf (f - '{x} \cap I) d M) (Pi-pmf F (\lambda-. d) (\lambda j. Pi-pmf (f - '{j} \cap 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* ($\lambda \omega i. \text{if } i \in I \text{ then } \omega (f i) i \text{ else } d$) *(Pi-pmf (insert x F) (\lambda-. d) (\lambda j. Pi-pmf (f - '{j} \cap 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 \cap 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) (\lambda-. discrete) (\lambda i \omega. restrict-dfl \omega (f - '{i} \cap I) d) (f ' I)*
proof –
let ?*Q* = *(Pi-pmf (f ' I) (\lambda-. d) (\lambda i. Pi-pmf (I \cap f - '{i} \cap I) d M))*
have *a:prob-space.indep-vars ?Q (\lambda-. discrete) (\lambda i \omega. \omega i) (f ' I)*
using *assms* **by** (*intro indep-vars-Pi-pmf, blast*)
have *b: AE x in measure-pmf ?Q. \forall i \in f ' I. x i = restrict-dfl (\lambda i. x (f i) i) (I \cap f - '{i} \cap 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 (\lambda-. discrete) (\lambda i x. restrict-dfl (\lambda i. x (f i) i) (I \cap f - '{i} \cap 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 $\bigwedge i \omega. i \in J \implies X' i \omega = X' i (restrict-dfl \omega (f - '{i} \cap I) d)$
assumes $J = f ' I$

assumes $\bigwedge \omega \ i. \ i \in J \implies X' \ i \ \omega \in \text{space } (M' \ i)$
shows *prob-space.indep-vars* (*measure-pmf* (*Pi-pmf* *I d p*)) $M' (\lambda i \ \omega. \ X' \ i \ \omega) \ J$
proof –
define *M* **where** $M \equiv \text{measure-pmf } (Pi\text{-pmf } I \ d \ p)$
interpret *prob-space* *M*
using *M-def prob-space-measure-pmf* **by** *blast*
have *indep-vars* ($\lambda \cdot. \ \text{discrete}$) ($\lambda i \ x. \ \text{restrict-dfl } x \ (f \ -' \ \{i\} \cap I) \ d$) ($f \ ' \ I$)
unfolding *M-def* **by** (*rule indep-vars-restrict* [*OF assms(1)*])
hence *indep-vars* $M' (\lambda i \ \omega. \ X' \ i \ (\text{restrict-dfl } \omega \ (f \ -' \ \{i\} \cap I) \ d)) \ (f \ ' \ I)$
using *assms(4)*
by (*intro indep-vars-compose2* [**where** $Y=X'$ **and** $N=M'$ **and** $M'=\lambda \cdot. \ \text{discrete}$])
(*auto simp:assms(3)*)
hence *indep-vars* $M' (\lambda i \ \omega. \ X' \ i \ \omega) \ (f \ ' \ I)$
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 $f :: 'b \Rightarrow ('c :: \{\text{second-countable-topology, banach, real-normed-field}\})$
assumes *finite* *I*
assumes $i \in I$
assumes *integrable* (*measure-pmf* (*M i*)) *f*
shows *integrable-Pi-pmf-slice*: *integrable* (*Pi-pmf* *I d M*) ($\lambda x. \ f \ (x \ i)$)
and *expectation-Pi-pmf-slice*: *integral^L* (*Pi-pmf* *I d M*) ($\lambda x. \ f \ (x \ i)$) = *integral^L*
(*M i*) *f*
proof –
have $a: \text{distr } (Pi\text{-pmf } I \ d \ M) \ (M \ i) \ (\lambda \omega. \ \omega \ i) = \text{distr } (Pi\text{-pmf } I \ d \ M) \ \text{discrete}$
($\lambda \omega. \ \omega \ i$)
by (*rule distr-cong, auto*)

have $b: \text{measure-pmf.random-variable } (M \ i) \ \text{borel } f$
using *assms(3)* **by** *simp*

have $c: \text{integrable } (\text{distr } (Pi\text{-pmf } I \ d \ M) \ (M \ i) \ (\lambda \omega. \ \omega \ i)) \ f$
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*) ($\lambda x. \ f \ (x \ i)$)
by (*rule integrable-distr* [**where** $f=f$ **and** $M'=M \ i$]) (*auto intro: c*)

have *integral^L* (*Pi-pmf* *I d M*) ($\lambda x. \ f \ (x \ i)$) = *integral^L* (*distr* (*Pi-pmf* *I d M*)
(*M i*) ($\lambda \omega. \ \omega \ i$)) *f*
using *b* **by** (*intro integral-distr* [*symmetric*], *auto*)
also have $\dots = \text{integral^L} \ (\text{map-pmf } (\lambda \omega. \ \omega \ i) \ (Pi\text{-pmf } I \ d \ M)) \ f$
by (*subst a, subst map-pmf-rep-eq* [*symmetric*], *simp*)
also have $\dots = \text{integral^L} \ (M \ i) \ f$
using *assms(1,2)* **by** (*simp add: Pi-pmf-component*)

