# Enumeration of Equivalence Relations 

Emin Karayel

March 11, 2024


#### Abstract

This entry contains a formalization of an algorithm enumerating all equivalence relations on an initial segment of the natural numbers. The approach follows the method described by Stanton and White [5, §1.5] using restricted growth functions.

The algorithm internally enumerates restricted growth functions (as lists), whose equivalence kernels then form the equivalence relations. This has the advantage that the representation is compact and lookup of the relation reduces to a list lookup operation.

The algorithm can also be used within a proof and an example application is included, where a sequence of variables is split by the possible partitions they can form.


## 1 Introduction

theory Equivalence-Relation-Enumeration imports HOL-Library.Sublist HOL-Library.Disjoint-Sets Card-Equiv-Relations.Card-Equiv-Relations<br>begin

As mentioned in the abstract the enumeration algorithm relies on the bijection between restricted growth functions (RGFs) of length $n$ and the equivalence relations on $\{. .<n\}$, where the bijection is the operation that forms the equivalence kernels of an RGF. The method is being dicussed, for example, by $[3,4]$ or $[5, \S 1.5]$.
An enumeration algorithm for RGFs is less convoluted than one for equivalence relations or partitions and the representation has the advantage that checking whether a pair of elements are equivalent can be done by performing two list lookup operations.

After a few preliminary results in the following section, Section 3 introduces the enumeration algorithm for RGFs and shows that the function enumerates all of them (for the given length) without repetition. Section 4 shows that the operation of forming the equivalence kernel is a bijection and concludes with the correctness of the entire algorithm. In Section 5 an interesting application is being discussed, where the enumeration of partitions is applied
within a proof. Section 6 contains a few additional results, such as the fact that the length of the enumerated list is a Bell number. The latter result relies on the formalization of the cardinality of equivalence relations by Bulwahn [2].

## 2 Preliminary Results

This section contains a few preliminary results used in the proofs below.
lemma length-filter:length (filter $p x s)=$ sum-list (map $(\lambda x$.of-bool ( $p x)$ ) xs)
by (induct xs, simp-all)
lemma count-list-expand:count-list xs $x=$ length $($ filter $((=) x) x s)$
by (induct xs, simp-all)
An induction schema (similar to list-induct2 and rev-induct) for two lists of equal length, where induction step is shown appending elements at the end.

```
lemma list-induct-2-rev[consumes 1, case-names Nil Cons]:
    assumes length }x=\mathrm{ length }
    assumes P [] []
    assumes \x xs y ys. length xs = length ys \LongrightarrowP xs ys \LongrightarrowP(xs@[x])(ys@[y])
    shows P x y
    using assms(1)
proof (induct length x arbitrary: x y)
    case 0
    then show ?case using assms(2) by simp
next
    case (Suc n)
    obtain x1 x2 where a:x=x1@[x2] and c:length x1 = n
        by (metis Suc(2) append-butlast-last-id length-append-singleton
            length-greater-0-conv nat.inject zero-less-Suc)
    obtain y1 y2 where b:y= y1@[y2] and d:length y1 = n
        by (metis Suc(2,3) append-butlast-last-id length-append-singleton
            length-greater-0-conv nat.inject zero-less-Suc)
    have P x1 y1 using c d Suc by simp
    hence P(x1@[x2]) (y1@[y2]) using assms(3) c d by simp
    thus ?case using a b by simp
qed
```

If all but one value of a sum is zero then it can be evaluated on the remaining point:
lemma sum-collapse:
fixes $f:: ' a \Rightarrow$ ' $b::\{$ comm-monoid-add $\}$
assumes finite $A$
assumes $z \in A$

```
assumes \(\backslash y . y \in A \Longrightarrow y \neq z \Longrightarrow f y=0\)
shows sum \(f A=f z\)
using sum.union-disjoint \([\) where \(A=A-\{z\}\) and \(B=\{z\}\) and \(g=f]\)
by (simp add: assms sum.insert-if)
```

Number of occurrences of elements in lists is preserved under injective maps.

```
lemma count-list-inj-map:
    assumes inj-onf (set x)
    assumes }y\in\operatorname{set}
    shows count-list (map fx) (fy)= count-list x y
    using assms by (induction x, simp-all, fastforce)
```

