Van der Waerden's Theorem

Katharina Kreuzer, Manuel Eberl

March 17, 2025

Abstract

This article formalises the proof of Van der Waerden's Theorem from Ramsey theory.

Van der Waerden's Theorem states that for integers k and l there exists a number N which guarantees that if an integer interval of length at least N is coloured with k colours, there will always be an arithmetic progression of length l of the same colour in said interval. The proof goes along the lines of Swan [1].

The smallest number $N_{k,l}$ fulfilling Van der Waerden's Theorem is then called the Van der Waerden Number. Finding the Van der Waerden Number is still an open problem for most values of k and l.

Contents

```
theory Digits
  imports Complex_Main
begin
```

1 Representation of integers in different bases

First, we look at some useful lemmas for splitting sums.

```
lemma split_sum_first_elt_less: assumes "n<m"
  shows "(∑i∈{n..<m}. f i) = f n + (∑i∈{Suc n ..<m}. f i)"
  using sum.atLeast_Suc_lessThan assms by blast

lemma split_sum_mid_less: assumes "i<(n::nat)"
  shows "(∑j<n. f j) = (∑j<i. f j) + (∑j=i..<n. f j)"
proof -
  have "(∑j<n. f j) = (∑j∈{..<i} ∪ {i..<n}. f j)"
    using ‹i < n› by (intro sum.cong) auto
  also have "... = (∑j<i. f j) + (∑j=i..<n. f j)"
    by (subst sum.union_disjoint) auto
  finally show "(∑j<n. f j) = (∑j<i. f j) + (∑j=i..<n. f j)" .
qed
```

In order to use representation of numbers in a basis `base` and to calculate the conversion to and from integers, we introduce the following locale.

```
locale digits =
  fixes base :: nat
  assumes base_pos: "base > 0"
begin
```

Conversion from basis base to integers: `from_digits n d`

n:	`nat`	length of representation in basis base
d:	`nat ⇒ nat`	function of digits in basis base where `d i` is the i-th digit in basis base
output:	`nat`	natural number corresponding to $d(n-1)\dots d(0)$ as integer

```
fun from_digits :: "nat ⇒ (nat ⇒ nat) ⇒ nat" where
  "from_digits 0 d = 0"
| "from_digits (Suc n) d = d 0 + base * from_digits n (d ∘ Suc)"
```

Alternative definition using sum:

```
lemma from_digits_altdef: "from_digits n d = (∑i<n. d i * base ^ i)"
  by (induction n d rule: from_digits.induct)
     (auto simp add: sum.lessThan_Suc_shift o_def sum_distrib_left
       sum_distrib_right mult_ac simp del: sum.lessThan_Suc)
```

Digit in basis base of some integer number: `digit x i`

x:	*nat*	integer
i:	*nat*	index
output:	*nat*	i-th digit of representation in basis base of x

```
fun digit :: "nat ⇒ nat ⇒ nat" where
  "digit x 0 = x mod base"
| "digit x (Suc i) = digit (x div base) i"
```

Alternative definition using divisor and modulo:

```
lemma digit_altdef: "digit x i = (x div (base ^ i)) mod base"
  by (induction x i rule: digit.induct) (auto simp: div_mult2_eq)
```

Every digit must be smaller that the base.

```
lemma digit_less_base: "digit x i < base"
  using base_pos by (auto simp: digit_altdef)
```

A representation in basis *base* of length n must be less than $base^n$.

```
lemma from_digits_less:
  assumes "∀i<n. d i < base"
  shows "from_digits n d < base ^ n"
using assms proof (induct n d rule: from_digits.induct)
  case (2 n d)
  have "from_digits n (d ∘ Suc) ≤ base ^ n -1" using 2
    by (metis One_nat_def Suc_leI Suc_pred base_pos comp_apply
        less_Suc_eq_le zero_less_power)
  moreover have "d 0 ≤ base -1" using 2
    by (metis One_nat_def Suc_pred base_pos less_Suc_eq_0_disj
        less_Suc_eq_le)
  ultimately have "d 0 + base * from_digits n (d ∘ Suc) ≤
      base - 1 + base * (base^(n) - 1)"
    by (simp add: add_mono_thms_linordered_semiring(1))
  then show "from_digits (Suc n) d < base ^ Suc n"
    using base_pos by (auto simp:comp_def)
    (metis Suc_pred add_gr_0 le_imp_less_Suc mult_Suc_right
      zero_less_power)
qed auto
```