finally show $\text{integral}^L (Pi\text{-pmf } I \text{ d } M) (\lambda x. f (x \ i)) = \text{integral}^L (M \ i) f$ **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 $f :: 'a \Rightarrow 'b \Rightarrow ('c :: \{\text{second-countable-topology, banach, real-normed-field}\})$

assumes *finite I*

assumes $\bigwedge i. i \in I \implies \text{integrable (measure-pmf (M i)) (f i)}$

shows $\text{integral}^L (Pi\text{-pmf } I \text{ d } M) (\lambda x. (\prod i \in I. f \ i (x \ i))) = (\prod i \in I. \text{integral}^L (M \ i) (f \ i))$

proof –

have $a: \text{prob-space.indep-vars (measure-pmf (Pi-pmf } I \text{ d } M)) (\lambda-. \text{ borel}) (\lambda i \ \omega. f \ i (\omega \ i)) \ I$

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 $\text{integral}^L (Pi\text{-pmf } I \text{ d } M) (\lambda x. (\prod i \in I. f \ i (x \ i))) = (\prod i \in I. \text{integral}^L (Pi\text{-pmf } I \text{ d } M) (\lambda x. f \ i (x \ i)))$

by (*intro prob-space.indep-vars-lebesgue-integral prob-space-measure-pmf assms(1,2) a integrable-Pi-pmf-slice*) *auto*

also have $\dots = (\prod i \in I. \text{integral}^L (M \ i) (f \ i))$

by (*intro prod.cong expectation-Pi-pmf-slice assms(1,2)*) *auto*

finally show *?thesis* **by simp**

qed

lemma *variance-prod-pmf-slice*:

fixes $f :: 'a \Rightarrow \text{real}$

assumes $i \in I$ *finite I*

assumes $\text{integrable (measure-pmf (M i)) (\lambda \omega. f \ \omega \ \mathcal{Q})}$

shows $\text{prob-space.variance (Pi-pmf } I \text{ d } M) (\lambda \omega. f (\omega \ i)) = \text{prob-space.variance (M i) f}$

proof –

have $a: \text{integrable (measure-pmf (M i)) f}$

using *assms(3) measure-pmf.square-integrable-imp-integrable* **by** *auto*

have $b: \text{integrable (measure-pmf (Pi-pmf } I \text{ d } M)) (\lambda x. (f (x \ i))^2)$

by (*rule integrable-Pi-pmf-slice[OF assms(2) assms(1)],metis assms(3)*)

have $c: \text{integrable (measure-pmf (Pi-pmf } I \text{ d } M)) (\lambda x. (f (x \ i)))$

by (*rule integrable-Pi-pmf-slice[OF assms(2) assms(1)],metis a*)

have $\text{measure-pmf.expectation (Pi-pmf } I \text{ d } M) (\lambda x. (f (x \ i))^2) - (\text{measure-pmf.expectation (Pi-pmf } I \text{ d } M) (\lambda x. f (x \ i)))^2 =$

$\text{measure-pmf.expectation (M i) (\lambda x. (f x)^2) - (\text{measure-pmf.expectation (M i) f})^2$

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 $Pi\text{-pmf } I d (\lambda i. M i \gg (\lambda x. \text{return-pmf } (f i x))) = Pi\text{-pmf } I d' M \gg (\lambda x. \text{return-pmf } (\lambda i. \text{if } i \in I \text{ then } f i (x i) \text{ else } d))$
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 $\text{pmf-of-set } (A \times B) = \text{pair-pmf } (\text{pmf-of-set } A) (\text{pmf-of-set } B)$
proof –
have $\text{indicat-real } (A \times B) (i, j) = \text{indicat-real } A i * \text{indicat-real } B j$ **for** $i j$
by (*cases* $i \in A$; *cases* $j \in B$) *auto*
hence $\text{pmf } (\text{pmf-of-set } (A \times B)) (i, j) = \text{pmf } (\text{pair-pmf } (\text{pmf-of-set } A) (\text{pmf-of-set } B)) (i, j)$
for $i j$ **using** *assms* **by** (*simp add: pmf-pair*)
thus *?thesis*
by (*intro pmf-eqI*) *auto*
qed

lemma *split-pmf-mod-div'*:
assumes $a > (0::\text{nat})$
assumes $b > 0$
shows $\text{map-pmf } (\lambda x. (x \bmod a, x \text{ div } a)) (\text{pmf-of-set } \{..<a * b\}) = \text{pmf-of-set } (\{..<a\} \times \{..<b\})$
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 $a > (0::\text{nat})$
assumes $b > 0$
shows $\text{map-pmf } (\lambda x. (x \bmod a, x \text{ div } a)) (\text{pmf-of-set } \{..<a * b\}) = \text{pair-pmf } (\text{pmf-of-set } \{..<a\}) (\text{pmf-of-set } \{..<b\})$
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  $S = \text{pro-last } S + 1$ 
```

```
definition sample-pro where sample-pro  $S = \text{map-pmf } (\text{pro-select } S) (\text{pmf-of-set } \{0.. \text{pro-last } S\})$ 
```

```
declare [[coercion sample-pro]]
```

```
abbreviation pro-set where pro-set  $S \equiv \text{set-pmf } (\text{sample-pro } S)$ 
```