A relation cannot be an equivalence relation on two distinct sets.

```
lemma equiv-on-unique:
    assumes equiv A p
    assumes equiv B p
    shows }A=
    by (meson assms equalityI equiv-class-eq-iff subsetI)
```

The restriction of an equivalence relation is itself an equivalence relation.

```
lemma equiv-subset:
    assumes B\subseteqA
    assumes equiv A p
    shows equiv B (Restr p B)
proof -
    have refl-on B (Restr p B) using assms by (simp add:refl-on-def equiv-def, blast)
    moreover have sym (Restr p B) using assms by (simp add:sym-def equiv-def)
    moreover have trans (Restr p B)
    using assms by (simp add:trans-def equiv-def, blast)
    ultimately show ?thesis by (simp add:equiv-def)
qed
```


## 3 Enumerating Restricted Growth Functions

```
fun rgf-limit :: nat list \(\Rightarrow\) nat
    where
        rgf-limit [] \(=0 \mid\)
        rgf-limit \((x \# x s)=\max (x+1)(\) rgf-limit \(x s)\)
```

lemma rgf-limit-snoc: rgf-limit $(x @[y])=\max (y+1)($ rgf-limit $x)$
by (induction $x$, simp-all)
lemma rgf-limit-ge: $y \in$ set $x s \Longrightarrow y<r g f-l i m i t ~ x s ~$
by (induction xs, simp-all, metis lessI max-less-iff-conj not-less-eq)
definition rgf :: nat list $\Rightarrow$ bool
where $\operatorname{rgf} x=(\forall y s y$. prefix $(y s @[y]) x \longrightarrow y \leq r g f-l i m i t ~ y s)$

The function rgf-limit returns the smallest natural number larger than all list elements, it is the largest allowed value following $x s$ for restricted growth functions. The definition rgf is the predicate capturing the notion.

```
fun enum-rgfs :: nat \(\Rightarrow\) (nat list) list
    where
        enum-rgfs \(0=[[]]\) |
        enum-rgfs \((\) Suc \(n)=[(x @[y]) . x \leftarrow\) enum-rgfs \(n, y \leftarrow[0 . .<\) rgf-limit \(x+1]]\)
```