Lemmas for *mod* and *div* in number systems of basis *base*:

```
lemma mod_base: assumes "⋀i. i<n ⟹ d i < base" "n>0"
  shows "from_digits n d mod base = d 0 "
proof -
  have "(∑i<n. d i * base ^ i) mod base =
          (∑i<n. d i * base ^ i mod base) mod base"
  by (subst mod_sum_eq[symmetric]) simp
  then show ?thesis using assms
      sum.lessThan_Suc_shift[of "(λi. d i * base ^ i mod base)" "n-1"]
    unfolding from_digits_altdef by simp
qed
```

```
lemma mod_base_i:
  assumes "⋀i. i<n ⟹ d i < base" "n>0" "i<n"
  shows "(∑j=i..<n. d j * base ^ (j-i)) mod base = d i "
proof -
  have "(∑j=i..<n. d j * base ^ (j-i)) mod base =
        (∑j=i..<n. d j * base ^ (j-i) mod base) mod base"
    by (subst mod_sum_eq[symmetric]) simp
  then show ?thesis
    using assms split_sum_first_elt_less[where
        f = "(λj. d j * base ^ (j-i) mod base)"]
    unfolding from_digits_altdef by simp
qed

lemma div_base_i:
  assumes "⋀i. i<n ⟹ d i < base" "n>0" "i<n"
  shows "from_digits n d div (base ^i) = (∑j=i..<n. d j * base ^ (j-i))"
  unfolding from_digits_altdef proof -
  have base_exp: "base^(j) =  base^(j-i) * base^i"
    if "j∈{i..<n}" for j
    by (metis Nat.add_diff_assoc2 add_diff_cancel_right' atLeastLessThan_iff

        power_add that)
  have first:"(∑j<i. d j * base ^ j)< base ^ i"
    using assms from_digits_less[where n="i"]
    unfolding from_digits_altdef by auto
  have "(∑j<n. d j * base ^ j) =
          (∑j<i. d j * base ^ j) + (∑j=i..<n. d j * base ^ j)"
    using assms split_sum_mid_less[where f="(λj. d j * base^j)"] by auto
  then have split_sum: "(∑j<n. d j * base ^ j) =
      (∑j<i. d j * base ^ j) + base^i * (∑j=i..<n. d j * base ^ (j-i))"
    using base_exp mult.assoc sum_distrib_right
    by (smt (z3) mult.commute sum.cong)
  then show "(∑i<n. d i * base ^ i) div base ^ i =
             (∑j = i..<n. d j * base ^ (j - i))"
    using first by (simp add:split_sum base_pos)
qed
```

Conversions are inverse to each other.

```
lemma digit_from_digits:
  assumes "⋀j. j<n ⟹ d j < base" "n>0" "i<n"
  shows   "digit (from_digits n d) i = d i"
  using assms proof (cases "i=0")
  case True
  then show ?thesis
    by (simp add: assms(1) assms(2) digits.mod_base digits_axioms)
next
  case False
  have "from_digits n d div base^i mod base = d i"
    using assms by (auto simp add: div_base_i mod_base_i)
```

```
  then show "digit (from_digits n d) i = d i"
    unfolding digit_altdef by auto
qed

lemma div_distrib: assumes "i<n"
  shows "(a*base^n + b) div base^i mod base = b div base^i mod base"
proof -
  have "base^i dvd (a*base^n)" using assms
    by (simp add: le_imp_power_dvd)
  moreover have "a*base^n div base^i mod base = 0"
    by (metis Suc_leI assms dvd_imp_mod_0 dvd_mult
        dvd_mult_imp_div le_imp_power_dvd power_Suc)
  ultimately show ?thesis
    by (metis add.right_neutral div_mult_mod_eq
        div_plus_div_distrib_dvd_left mod_mult_self3)
qed

lemma from_digits_digit:
  assumes "x < base ^ n"
  shows   "from_digits n (digit x) = x"
  using assms unfolding digit_altdef from_digits_altdef
proof (induction n arbitrary: x)
  case 0
  then show ?case by simp
next
  case (Suc n)
  define x_less where "x_less = x mod base^n"
  define x_n where "x_n = x div base^n"
  have "x_less < base^n"
    using x_less_def base_pos mod_less_divisor by presburger
  then have IH_x_less:
    "(∑i<n. x_less div base ^ i mod base * base ^ i) = x_less"
    using Suc.IH by simp
  have "x_n < base" using ‹x<base^Suc n›
    by auto (metis less_mult_imp_div_less x_n_def)
  then have "x_n mod base = x_n" by simp
  have x_less_i_eq_x_i:"x mod base^n div base ^i mod base =
    x div base^i mod base" if "i<n" for i
  proof -
    have "x div base^i mod base =
          ((x div base^n) * base^n + x mod base^n) div base^i mod base"
      using div_mult_mod_eq[of x "base^n"] by simp
    also have "... = x mod base^n div base^i mod base"
      using div_distrib[where a="x div base^n" and b = "x mod base^n"]
        that by auto
    finally show ?thesis by simp
  qed
  have "x = (x_n mod base)*base^n + x_less"
    unfolding ‹x_n mod base=x_n›
```

```
    using x_n_def x_less_def div_mod_decomp by blast
  also have "... = (x div base^n mod base) * base^n +
                (∑i<n. x div base ^ i mod base * base ^ i)"
    using IH_x_less x_less_def x_less_i_eq_x_i x_n_def by auto
  finally show ?case using sum.atMost_Suc
    by (simp add: add.commute)
qed
```