```
lemma sample-pro-alt: sample-pro  $S = \text{map-pmf } (\text{pro-select } S) (\text{pmf-of-set } \{.. < \text{pro-size } S\})$ 
```

```
  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  $S = \text{pro-select } S \text{ ' } \{.. < \text{pro-size } S\}$ 
  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  $\{.. < \text{pro-size } S\}) = \{.. < \text{pro-size } S\}$ 
  using pro-size-gt-0 by (intro set-pmf-of-set) auto
```

```
lemma pro-select-in-set: pro-select  $S (x \bmod \text{pro-size } S) \in \text{pro-set } S$ 
  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  $f :: 'a \Rightarrow 'c::\{\text{banach, second-countable-topology}\}$ 
  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  $ls = (\text{pro-last} = \text{length } ls - 1, \text{pro-select} = (!) ls)$ 
```

lemma *list-pro*:
assumes $xs \neq []$
shows *sample-pro* (*list-pro* xs) = *pmf-of-multiset* (*mset* xs) (**is** $?L = ?R$)
proof –
have $?L = \text{map-pmf } (!) \text{ } xs \text{ } (\text{pmf-of-set } \{..<\text{length } xs\})$
using *assms* **unfolding** *list-pro-def* *sample-pro-alt* *pro-size-def* **by** *simp*
also have $\dots = \text{pmf-of-multiset } (\text{image-mset } (!) \text{ } xs) \text{ } (\text{mset-set } \{..<\text{length } xs\})$
using *assms* **by** (*subst* *map-pmf-of-set*) *auto*
also have $\dots = ?R$
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 $?L = \text{map-pmf } (!) \text{ } xs \text{ } (\text{pmf-of-set } \{..<\text{length } xs\})$
using *assms* **unfolding** *list-pro-def* *sample-pro-alt* *pro-size-def* **by** *simp*
also have $\dots = \text{pmf-of-set } (!) \text{ } xs \text{ } \{..<\text{length } xs\}$
using *assms* *nth-eq-iff-index-eq* **by** (*intro* *map-pmf-of-set-inj* *inj-onI*) *auto*
also have $\dots = ?R$
by (*intro* *arg-cong*[**where** $f = \text{pmf-of-set}$]) (*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 $(!) \text{ } xs \text{ } \{..<\text{length } xs\} = \text{set } xs$ **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* :: $nat \Rightarrow nat$ *pseudorandom-object* **where**
 $\text{nat-pro } n = (\text{pro-last} = n-1, \text{pro-select} = \text{id})$

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* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$ **where**
count-zeros 0 k = 0 |
count-zeros (*Suc* n) k = (*if odd* k *then* 0 *else* 1 + *count-zeros* n ($k \text{ div } 2$))

lemma *count-zeros-iff*: $j \leq n \implies \text{count-zeros } n \ k \geq j \iff 2^j \text{ dvd } k$
proof (*induction* j *arbitrary: n k*)
case 0
then show ?*case* **by** *simp*
next
case (*Suc* j)
then obtain n' **where** *n-def*: $n = \text{Suc } n'$ **using** *Suc-le-D* **by** *presburger*
show ?*case* **using** *Suc* **unfolding** *n-def* **by** *auto*
qed

lemma *count-zeros-max*:
count-zeros n $k \leq n$
by (*induction* n *arbitrary: k*) *auto*

definition *geom-pro* :: $\text{nat} \Rightarrow \text{nat}$ *pseudorandom-object* **where**
geom-pro $n = (\text{pro-last} = 2^{\widehat{n}} - 1, \text{pro-select} = \text{count-zeros } n)$

lemma *geom-pro-size*: *pro-size* (*geom-pro* n) = $2^{\widehat{n}}$
unfolding *geom-pro-def* *pro-size-def* **by** *simp*

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

lemma *geom-pro-prob*:
measure (*sample-pro* (*geom-pro* n)) $\{\omega. \omega \geq j\} = \text{of-bool } (j \leq n) / 2^{\widehat{j}}$ (**is** ?*L* = ?*R*)
proof (*cases* $j \leq n$)
case *True*
have $a: \{..<(2^{\widehat{n}})::\text{nat}\} \neq \{\}$
by (*simp add: lessThan-empty-iff*)
have $b: \text{finite } \{..<(2^{\widehat{n}})::\text{nat}\}$ **by** *simp*

define $f :: \text{nat} \Rightarrow \text{nat}$ **where** $f = (\lambda x. x * 2^{\wedge}j)$
have $d:\text{inj-on } f \{..<2^{\wedge}(n-j)\}$ **unfolding** $f\text{-def}$ **by** $(\text{intro inj-onI}) \text{ simp}$

have $e:2^{\wedge}j > (0::\text{nat})$ **by** simp

have $y \in f \text{ ' } \{..<2^{\wedge}(n-j)\} \longleftrightarrow y \in \{x. x < 2^{\wedge}n \wedge 2^{\wedge}j \text{ dvd } x\}$ **for** $y :: \text{nat}$
proof –
have $y \in f \text{ ' } \{..<2^{\wedge}(n-j)\} \longleftrightarrow (\exists x. x < 2^{\wedge}(n-j) \wedge y = 2^{\wedge}j * x)$
unfolding $f\text{-def}$ **by** auto
also have $\dots \longleftrightarrow (\exists x. 2^{\wedge}j * x < 2^{\wedge}j * 2^{\wedge}(n-j) \wedge y = 2^{\wedge}j * x)$
using e **by** simp
also have $\dots \longleftrightarrow (\exists x. 2^{\wedge}j * x < 2^{\wedge}n \wedge y = 2^{\wedge}j * x)$
using True **by** $(\text{subst power-add[symmetric]}) \text{ simp}$
also have $\dots \longleftrightarrow (\exists x. y < 2^{\wedge}n \wedge y = x * 2^{\wedge}j)$
by $(\text{metis Groups.mult-ac}(2))$
also have $\dots \longleftrightarrow y \in \{x. x < 2^{\wedge}n \wedge 2^{\wedge}j \text{ dvd } x\}$ **by** auto
finally show $?thesis$ **by** simp
qed

hence $c:f \text{ ' } \{..<2^{\wedge}(n-j)\} = \{x. x < 2^{\wedge}n \wedge 2^{\wedge}j \text{ dvd } x\}$ **by** auto

have $?L = \text{measure } (\text{pmf-of-set } \{..<2^{\wedge}n\}) \{ \omega. \text{count-zeros } n \ \omega \geq j \}$
unfolding $\text{sample-pro-alt geom-pro-size}$ **by** $(\text{simp add:geom-pro-def})$
also have $\dots = \text{real } (\text{card } \{x::\text{nat}. x < 2^{\wedge}n \wedge 2^{\wedge}j \text{ dvd } x\}) / 2^{\wedge}n$
by $(\text{simp add:measure-pmf-of-set[OF a b] count-zeros-iff[OF True]})$
 $(\text{simp add:lessThan-def Collect-conj-eq})$
also have $\dots = \text{real } (\text{card } (f \text{ ' } \{..<2^{\wedge}(n-j)\})) / 2^{\wedge}n$
by (simp add:c)
also have $\dots = \text{real } (\text{card } (\{..<(2^{\wedge}(n-j))::\text{nat}\})) / 2^{\wedge}n$
by $(\text{simp add:card-image[OF d]})$
also have $\dots = ?R$
using True **by** $(\text{simp add:frac-eq-eq power-add[symmetric]})$
finally show $?thesis$ **by** simp

next

case False
have $\text{set-pmf } (\text{sample-pro } (\text{geom-pro } n)) \subseteq \{..n\}$
using geom-pro-range **by** simp
hence $?L = \text{measure } (\text{sample-pro } (\text{geom-pro } n)) \{ \}$
using False **by** $(\text{intro measure-pmf-cong}) \text{ auto}$
also have $\dots = ?R$
using False **by** simp
finally show $?thesis$
by simp

qed

lemma $\text{geom-pro-prob-single}$:

$\text{measure } (\text{sample-pro } (\text{geom-pro } n)) \{j\} \leq 1 / 2^{\wedge}j$ **(is** $?L \leq ?R)$

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)) { $\omega. \omega \geq j$ } – measure (sample-pro
(geom-pro n)) { $\omega. \omega \geq (j+1)$ }
  by (intro arg-cong2[where  $f=(-)$ ] measure-pmf-cong) auto
also have ... = of-bool ( $j \leq n$ ) *  $1 / 2^j$  – of-bool ( $j + 1 \leq n$ ) /  $2^{j+1}$ 
  unfolding geom-pro-prob by simp
also have ...  $\leq 1/2^j - 0$ 
  by (intro diff-mono) auto
also have ... = ?R by simp
finally show ?thesis by simp
qed