The function enum-rgfs $n$ returns all RGFs of length $n$ without repetition. The fact is verified in the three lemmas at the end of this section.

```
lemma rgf-snoc:
    rgf (xs@[x])\longleftrightarrow rgf xs ^ x < rgf-limit xs + 1
    unfolding rgf-def by (rule order-antisym, (simp add:less-Suc-eq-le)+)
lemma rgf-imp-initial-segment:
    rgf xs \Longrightarrow set xs = {..<rgf-limit xs }
proof (induction xs rule:rev-induct)
    case Nil
    then show?case by simp
next
    case (snoc x xs)
    have c:rgf xs using snoc(2) rgf-snoc by simp
    hence a:set xs = {..<rgf-limit xs} using snoc(1) by simp
    have b: x \leq rgf-limit xs using snoc(2) rgf-snoc c by simp
    have set (xs@[x])= insert x {..<rgf-limit xs }
        using a by simp
    also have ... ={..<max (x+1) (rgf-limit xs)} using b
        by (cases x < rgf-limit xs, simp add:insert-absorb, simp add:lessThan-Suc)
    also have ... ={..<rgf-limit (xs@[x])}
        using rgf-limit-snoc by simp
    finally show ?case by simp
qed
lemma enum-rgfs-returns-rgfs:
    assumes }x\in\mathrm{ set (enum-rgfs n)
    shows rgf x
    using assms
proof (induction n arbitrary: x)
    case 0
    then show ?case by (simp add:rgf-def)
next
    case (Suc n)
    obtain x1 x2 where
        x-def:x = x1@[x2] x2 < rgf-limit x1 + 1 x1 \in set (enum-rgfs n)
        using Suc by (simp add:image-iff, force)
    have a:rgf x1 using Suc x-def by blast
    thus ?case using x-def by (simp add:rgf-snoc)
qed
```

```
lemma enum-rgfs-len:
    assumes }x\in\mathrm{ set (enum-rgfs n)
    shows length x = n
    using assms by (induction n arbitrary: x, simp-all, fastforce)
lemma equiv-rels-enum:
    assumes rgf x
    shows count-list (enum-rgfs (length x)) x=1
    using assms
proof (induction x rule:rev-induct)
    case Nil
    then show ?case by simp
next
    case (snoc x xs)
    have b:rgf xs using snoc(2) rgf-def by simp
    hence }x<rgf-limit xs + 1 using rgf-snoc snoc by blas
    hence a:card ({0..<rgf-limit xs + 1}\cap{x})=1 by force
    have 1 = count-list (enum-rgfs (length xs)) xs using snoc b by simp
    also have ... = (\sumr1\leftarrowenum-rgfs (length xs). of-bool (xs = r1)*
        card ({0..<rgf-limit xs + 1} \cap{x}))
        using a by (simp add:length-concat filter-concat count-list-expand length-filter)
    also have ... = (\sumr1\leftarrowenum-rgfs (length xs). of-bool (xs=r1)*
        card ({0..<rgf-limit r1 + 1} \cap{x}))
    by (metis (mono-tags, opaque-lifting) mult-eq-0-iff of-bool-eq-0-iff)
    also have ... = (\sumr1\leftarrowenum-rgfs (length xs). of-bool (xs = r1)*
        (\sumr2\leftarrow[0..<rgf-limit r1 + 1]. of-bool (x=r2)))
    by (simp add:interv-sum-list-conv-sum-set-nat del:One-nat-def)
    also have ... = length (filter ((=) (xs@[x])) (enum-rgfs (length (xs@[x]))))
    by (simp add:length-concat filter-concat length-filter comp-def
        of-bool-conj sum-list-const-mult del:upt-Suc)
    also have ... = count-list (enum-rgfs (length (xs@[x]))) (xs@[x])
    by (simp add:count-list-expand length-filter del:enum-rgfs.simps)
    finally show ?case by presburger
qed
```