Stronger formulation of above lemma.

```
lemma from_digits_digit':
  "from_digits n (digit x) = x mod (base ^ n)"
  unfolding from_digits_altdef digit_altdef
proof (induction n arbitrary: x)
  case 0
  then show ?case by simp
next
  case (Suc n)
  define x_less where "x_less = x mod base^n"
  define x_n where "x_n = x div base^n mod base"
  have "x_less < base^n" using x_less_def base_pos
      mod_less_divisor by presburger
  then have IH_x_less:
    "(∑i<n. x_less div base ^ i mod base * base ^ i) = x_less"
    using Suc.IH by simp
  have "x_n < base" using base_pos mod_less_divisor x_n_def
    by blast
  then have "x_n mod base = x_n" by simp
  have x_less_i_eq_x_i:"x mod base^n div base ^i mod base =
    x div base^i mod base" if "i<n" for i
  proof -
    have "x div base^i mod base =
      ((x div base^n) * base^n + x mod base^n) div base^i mod base"
      using div_mult_mod_eq[of x "base^n"] by simp
    also have "... = x mod base^n div base^i mod base"
      using div_distrib[where a="x div base^n" and b = "x mod base^n"]

        that by auto
    finally show ?thesis by simp
  qed
  have "x mod base^Suc n = x_n*base^n + x_less"
    by (metis mod_mult2_eq mult.commute power_Suc2 x_less_def x_n_def)
  also have "... = (x div base^n mod base) * base^n +
                (∑i<n. x div base ^ i mod base * base ^ i)"
    using IH_x_less x_less_def x_less_i_eq_x_i x_n_def by auto
  finally show ?case using sum.atMost_Suc
    by (simp add: add.commute)
qed

end
```

```
end
theory Van_der_Waerden
  imports Main "HOL-Library.FuncSet" Digits
begin
```

2 Van der Waerden's Theorem

In combinatorics, Van der Waerden's Theorem is about arithmetic progressions of a certain length of the same colour in a colouring of an interval. In order to state the theorem and to prove it, we need to formally introduce arithmetic progressions. We will express k-colourings as functions mapping an integer interval to the set $\{0, \dots, k-1\}$ of colours.

2.1 Arithmetic progressions

A sequence of integer numbers with the same step size is called an arithmetic progression. We say an m-fold arithmetic progression is an arithmetic progression with multiple step lengths.

Arithmetic progressions are defined in the following using the variables:

start:	`int`	starting value
step:	`nat`	positive integer for step length
i:	`nat`	i-th value in the arithmetic progression

```
definition arith_prog :: "int ⇒ nat ⇒ nat ⇒ int"
  where "arith_prog start step i = start + int (i * step)"
```

An m-fold arithmetic progression (which we will also call a multi-arithmetic progression) is defined in the following using the variables:

dims:	`nat`	number of dimensions/step directions of m-fold arithmetic progression
start:	`int`	starting value
steps:	`nat ⇒ nat`	function of steps, returns step in i-th dimension for $i \in [0.. < dims]$
c:	`nat ⇒ nat`	function of coefficients, returns coefficient in i-th dimension for $i \in [0.. < dims]$

```
definition multi_arith_prog ::
    "nat ⇒ int ⇒ (nat ⇒ nat) ⇒ (nat ⇒ nat) ⇒ int"
  where "multi_arith_prog dims start steps c =
          start + int (∑i<dims. c i * steps i)"
```