```

```

definition prod-pro ::
  'a pseudorandom-object  $\Rightarrow$  'b pseudorandom-object  $\Rightarrow$  ('a  $\times$  'b) pseudorandom-object
where
  prod-pro P Q =
    ( $\lfloor$  pro-last = pro-size P * pro-size Q – 1,
     pro-select = ( $\lambda k. (\text{pro-select } P (k \bmod \text{pro-size } P), \text{pro-select } Q (k \text{ div } \text{pro-size } P))$ )
     $\rfloor$ )

```

```

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 ( $\lambda k. (\text{pro-select } P (k \bmod ?p), \text{pro-select } Q (k \text{ div } ?p))$ )
(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 ( $\lambda k. (k \bmod ?p, k \text{ div } ?p)$ ) (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) *Natural-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
 $\Rightarrow (nat \Rightarrow 'c) pseudorandom-object$

where *hash-space' R k S = (*

(

pro-last = idx-size R $\hat{k}-1$,

pro-select = ($\lambda x i.$

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)) { $\omega.$ ring.hash F x $\omega = y$ }* =

of-bool (y ∈ carrier F) / (real (card (carrier F))) (is ?L = ?R)

proof (*cases y ∈ carrier F*)

case *True*

have *?L = $\mathcal{P}(\omega$ in pmf-of-set (bounded-degree-polynomials F n). ring.hash F x $\omega = y$)* **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 enumC R fieldC R
  assumes pro-size S dvd order (ring-of R)
  assumes I ⊆ {..order (ring-of R)} card I ≤ k
  shows map-pmf (λf. (λi∈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 = λx i. poly-eval R (poly-enum R k x) (idx-enum R i)
  let ?f = λx. 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 fieldC-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 ... ≤ 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 ^ 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) ≤ 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) ∘ idx-enum R) (λx ω. 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) (λx ω. eval ω (idx-enum R
x)) I
by (simp add:comp-def)

have 0: pmf (map-pmf (λx. λi∈I. eval x (idx-enum R i)) ?T) ω = pmf (prod-pmf
I (λ-. ?R-pmf)) ω
(is ?L1 = ?R1) for ω
proof (cases ω ∈ extensional I)
case True
have ?L1 = measure ?T {x. (λi∈I. eval x (idx-enum R i)) = ω}
by (simp add:pmf-map vimage-def)
also have ... = measure ?T {x. (∀i∈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 ... = (∏ i∈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 ... = (∏ i∈I. measure ?T {x. hash (idx-enum R i) x = ω i})
unfolding hash-def by simp