## 4 Enumerating Equivalence Relations

The following definition returns the equivalence relation induced by a list, for example, by a restricted growth function.
definition kernel-of :: 'a list $\Rightarrow$ nat rel
where kernel-of $x s=\{(i, j) . i<$ length $x s \wedge j<$ length $x s \wedge x s!i=x s!j\}$
Using that the enumeration function for equivalence relations on $\{. .<n\}$ is straight-forward to define:
definition equiv-rels where equiv-rels $n=$ map kernel-of (enum-rgfs $n$ )
The following lemma shows that the image of kernel-of is indeed an equivalence relation:

```
lemma kernel-of-equiv: equiv \(\{. .<\) length \(x s\}\) (kernel-of \(x s\) )
proof -
    have kernel-of \(x s \subseteq\{. .<\) length \(x s\} \times\{. .<\) length \(x s\}\)
        by (rule subsetI, simp add:kernel-of-def mem-Times-iff case-prod-beta)
    thus ?thesis by (simp add:equiv-def refl-on-def sym-def trans-def kernel-of-def)
qed
lemma kernel-of-eq-len:
    assumes kernel-of \(x=\) kernel-of \(y\)
    shows length \(x=\) length \(y\)
proof -
    have \(\{. .<\) length \(x\}=\{. .<\) length \(y\}\)
        by (metis kernel-of-equiv equiv-on-unique assms)
    thus ?thesis by simp
qed
lemma kernel-of-eq:
    (kernel-of \(x=\) kernel-of \(y) \longleftrightarrow\)
    (length \(x=\) length \(y \wedge(\forall j<\) length \(x . \forall i<j .(x!i=x!j)=(y!i=y!j)))\)
proof (cases length \(x=\) length \(y\) )
    case True
    have (kernel-of \(x=\) kernel-of \(y\) ) \(\longleftrightarrow\)
        \((\forall j<\) length \(x . \forall i<\) length \(x .(x!i=x!j)=(y!i=y!j))\)
        unfolding set-eq-iff kernel-of-def using True by (simp, blast)
    also have \(\ldots \longleftrightarrow(\forall j<\) length \(x . \forall i<j .(x!i=x!j)=(y!i=y!j))\)
        by (metis (no-types, lifting) linorder-cases order.strict-trans)
    finally show ?thesis using True by simp
next
    case False
    then show ?thesis using kernel-of-eq-len by blast
qed
lemma kernel-of-snoc:
    kernel-of \((x s)=\operatorname{Restr}(\) kernel-of \((x s @[x])) \quad\{. .<\) length \(x s\}\)
    by (simp add:kernel-of-def nth-append set-eq-iff)
lemma kernel-of-inj-on-rgfs-aux:
    assumes length \(x=\) length \(y\)
    assumes rgf \(x\)
    assumes rgf \(y\)
    assumes kernel-of \(x=\) kernel-of \(y\)
    shows \(x=y\)
    using assms
proof (induct \(x\) y rule: list-induct-2-rev)
    case Nil
    then show? case by simp
next
    case (Cons \(x\) xs y ys)
    have a:kernel-of \(x s=\) kernel-of ys
```

```
    using Cons(1,5) kernel-of-snoc by metis
    have d:rgf xs rgf ys using Cons rgf-def by auto
    hence b:xs = ys using Cons(2) a by auto
    have }\bigwedgei.i<length xs \Longrightarrow(xs!i=x)=(ys!i=y
    proof -
    fix i
    assume i-l:i< length xs
    have (xs ! i=x)\longleftrightarrow \longleftrightarrow (,length xs) \in kernel-of (xs@[x]) using i-l
        by (simp add:kernel-of-def less-Suc-eq nth-append)
    also have }\ldots\longleftrightarrow(i,length xs)\in kernel-of (ys@[y]
        using Cons(5) by simp
    also have }..\longleftrightarrow(ys!i=y)\mathrm{ using i-l Cons(1)
        by (simp add:kernel-of-def less-Suc-eq nth-append)
    finally show (xs!i=x)=(ys!i=y) by simp
qed
hence c:(x\in set xs \longrightarrowx=y)\wedge(x\not\in set xs \longrightarrow}y\not=\mathrm{ set ys)
    by (metis b in-set-conv-nth)
have x-bound:x < rgf-limit xs + 1
    using Cons(3) rgf-snoc d by simp
have y-bound:y<rgf-limit ys + 1
    using Cons(4) rgf-snoc d by simp
have }x=y\mathrm{ using b c d rgf-imp-initial-segment Cons x-bound y-bound
    apply (cases x < rgf-limit xs, simp)
    by (cases y < rgf-limit ys, simp+)
    then show ?case using b by simp
qed
lemma kernel-of-inj-on-rgfs:
    inj-on kernel-of {x.rgf x}
    by (rule inj-onI, simp, metis kernel-of-eq-len kernel-of-inj-on-rgfs-aux)
Applying an injective map to a list preserves the induced relation:
```

```
lemma kernel-of-under-inj-map:
```

lemma kernel-of-under-inj-map:
assumes inj-on $f$ (set $x$ )
assumes inj-on $f$ (set $x$ )
shows kernel-of $x=$ kernel-of ( $\operatorname{map} f x$ )
shows kernel-of $x=$ kernel-of ( $\operatorname{map} f x$ )
proof -
proof -
have $\bigwedge i j . i<$ length $x \Longrightarrow j<$ length $x$
have $\bigwedge i j . i<$ length $x \Longrightarrow j<$ length $x$
$\Longrightarrow(\operatorname{map} f x)!i=(\operatorname{map} f x)!j \Longrightarrow x!i=x!j$
$\Longrightarrow(\operatorname{map} f x)!i=(\operatorname{map} f x)!j \Longrightarrow x!i=x!j$
using assms by (simp add: inj-on-eq-iff)
using assms by (simp add: inj-on-eq-iff)
thus ?thesis unfolding kernel-of-def by fastforce
thus ?thesis unfolding kernel-of-def by fastforce
qed
qed
lemma all-rels-are-kernels:
lemma all-rels-are-kernels:
assumes equiv $\{. .<n\} p$
assumes equiv $\{. .<n\} p$
shows $\exists(x::$ nat set list $)$. kernel-of $x=p \wedge$ length $x=n$
shows $\exists(x::$ nat set list $)$. kernel-of $x=p \wedge$ length $x=n$
proof -
proof -
define $r$ where $r=\operatorname{map}\left(\lambda k . p^{\prime \prime}\{k\}\right)[0 . .<n]$
define $r$ where $r=\operatorname{map}\left(\lambda k . p^{\prime \prime}\{k\}\right)[0 . .<n]$
have $\wedge u v .(u, v) \in$ kernel-of $r \longleftrightarrow(u, v) \in p$

```
    have \(\wedge u v .(u, v) \in\) kernel-of \(r \longleftrightarrow(u, v) \in p\)
```

```
    proof -
    fix u v :: nat
    have }(u,v)\in\mathrm{ kernel-of r }\longleftrightarrow((u,v)\in{..<n}\times{..<n}\wedge p'"{u} = p'،{v}
        unfolding kernel-of-def r-def by auto
    also have ...\longleftrightarrow(u,v) & b by (metis assms equiv-class-eq-iff mem-Sigma-iff)
    finally show (u,v)\in kernel-of r \longleftrightarrow(u,v) \inp by simp
qed
hence kernel-of r = p by auto
moreover have length r=n using r-def by simp
ultimately show ?thesis by auto
qed
```