An m-fold arithmetic progression of dimension 1 is also an arithmetic progression and vice versa. This is shown in the following lemmas.

```
lemma multi_to_arith_prog:
  "multi_arith_prog 1 start steps c =
```

```
    arith_prog start (steps 0) (c 0)"
  unfolding multi_arith_prog_def arith_prog_def by auto

lemma arith_prog_to_multi:
  "arith_prog start step c =
    multi_arith_prog 1 start (λ_. step) (λ_. c)"
  unfolding multi_arith_prog_def arith_prog_def by auto
```

To show that an arithmetic progression is well-defined, we introduce the following predicate. It assures that `arith_prog start step ‘ [0..<l]` is contained in the integer interval $[a..b]$.

```
definition is_arith_prog_on ::
    "nat ⇒ int ⇒ nat ⇒ int ⇒ int ⇒ bool"
  where "is_arith_prog_on l start step a b ⟷
    (start ≥ a ∧ arith_prog start step (l-1) ≤ b)"
```

Furthermore, we have monotonicity for arithmetic progressions.

```
lemma arith_prog_mono:
  assumes "c ≤ c'"
  shows   "arith_prog start step c ≤ arith_prog start step c'"
  using assms unfolding arith_prog_def by (auto intro: mult_mono)
```

Now, we state the well-definedness of an arithmetic progression of length l in an integer interval $[a..b]$. Indeed, `is_arith_prog_on` guarantees that every element of `arith_prog start step` of length l lies in $[a..b]$.

```
lemma is_arith_prog_onD:
  assumes "is_arith_prog_on l start step a b"
  assumes "c ∈ {0..<l}"
  shows   "arith_prog start step c ∈ {a..b}"
proof -
  have "arith_prog start step 0 ≤ arith_prog start step c"
    by (rule arith_prog_mono) auto
  hence "arith_prog start step c ≥ a"
    using assms by (simp add: arith_prog_def is_arith_prog_on_def
                     add_increasing2)
  moreover have "arith_prog start step (l-1) ≥
                 arith_prog start step c"
    by (rule arith_prog_mono) (use assms(2) in auto)
  hence "arith_prog start step c ≤ b"
    using assms unfolding arith_prog_def is_arith_prog_on_def
    by linarith
  ultimately show ?thesis
    by auto
qed
```

We also need a predicate for an m-fold arithmetic progression to be well-defined. It assures that `multi_arith_prog start step ‘ [0..<l]^m` is contained in $[a..b]$.

```
definition is_multi_arith_prog_on ::
    "nat ⇒ nat ⇒ int ⇒ (nat ⇒ nat) ⇒ int ⇒ int ⇒ bool"
  where "is_multi_arith_prog_on l m start steps a b ⟷
     (start ≥ a ∧ multi_arith_prog m start steps (λ_. l-1) ≤ b)"
```

Moreover, we have monotonicity for m-fold arithmetic progressions as well.

```
lemma multi_arith_prog_mono:
  assumes "⋀i. i < m ⟹ c i ≤ c' i"
  shows   "multi_arith_prog m start steps c ≤
           multi_arith_prog m start steps c'"
  using assms unfolding multi_arith_prog_def
  by (auto intro!: sum_mono intro: mult_right_mono)
```

Finally, we get the well-definedness for m-fold arithmetic progressions of length l. Here, *is_multi_arith_prog_on* guarantees that every element of *multi_arith_prog start step* of length l lies in $[a..b]$.

```
lemma is_multi_arith_prog_onD:
  assumes "is_multi_arith_prog_on l m start steps a b"
  assumes "c ∈ {0..<m} → {0..<l}"
  shows   "multi_arith_prog m start steps c ∈ {a..b}"
proof -
  have "multi_arith_prog m start steps (λ_. 0) ≤
        multi_arith_prog m start steps c"
    by (rule multi_arith_prog_mono) auto
  hence "multi_arith_prog m start steps c ≥ a"
    using assms by (simp add: multi_arith_prog_def
       is_multi_arith_prog_on_def)
  moreover have "multi_arith_prog m start steps (λ_. l-1) ≥
                 multi_arith_prog m start steps c"
    by (rule multi_arith_prog_mono) (use assms in force)
  hence "multi_arith_prog m start steps c ≤ b"
    using assms by (simp add: multi_arith_prog_def
        is_multi_arith_prog_on_def)
  ultimately show ?thesis
    by auto
qed
```

2.2 Van der Waerden's Theorem

The property for a number n to fulfill Van der Waerden's theorem is the following:
For a k-colouring col of $[a..b]$ there exist

- *start*: starting value of an arithmetic progression
- *step*: step length of an arithmetic progression
- j: colour

such that `arith_prog start step` is a valid arithmetic progression of length l lying in $[a..b]$ of the same colour j.