```

```

also have ... = ( $\prod_{i \in I} \text{of\_bool}(\omega \ i \in \text{carrier}(\text{ring-of } R)) / \text{real}(\text{card}(\text{carrier}(\text{ring-of } R))))$ )
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 ... = ( $\prod_{i \in I} \text{pmf}(\text{pmf-of-set}(\text{carrier}(\text{ring-of } R))) (\omega \ i)$ )
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 ( $\lambda x. \lambda i \in I. \text{poly-eval } R \ x \ (\text{idx-enum } R \ i)$ ) (map-pmf (poly-enum R k)
(pmf-of-set  $\{..<?b^k\}$ ))
by (simp add:map-pmf-comp)
also have ... = map-pmf ( $\lambda x. \lambda i \in I. \text{poly-eval } R \ x \ (\text{idx-enum } R \ i)$ ) (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 ( $\lambda x. \lambda i \in I. \text{poly-eval } R \ x \ (\text{idx-enum } R \ i)$ ) ?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 ( $\lambda x. \lambda i \in I. \text{eval } x \ (\text{idx-enum } R \ i)$ ) ?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 ( $\lambda \cdot. ?R\text{-pmf}$ ) (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 ( $\lambda \cdot. ?R\text{-pmf}$ )
by simp

have 1: map-pmf ( $\lambda x. x \ \text{mod } ?s$ ) (pmf-of-set  $\{..<?b\}$ ) = pmf-of-set  $\{..<?s\}$  (is
?L1 = ?R1)
proof –
have ?L1 = map-pmf fst (map-pmf ( $\lambda x. (x \ \text{mod } ?s, x \ \text{div } ?s)$ ) (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** $f = \text{map-pmf}$] *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** $f = \text{map-pmf}$] *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 ($\lambda x. \text{pro-select } S (x \text{ mod } ?s)$) (map-pmf*
(idx-enum-inv R) ?R-pmf)
by (*simp add:map-pmf-comp*)
also have ... = *map-pmf ($\lambda x. \text{pro-select } S (x \text{ mod } ?s)$) (pmf-of-set {.. $?b$ })*
using *enum-cD(1,2,4)[OF assms(1)] carrier-not-empty*
by (*intro arg-cong2*[**where** $f = \text{map-pmf}$] *refl map-pmf-of-set-bij-betw*) *auto*
also have ... = *map-pmf (pro-select S) (map-pmf ($\lambda x. x \text{ mod } ?s$) (pmf-of-set*
{.. $?b$ }))
by (*simp add:map-pmf-comp*)
also have ... = *sample-pro S unfolding sample-pro-alt 1* **by** *simp*
finally have $2 : \text{map-pmf } ?f ?R\text{-pmf} = \text{sample-pro } S$ **by** *simp*

have $?L = \text{map-pmf } (\lambda x. \lambda i \in I. ?f (?g x i)) (\text{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 ($\lambda f. \lambda i \in I. ?f (f i)$) (map-pmf ($\lambda x. \lambda i \in I. ?g x i$)*
(pmf-of-set {.. $?b^k$ }))
unfolding *map-pmf-comp* **by** (*intro arg-cong2*[**where** $f = \text{map-pmf}$] *refl re-*
strict-ext ext) *simp*
also have ... = *prod-pmf I ($\lambda \cdot. \text{map-pmf } ?f (\text{pmf-of-set } (\text{carrier } (\text{ring-of } R))))$*
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 $?L = \text{map-pmf } (\lambda f i. \text{undefined}) (\text{sample-pro } (\text{hash-space}' R k S))$

using *True* **by** (*intro map-pmf-cong refl*) *auto*

also have ... = *return-pmf ($\lambda f. \text{undefined}$)* **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 \in pro-set S

unfolding *hash-space'-def* **by** (*simp add: pro-select-in-set*)

definition *hash-pro* ::

$\text{nat} \Rightarrow \text{nat} \Rightarrow ('a, 'b) \text{ pseudorandom-object-scheme} \Rightarrow (\text{nat} \Rightarrow 'a) \text{ pseudorandom-object}$

where $\text{hash-pro } k \ d \ S = (
\text{let } (p, j) = \text{split-power } (\text{pro-size } S);
l = \max j \ (\text{floorlog } p \ (d-1))
\text{in } \text{hash-space}' \ (GF \ (p^\wedge l)) \ k \ S)$

definition $\text{hash-pro-spmf} ::$

$\text{nat} \Rightarrow \text{nat} \Rightarrow ('a, 'b) \text{ pseudorandom-object-scheme} \Rightarrow (\text{nat} \Rightarrow 'a) \text{ pseudorandom-object spmf}$

where $\text{hash-pro-spmf } k \ d \ S =
\text{do } \{
\text{let } (p, j) = \text{split-power } (\text{pro-size } S);