For any list there is always an injective map on its set, such that its image is an RGF.
lemma map-list-to-rgf:
$\exists f$. inj-on $f($ set $x) \wedge \operatorname{rgf}(\operatorname{map} f x)$
proof (induction length $x$ arbitrary: $x$ )
case 0
then show ?case by (simp add:rgf-def)
next
case (Suc n)
obtain $x 1$ x2 where $x$-def: $x=x 1 @[x 2]$ and $l$ - $x 1$ : length $x 1=n$
by (metis append-butlast-last-id length-append-singleton Suc(2)
length-greater-0-conv nat.inject zero-less-Suc)
obtain $f$ where $\operatorname{inj}-f: \operatorname{inj}$-on $f(s e t x 1)$ and $p c-f: r g f(\operatorname{map} f x 1)$
using $\operatorname{Suc}(1) l$-x1 by blast
show ?case
proof (cases x2 $\in$ set x1)
case True
have a:set $x=$ set $x 1$ using $x$-def True by auto
hence b:inj-on $f$ (set $x$ ) using inj-f by auto
have $f$ x2 < rgf-limit (map $f$ x1) using rgf-limit-ge True by auto
hence $\operatorname{rgf}(\operatorname{map} f x)$
by (simp add:x-def rgf-snoc pc-f)
then show ?thesis using $b$ by blast
next
case False
define $f^{\prime}$ where $f^{\prime}=(\lambda y$. if $y \in$ set $x 1$ then $f y$ else rgf-limit $(\operatorname{map} f x 1))$
have inj-on $f^{\prime}$ (set x1) using $f^{\prime}$-def inj-f by (simp add: inj-on-def)
moreover have rgf-limit (map fx1) $\notin \operatorname{set}($ map $f x 1)$
using rgf-limit-ge by blast
hence $f^{\prime} x 2 \notin f^{\prime}$ ' set $x 1$ using False by (simp add: $f^{\prime}$-def)
ultimately have inj-on $f^{\prime}$ (insert x2 (set x1)) using False by simp
hence $a$ :inj-on $f^{\prime}(\operatorname{set} x)$ using False $x$-def by simp
have $b: \operatorname{map} f x 1=\operatorname{map} f^{\prime} x 1$ using $f^{\prime}-\operatorname{def}$ by $\operatorname{simp}$
have $c: f^{\prime} x 2<$ Suc (rgf-limit (map $f$ x1)) by (simp add:f'-def False)

```
    have rgf (map f'x) by (simp add:x-def b[symmetric] rgf-snoc pc-f c)
    then show ?thesis using a by blast
    qed
qed
```