The following variables will be used:

k:	`nat`	number of colours in segment colouring on $[a..b]$
l:	`nat`	length of arithmetic progression
n:	`nat`	number fulfilling Van der Waerden's Theorem

```
definition vdw ::
    "nat ⇒ nat ⇒ nat ⇒ bool"
  where "vdw k l n ⟷
     (∀a b col. b + 1 ≥ a + int n ∧ col ∈ {a..b} → {..<k} ⟶
       (∃j start step. j < k ∧ step > 0 ∧
        is_arith_prog_on l start step a b ∧
        arith_prog start step ‘ {..<l} ⊆ col -‘ {j} ∩ {a..b}))"
```

To better work with the property of Van der Waerden's theorem, we introduce an elimination rule.

```
lemma vdwE:
  assumes "vdw k l n"
          "b + 1 ≥ a + int n"
          "col ∈ {a..b} → {..<k}"
  obtains j start step where
    "j < k" "step > 0"
    "is_arith_prog_on l start step a b"
    "arith_prog start step ‘ {..<l} ⊆ col -‘ {j} ∩ {a..b}"
  using assms that unfolding vdw_def by metis
```

Van der Waerden's theorem implies that the number fulfilling it is positive. This is show in the following lemma.

```
lemma vdw_imp_pos:
  assumes "vdw k l n"
          "l > 0"
  shows "n > 0"
proof (rule Nat.gr0I)
  assume [simp]: "n = 0"
  show False
    using assms
    by (elim vdwE[where a = 1 and b = 0 and col = "λ_. 0"])
       (auto simp: lessThan_empty_iff)
qed
```

Van der Waerden's Theorem is trivial for a non-existent colouring. It also makes no sense for arithmetic progressions of length 0.

```
lemma vdw_0_left [simp, intro]: "n>0 ⟹ vdw 0 l n"
  by (auto simp: vdw_def)
```

In the case of $k = 1$, Van der Waerden's Theorem holds. Then every number has the same colour, hence also the arithmetic progression. A possible choice for the number fulfilling Van der Waerden Theorem is l.

```
lemma vdw_1_left:
  assumes "l>0"
  shows "vdw 1 l l"
unfolding vdw_def
proof (safe, goal_cases)
  case (1 a b col)
  have "arith_prog a 1 ‘ {..<l} ⊆ {a..b}"
    using 1(1) by (auto simp: arith_prog_def)
  also have "{a..b} = col -‘ {0} ∩ {a..b}"
    using 1(2) by auto
  finally have "arith_prog a 1 ‘ {..<l} ⊆ col -‘ {0} ∩ {a..b}"
    by auto
  moreover have "is_arith_prog_on l a 1 a b"
    unfolding is_arith_prog_on_def arith_prog_def using 1 assms
    by auto
  ultimately show "∃j start step. j < 1 ∧ 0 < step ∧
        is_arith_prog_on l start step a b ∧
        arith_prog start step ‘ {..<l} ⊆ col -‘ {j} ∩ {a..b}"
    by auto
qed
```

In the case $l = 1$, Van der Waerden's Theorem holds. As the length of the arithmetic progression is 1, it consists of just one element. Thus every nonempty integer interval fulfills the Van der Waerden property. We can prove $N_{k,1}$ to be 1.

```
lemma vdw_1_right: "vdw k 1 1"
unfolding vdw_def
proof safe
  fix a b :: int and col :: "int ⇒ nat"
  assume *: "a + int 1 ≤ b + 1" "col ∈ {a..b} → {..<k}"
  have "col a < k" using * by auto
  have "arith_prog a 1 ‘ {..<1} = {a}"
    using *(1) by (auto simp: arith_prog_def)
  also have "{a} ⊆ col -‘ {col a} ∩ {a..b}"
    using * by auto
  finally have "arith_prog a 1 ‘ {..<1} ⊆ col -‘ {col a} ∩ {a..b}"
    by auto
  moreover have "is_arith_prog_on 1 a 1 a b"
    unfolding is_arith_prog_on_def arith_prog_def
    using * by auto
  ultimately show  "∃j start step.
          j < k ∧ 0 < step ∧ is_arith_prog_on 1 start step a b ∧
          arith_prog start step ‘ {..<1} ⊆ col -‘ {j} ∩ {a..b}"
    using ‹col a <k› by blast
qed
```