\text{let } l = \max j \ (\text{floorlog } p \ (d-1));
R \leftarrow GF_R \ (p^\wedge l);
\text{return-spmf } (\text{hash-space}' \ R \ k \ S)
\}$

definition $\text{hash-pro-pmf} ::$

$\text{nat} \Rightarrow \text{nat} \Rightarrow ('a, 'b) \text{ pseudorandom-object-scheme} \Rightarrow (\text{nat} \Rightarrow 'a) \text{ pseudorandom-object pmf}$

where $\text{hash-pro-pmf } k \ d \ S = \text{map-pmf the } (\text{hash-pro-spmf } k \ d \ S)$

syntax

$\text{-FLIPBIND} \quad :: ('a \Rightarrow 'b) \Rightarrow 'c \Rightarrow 'b \ (\text{infixr } =\langle\langle 54)$

translations

$\text{-FLIPBIND } f \ g \ \Rightarrow\> \ g \ \gg\> \ f$

context

fixes S

fixes $d :: \text{nat}$

fixes $k :: \text{nat}$

assumes $\text{size-prime-power: is-prime-power } (\text{pro-size } S)$

begin

private definition p **where** $p = \text{fst } (\text{split-power } (\text{pro-size } S))$

private definition j **where** $j = \text{snd } (\text{split-power } (\text{pro-size } S))$

private definition l **where** $l = \max j \ (\text{floorlog } p \ (d-1))$

private lemma $\text{split-power: } (p, j) = \text{split-power } (\text{pro-size } S)$

using $p\text{-def } j\text{-def}$ **by** auto

private lemma $\text{hash-sample-space-alt: hash-pro } k \ d \ S = \text{hash-space}' \ (GF \ (p^\wedge l)) \ k \ S$

unfolding $\text{hash-pro-def split-power[symmetric]}$ **by** $(\text{simp add:j-def l-def Let-def})$

private lemma $p\text{-prime} : \text{prime } p$ **and** $j\text{-gt-0: } j > 0$

proof $-$

obtain $q\ r$ **where** $0:\text{pro-size } S = q \hat{\ } r$ **and** $q\text{-prime}$: prime q **and** $r\text{-gt-0}$: $r > 0$
using size-prime-power $\text{is-prime-power-def}$ **by** blast

have $(p,j) = \text{split-power } (q \hat{\ } r)$ **unfolding** $\text{split-power } 0$ **by** simp
also have $\dots = (q,r)$ **by** $(\text{intro } \text{split-power-prime } q\text{-prime } r\text{-gt-0})$
finally have $(p,j) = (q,r)$ **by** simp
thus prime p $j > 0$ **using** $q\text{-prime } r\text{-gt-0}$ **by** auto
qed

private lemma $l\text{-gt-0}$: $l > 0$
unfolding $l\text{-def}$ **using** $j\text{-gt-0}$ **by** simp

private lemma prime-power : $\text{is-prime-power } (p \hat{\ } l)$
using $p\text{-prime } l\text{-gt-0}$ **unfolding** $\text{is-prime-power-def}$ **by** auto

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

lemma $\text{lossless-hash-pro-spmf}$: $\text{lossless-spmf } (\text{hash-pro-spmf } k\ d\ S)$
proof –
have $\text{lossless-spmf } (\text{GF}_R (p \hat{\ } l))$ **by** $(\text{intro } \text{galois-field-random-1 } \text{prime-power})$
thus $?thesis$ **unfolding** hash-pro-spmf-def $\text{split-power}[\text{symmetric}]$ $l\text{-def}$ **by** simp
qed

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

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

lemma $\text{hash-pro-in-hash-pro-pmf}$: $\text{hash-pro } k\ d\ S \in \text{set-pmf } (\text{hash-pro-pmf } k\ d\ S)$
unfolding $\text{hashp-eq-hash-pro-spmf}$ **by** $(\text{intro } \text{hash-in-hash-pro-spmf})$

lemma $\text{hash-pro-spmf-distr}$:
assumes $s \in \text{set-spmf } (\text{hash-pro-spmf } k\ d\ S)$
assumes $I \subseteq \{..<d\}$ $\text{card } I \leq k$
shows $\text{map-pmf } (\lambda f. (\lambda i \in I. f\ i)) (\text{sample-pro } s) = \text{prod-pmf } I (\lambda \cdot. \text{sample-pro } S)$
proof –
have $(d-1) < p \hat{\ } \text{floorlog } p\ (d-1)$
using $\text{floorlog-leD } \text{prime-gt-1-nat}[OF\ p\text{-prime}]$ **by** simp
hence $d \leq p \hat{\ } \text{floorlog } p\ (d-1)$ **by** $(\text{cases } d)$ auto
also have $\dots \leq p \hat{\ } l$

using *prime-gt-0-nat*[*OF p-prime*] **unfolding** *l-def* **by** (*intro power-increasing*)
auto
finally have $0: d \leq p \wedge l$ **by** *simp*

obtain R **where** *R-in*: $R \in \text{set-spmf } (GF_R (p \wedge l))$ **and** *s-def*: $s = \text{hash-space}' R$