For any relation there is a corresponding RGF:
lemma rgf-exists:
assumes equiv $\{. .<n\} r$
shows $\exists x$. rgf $x \wedge$ length $x=n \wedge$ kernel-of $x=r$
proof -
obtain $y$ :: nat set list where a:kernel-of $y=r$ length $y=n$
using all-rels-are-kernels assms by blast
then obtain $f$ where b:inj-on $f($ set $y)$ rgf (map $f y)$ using map-list-to-rgf by blast
have kernel-of $(\operatorname{map} f y)=r$
using kernel-of-under-inj-map abby blast
moreover have length (map $f y$ ) $=n$ using $a$ by simp
ultimately show? thesis
using $b$ by blast
qed
These are the main result of this entry: The function equiv-rels $n$ enumerates the equivalence relations on $\{. .<n\}$ without repetition.
theorem equiv-rels-set:
assumes $x \in$ set (equiv-rels $n$ )
shows equiv $\{. .<n\} x$
using assms equiv-rels-def kernel-of-equiv enum-rgfs-len by auto
theorem equiv-rels:
assumes equiv $\{. .<n\} r$
shows count-list (equiv-rels n) $r=1$
proof -
obtain $y$ where $y$-def: rgf $y$ length $y=n$ kernel-of $y=r$
using rgf-exists assms by blast
have $a: \wedge x . x \in$ set (enum-rgfs $n) \Longrightarrow$ (kernel-of $y=$ kernel-of $x)=(y=x)$
using enum-rgfs-returns-rgfs y-def(1,2) enum-rgfs-len inj-onD[OF kernel-of-inj-on-rgfs]
by auto
have count-list (equiv-rels n) $r=$
length (filter ( $\lambda x . r=$ kernel-of $x)$ (enum-rgfs $n$ ))
by (simp add:equiv-rels-def count-list-expand length-filter comp-def)
also have $\ldots=$ length $($ filter $(\lambda x$. kernel-of $y=$ kernel-of $x)($ enum-rgfs $n))$
using $y$ - $\operatorname{def}(3)$ by simp
also have $\ldots=$ length $($ filter $(\lambda x . y=x)($ enum-rgfs $n))$
using $a$ by (simp cong:filter-cong)
also have..$=$ count-list (enum-rgfs $n$ ) $y$
by (simp add:count-list-expand length-filter)
also have $\ldots=1$

```
    using equiv-rels-enum y-def(1,2) by auto
    finally show ?thesis by simp
qed
```

A corollary of the previous theorem is that the sum of the indicator function for a relation over equiv-rels $n$ is always one.

```
corollary equiv-rels-2:
    assumes n= length xs
    shows (\sumx\leftarrowequiv-rels n. of-bool (kernel-of xs = x) ) = (1 :: 'a :: {semiring-1})
proof -
    have length (filter }(\lambdax\mathrm{ . kernel-of xs =x) (equiv-rels (length xs))) = 1
    using equiv-rels[OF kernel-of-equiv[where xs=xs]] assms by (simp add:count-list-expand)
    thus ?thesis
    using assms by (simp add:of-bool-def sum-list-map-filter'[symmetric] sum-list-triv)
qed
```


## 5 Example Application

In this section, I wanted to discuss an interesting application within the context of a proof in Isabelle. This is motivated by a real-world example $[1, \S 2.2]$, where a function in a 4-times iterated sum could only be reduced by splitting it according to the equivalence relation formed by the indices. The notepad below illustrates how this can be done (in the case of 3 index variables).

```
notepad
begin
    fix f :: nat }\times\mathrm{ nat }\times\mathrm{ nat }=>\mathrm{ nat
    fix I :: nat set
    assume a:finite I
```

To be able to break down such a sum by partitions let us introduce the function $P$ which is defined to be sum of an indicator function over all possible equivalence relations its argument can form:

```
define \(P\) :: nat list \(\Rightarrow\) nat
    where \(P=\left(\lambda x s\right.\). \(\left(\sum x \leftarrow\right.\) equiv-rels (length \(\left.x s\right)\). of-bool \((\) kernel-of \(\left.\left.x s=x)\right)\right)\)
```

Note that its value is always one, hence we can introduce it in an algebraic equation easily:

```
have P-one: }\xs.Pxs=
    by (simp add: P-def equiv-rels-2)
note unfold-equiv-rels = P-def equiv-rels-def numeral-eq-Suc kernel-of-eq
    neq-commute All-less-Suc comp-def
define r where r = (\sumi\inI. (\sumj\inI. (\sumk\inI. f(i,j,k))))
```

As a first step, we just introduce the factor $P[i, j, k]$.

```
have \(r=\left(\sum i \in I .\left(\sum j \in I .\left(\sum k \in I . f(i, j, k) * P[i, j, k]\right)\right)\right)\)
    by (simp add:P-one \(r\)-def cong:sum.cong)
```

By expanding the definition of P and distributing, the sum can be expanded into 5 sums each representing a distinct equivalence relation formed by the indices.

```
also have ...=
    (\sumi\inI.f (i,i,i))+
    (\sumi\inI. \sumj\inI.f(i,i,j)* of-bool (i\not=j))+
    (\sumi\inI. \sumj\inI.f (i,j,i)* of-bool (i\not=j))+
    (\sumi\inI. \sumj\inI.f (i,j,j)* of-bool (i\not=j))+
    (\sumi\inI. \sumj\inI.\sumk\inI.f (i,j,k)* of-bool ( }j\not=k\wedgei\not=k\wedgei\not=j)
    (is - = ?rhs)
    by (simp add:unfold-equiv-rels sum.distrib distrib-left sum-collapse[OF a])
    finally have r=?rhs by simp
end
```


## 6 Additional Results

If two lists induce the same equivalence relation, then there is a bijection between the sets that preserves the multiplicities of its elements.

```
lemma kernel-of-eq-imp-bij:
    assumes kernel-of \(x=\) kernel-of \(y\)
    shows \(\exists f\). bij-betw \(f(\) set \(x)(\) set \(y) \wedge\)
    \((\forall z \in\) set \(x\). count-list \(x z=\) count-list \(y(f z))\)
proof -
    obtain \(x^{\prime}\) where \(x^{\prime}\)-def: inj-on \(x^{\prime}(\) set \(x) \operatorname{rgf}\left(\right.\) map \(\left.x^{\prime} x\right)\)
        using map-list-to-rgf by blast
    obtain \(y^{\prime}\) where \(y^{\prime}\)-def: inj-on \(y^{\prime}(\) set \(y)\) rgf (map \(\left.y^{\prime} y\right)\)
        using map-list-to-rgf by blast
    have kernel-of (map \(\left.x^{\prime} x\right)=\) kernel-of (map \(y^{\prime} y\) )
        using assms \(x^{\prime}-\operatorname{def}(1) y^{\prime}-\operatorname{def}(1)\)
        by (simp add: kernel-of-under-inj-map[symmetric])
    hence b:map \(x^{\prime} x=\) map \(y^{\prime} y\)
        using inj-onD[OF kernel-of-inj-on-rgfs] \(x^{\prime}\)-def(2) \(y^{\prime}\)-def(2) length-map by simp
    hence \(f\) : \(x^{\prime}\) ' set \(x=y^{\prime}\) ' set \(y\)
        by (metis list.set-map)
    define \(f\) where \(f=\) the-inv-into (set \(y) y^{\prime} \circ x^{\prime}\)
    have \(g: \bigwedge z . z \in\) set \(x \Longrightarrow\) count-list \(x z=\) count-list \(y(f z)\)
    proof -
        fix \(z\)
        assume \(a: z \in \operatorname{set} x\)
        have \(e: x^{\prime} z \in y^{\prime}\) 'set \(y\)
        by (metis a \(b\) imageI image-set)
    have \(c\) : the-inv-into \((\) set \(y) y^{\prime}\left(x^{\prime} z\right) \in\) set \(y\)
        using e the-inv-into-into[OF \(y^{\prime}\)-def(1)] by simp
```



```
    using ef-the-inv-into-f y'-def(1) by force
    have count-list x z = count-list (map x' x) ( }\mp@subsup{x}{}{\prime}z
        using a x'-def by (simp add:count-list-inj-map)
    also have ... = count-list (map y' y)( (x'z)
        by (simp add:b)
    also have ... = count-list (map y' y) ( }\mp@subsup{y}{}{\prime}\mathrm{ (the-inv-into (set y) y' ( }\mp@subsup{x}{}{\prime}z))\mathrm{ )
        by (simp add:d)
    also have ... = count-list y (the-inv-into (set y) y' (x'z))
        using c count-list-inj-map[OF y'-def(1)] by simp
    also have ... = count-list y (fz) by (simp add:f-def)
    finally show count-list xz= count-list y(fz) by simp
qed
have bij-betw x' (set x) ( }\mp@subsup{x}{}{\prime}\mathrm{ ' set }x\mathrm{ )
    using x'-def(1) bij-betw-imageI by auto
moreover have bij-betw (the-inv-into (set y) y') ( y' ' set y) (set y)
    using bij-betw-the-inv-into[OF bij-betw-imageI] y'-def(1) by auto
    hence bij-betw (the-inv-into (set y) y') ( ( '' ' set x) (set y)
    using f by simp
ultimately have bij-betwf (set x) (set y)
    using bij-betw-trans f-def by blast
thus ?thesis using g}\mathrm{ by blast
qed
```