 $k S$

using *assms(1)* **unfolding** *hash-pro-spmf-def split-power[symmetric]* *l-def*
by (*auto simp add:set-bind-spmf*)
have $1: \text{order } (\text{ring-of } R) = p \wedge l$
using *galois-field-random-1(1)*[*OF prime-power R-in*] **by** *auto*
have $I \subseteq \{..<d\}$ **using** *assms* **by** *auto*
also have $\dots \subseteq \{..<\text{order } (\text{ring-of } R)\}$ **using** 0 **unfolding** 1 **by** *auto*
finally have $I \subseteq \{..<\text{order } (\text{ring-of } R)\}$ **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 $s \in \text{set-spmf } (\text{hash-pro-spmf } k d S)$
assumes $i < d$ $k > 0$
shows $\text{map-pmf } (\lambda f. f i) (\text{sample-pro } s) = \text{sample-pro } S$ (**is** $?L = ?R$)
proof –
have $?L = \text{map-pmf } (\lambda f. f i) (\text{map-pmf } (\lambda i \in \{i\}. f i) (\text{sample-pro } s))$
using *assms(1)* **unfolding** *map-pmf-comp* **by** (*intro map-pmf-cong refl*) *auto*
also have $\dots = \text{map-pmf } (\lambda f. f i) (\text{prod-pmf } \{i\} (\lambda \cdot. \text{sample-pro } S))$
using *assms* **by** (*subst hash-pro-spmf-distr*[*OF assms(1)*]) *auto*
also have $\dots = ?R$ **by** (*subst Pi-pmf-component*) *auto*
finally show *?thesis* **by** *simp*
qed

lemma *hash-pro-spmf-indep*:

assumes $s \in \text{set-spmf } (\text{hash-pro-spmf } k d S)$
assumes $I \subseteq \{..<d\}$ $\text{card } I \leq k$
shows $\text{prob-space.indep-vars } (\text{sample-pro } s) (\lambda \cdot. \text{discrete}) (\lambda i \omega. \omega i) I$
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 $\text{card } J \leq \text{card } I$ **using** *finite-subset*[*OF assms(2)*] **by** (*intro card-mono a*)
auto
also have $\dots \leq k$ **using** *assms(3)* **by** *simp*
finally have $1: \text{card } J \leq k$ **by** *simp*
let $?s = \text{sample-pro } s$

have $2: 0 < k$ **if** $x \in J$ **for** x

proof –
have $0 < \text{card } J$ **using** 0 that *card-gt-0-iff finite-nat-iff-bounded* **by** *auto*
also have $\dots \leq k$ **using** 1 **by** *simp*
finally show *?thesis* **by** *simp*
qed

have $\text{measure } ?s \{\omega. \forall j \in J. \omega j = x j\} = \text{measure } (\text{map-pmf } (\lambda \omega. \lambda j \in J. \omega j) ?s)$
 $\{\omega. \forall j \in J. \omega j = x j\}$
by *auto*
also have $\dots = \text{measure } (\text{prod-pmf } J (\lambda \cdot. \text{sample-pro } S)) (\text{Pi } J (\lambda j. \{x j\}))$
unfolding *hash-pro-spmf-distr[OF assms(1) 0 1]* **by** (*intro arg-cong2[where f=measure]*) (*auto simp:Pi-def*)
also have $\dots = (\prod j \in J. \text{measure } (\text{sample-pro } S) \{x j\})$
using *finite-subset[OF a] finite-subset[OF assms(2)]* **by** (*intro measure-Pi-pmf-Pi*)
auto
also have $\dots = (\prod j \in J. \text{measure } (\text{map-pmf } (\lambda \omega. \omega j) ?s) \{x j\})$
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 $\dots = (\prod j \in J. \text{measure } ?s \{\omega. \omega j = x j\})$ **by** (*simp add:vimage-def*)
finally show $\text{measure } ?s \{\omega. \forall j \in J. \omega j = x j\} = (\prod j \in J. \text{measure-pmf.prob } ?s$
 $\{\omega. \omega j = x j\})$
by *simp*
qed

lemma *hash-pro-spmf-k-indep*:
assumes $s \in \text{set-spmf } (\text{hash-pro-spmf } k\ d\ S)$
shows *prob-space.k-wise-indep-vars (sample-pro s) k* ($\lambda \cdot. \text{discrete}$) ($\lambda i\ \omega. \omega\ i$)
 $\{.. < d\}$
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 $s \in \text{set-spmf } (\text{hash-pro-spmf } k\ d\ S)$
shows *pro-size* $s = (p \wedge l) \wedge k$ (**is** $?L = ?R$)
proof –
obtain R **where** *R-in*: $R \in \text{set-spmf } (GF_R (p \wedge l))$ **and** *s-def*: $s = \text{hash-space}'\ R$
 $k\ S$
using *assms(1) unfolding hash-pro-spmf-def split-power[symmetric] l-def*
by (*auto simp add:set-bind-spmf*)