As expected the length of equiv-rels $n$ is the $n$-th Bell number.

```
lemma len-equiv-rels: length (equiv-rels \(n\) ) \(=\) Bell \(n\)
proof -
    have a:finite \(\{p\). equiv \(\{. .<n\} p\}\)
    by (simp add: finite-equiv)
    have \(b\) : set (equiv-rels \(n\) ) \(\subseteq\{\). equiv \(\{. .<n\} p\}\)
        using equiv-rels-set by blast
    have length (equiv-rels \(n\) ) \(=\)
        \(\left(\sum x \in\{p\right.\). equiv \(\{. .<n\} p\}\). count-list (equiv-rels \(n\) ) \(\left.x\right)\)
        using \(a b\) by (simp add:sum-count-set)
    also have \(\ldots=\) card \(\{\). . equiv \(\{. .<n\} p\}\)
        by (simp add: equiv-rels)
    also have \(\ldots=\) Bell \((\operatorname{card}\{. .<n\})\)
    using card-equiv-rel-eq-Bell by blast
    also have \(\ldots=\) Bell \(n\) by simp
    finally show ?thesis by simp
qed
```

Instead of forming an equivalence relation from a list, it is also possible to induce a partition from it:

```
definition induced-par :: 'a list \(\Rightarrow\) nat set set where
    induced-par xs \(=(\lambda k .\{i . i<\) length \(x s \wedge x s!i=k\})\) '(set \(x s)\)
```

The following lemma verifies the commutative diagram, i.e., induced-par xs is the same partition as the quotient of $\{. .<$ length $x s\}$ over the corresponding equivalence relation.

```
lemma quotient-of-kernel-is-induced-par:
    \(\{. .<\) length \(x s\} / /(\) kernel-of \(x s)=(\) induced-par \(x s)\)
proof (rule set-eqI)
    fix \(x\)
    have \(x \in\{. .<\) length \(x s\} / /(\) kernel-of \(x s) \longleftrightarrow\)
        \((\exists i<\) length \(x s . x=\{j . j<\) length \(x s \wedge x s!i=x s!j\})\)
        unfolding quotient-def kernel-of-def by blast
    also have \(\ldots \longleftrightarrow(\exists y \in\) set \(x s . x=\{j . j<\) length \(x s \wedge y=x s!j\})\)
        unfolding in-set-conv-nth Bex-def by (rule order-antisym, force+)
    also have \(\ldots \longleftrightarrow(x \in\) induced-par \(x s)\)
        unfolding induced-par-def by auto
    finally show \(x \in\{. .<\) length \(x s\} / /(\) kernel-of \(x s) \longleftrightarrow(x \in\) induced-par \(x s)\)
        by \(\operatorname{simp}\)
qed
end
```


## References

[1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. Journal of Computer and System Sciences, 58(1):137-147, 1999.
[2] L. Bulwahn. Cardinality of equivalence relations. Archive of Formal Proofs, May 2016. https://isa-afp.org/entries/Card_Equiv_Relations. html, Formal proof development.
[3] G. Hutchinson. Partioning algorithms for finite sets. Commun. ACM, 6(10):613-614, Oct. 1963.
[4] S. Milne. Restricted growth functions and incidence relations of the lattice of partitions of an n-set. Advances in Mathematics, 26(3):290-305, 1977.
[5] D. Stanton and D. White. Constructive Combinatorics. Springer, 1986.