have $1: \text{order } (\text{ring-of } R) = p \wedge l$ **and** *ec*: $\text{enum}_C\ R$
using *galois-field-random-1(1)[OF prime-power R-in]* **by** *auto*

have $?L = \text{idx-size } R \wedge k - 1 + 1$
unfolding *s-def pro-size-def hash-space'-def* **by** *simp*
also have $\dots = ((p \wedge l) \wedge k - 1) + 1$
using 1 *enum-cD(2)[OF ec]* **by** *simp*
also have $\dots = (p \wedge l) \wedge k$ **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 ⌈log (real b) (real a+1)⌉ else 0)

proof (cases a > 0 ∧ 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 (⌈log b (real a + 1)⌉)

using True by (intro iffD2[OF powr-le-cancel-iff]) auto

also have ... = b powr (real (nat ⌈log b (real a + 1)⌉))

using 1 by (intro arg-cong2[where f=(powr)] refl) linarith

also have ... = b ^ nat ⌈log (real b) (real a + 1)⌉ using True by (subst powr-realpow) auto

finally have a < b ^ nat ⌈log (real b) (real a + 1)⌉ by simp

hence 0:floorlog b a ≤ nat ⌈log (real b) (real a+1)⌉ using True by (intro floorlog-leI) auto

have b ^ (nat ⌈log b (real a + 1)⌉ - 1) = b powr (real (nat ⌈log b (real a + 1)⌉ - 1))

using True by (subst powr-realpow) auto

also have ... = b powr (⌈log b (real a + 1)⌉ - 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 ⌈log (real b) (real a + 1)⌉ - 1) < a + 1 by linarith

hence b ^ (nat ⌈log (real b) (real a + 1)⌉ - 1) ≤ a by simp

hence floorlog b a ≥ nat ⌈log (real b) (real a+1)⌉ using True by (intro floorlog-geI) auto

hence floorlog b a = nat ⌈log (real b) (real a+1)⌉ using 0 by linarith

also have ... = (if 1 < b then nat ⌈log (real b) (real a+1)⌉ 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-power (pro-size S)

shows pro-size s = (p' ^ (max j' (floorlog p' (d-1)))) ^ 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 $s \in \text{set-spmf } (\text{hash-pro-spmf } k \ d \ S)$ $d > 0$

assumes $(p', j') = \text{split-power } (\text{pro-size } S)$

shows $\text{pro-size } s = (p' \wedge^{k * \max j' (\text{nat } \lceil \log p' \ d \rceil)})$

proof –

have $\text{pro-size } s = (p \wedge^{\max j (\text{floorlog } p \ (d-1))}) \wedge^k$

unfolding hash-pro-spmf-size-aux[OF assms(1)] l-def by simp

also have $\dots = (p \wedge^{\max j (\text{nat } \lceil \log p \ (\text{real } (d-1)+1) \rceil)}) \wedge^k$

using prime-gt-1-nat[OF p-prime] by (simp add: floorlog-alt-def)

also have $\dots = (p \wedge^{\max j (\text{nat } \lceil \log p \ d \rceil)}) \wedge^k$ using assms(2) by (subst of-nat-diff)

auto

also have $\dots = p \wedge^{k * \max j (\text{nat } \lceil \log p \ d \rceil)}$ by (simp add: ac-simps power-mult[symmetric])

also have $\dots = p' \wedge^{k * \max j' (\text{nat } \lceil \log p' \ d \rceil)}$

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 $s \in \text{set-spmf } (\text{hash-pro-spmf } k \ d \ S)$

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 $s \in \text{set-spmf } (\text{hash-pro-spmf } k \ d \ S)$

shows pro-select s i q \in pro-set S

proof –

obtain R where R-in: $R \in \text{set-spmf } (GF_R \ (p \wedge^l))$ and s-def: $s = \text{hash-space}' \ R$
 $k \ S$

using assms(1) unfolding hash-pro-spmf-def split-power[symmetric] l-def

by (auto simp add: set-bind-spmf)

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

bundle pseudorandom-object-notation

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

bundle no-pseudorandom-object-notation

begin

```

no-notation hash-pro ( $\mathcal{H}$ )
no-notation hash-pro-spmf ( $\mathcal{H}_S$ )
no-notation hash-pro-pmf ( $\mathcal{H}_P$ )
no-notation list-pro ( $\mathcal{L}$ )
no-notation nat-pro ( $\mathcal{N}$ )
no-notation geom-pro ( $\mathcal{G}$ )
no-notation prod-pro (infixr  $\times_P$  65)

```

end

unbundle pseudorandom-object-notation

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 & Experiments, 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.