In the case $l = 2$, Van der Waerden's Theorem holds as well. Here, any two distinct numbers form an arithmetic progression of length 2. Thus we only have to find two numbers with the same colour. Using the pigeonhole principle on $k+1$ values, we can find two integers with the same colour.

```
lemma vdw_2_right: "vdw k 2 (k+1)"
unfolding vdw_def
proof safe
  fix a b :: int and col :: "int ⇒ nat"
  assume *: "a + int (k + 1) ≤ b + 1" "col ∈ {a..b} → {..<k}"

  have "col ‘ {a..b} ⊆ {..<k}" using *(2) by auto
  moreover have "k+1 ≤ card {a..b}" using *(1) by auto
  ultimately have "card (col ‘ {a..b}) < card {a..b}" using *
    by (metis card_lessThan card_mono finite_lessThan le_less_trans
        less_add_one not_le)
  then have "¬ inj_on col {a..b}" using pigeonhole[of col "{a..b}"]
    by auto
  then obtain start start_step
    where pigeon: "col start = col start_step"
      "start < start_step"
      "start ∈ {a..b}"
      "start_step ∈ {a..b}"
    using inj_onI[of "{a..b}" col]
    by (metis not_less_iff_gr_or_eq)

  define step where "step = nat (start_step - start)"
  define j where "j = col start"

  have "j < k" unfolding j_def using *(2) pigeon(3) by auto
  moreover have "0 < step" unfolding step_def using pigeon(2) by auto
  moreover have "is_arith_prog_on 2 start step a b"
    unfolding is_arith_prog_on_def arith_prog_def step_def
    using pigeon by auto
  moreover {
  have "arith_prog start step i ∈ {start, start_step}" if "i<2" for i
    using that arith_prog_def step_def by (auto simp: less_2_cases_iff)
  also have "... ⊆ col -‘ {j} ∩ {a..b}"
    using pigeon unfolding j_def by auto
  finally have "arith_prog start step ‘ {..<2} ⊆ col -‘ {j} ∩ {a..b}"

    by auto
  }
  ultimately show "∃j start step.
          j < k ∧
          0 < step ∧
          is_arith_prog_on 2 start step a b ∧
          arith_prog start step ‘ {..<2} ⊆ col -‘ {j} ∩ {a..b}" by blast
qed
```

In order to prove Van der Waerden's Theorem, we first prove a slightly different lemma. The statement goes as follows:
For a k-colouring *col* on $[a..b]$ there exist

- *start*: starting value of an arithmetic progression
- *steps*: step length of an arithmetic progression

such that `f = multi_arith_prog m start step` is a valid m-fold arithmetic progression of length l lying in $[a..b]$ such that for every $s < m$ have: if $cj < l$ for all $j \leq s$ then $f(c_0, c_1, \ldots, c_{m-1})$ and $f(0, \ldots, 0, c_{s+1}, \ldots, c_{m-1})$ have the same colour.

The property of the lemma uses the following variables:

k:	`nat`	number of colours in segment colouring of $[a..b]$
m:	`nat`	dimension of m-fold arithmetic progression
l:	`nat`	$l+1$ is length of m-fold arithmetic progression
n:	`nat`	number fulfilling `vdw_lemma`

```
definition vdw_lemma :: "nat ⇒ nat ⇒ nat ⇒ nat ⇒ bool" where
  "vdw_lemma k m l n ⟷
     (∀a b col. b + 1 ≥ a + int n ∧ col ∈ {a..b} → {..<k} ⟶
       (∃start steps. (∀i<m. steps i > 0) ∧
        is_multi_arith_prog_on (l+1) m start steps a b ∧ (
           let f = multi_arith_prog m start steps
           in  (∀c ∈ {0..<m} → {0..l}. ∀s<m. (∀ j ≤ s. c j < l) ⟶
                   col (f c) = col (f (λi. if i ≤ s then 0 else c i))))))"
```

To better work with this property, we introduce an elimination rule for `vdw_lemma`.

```
lemma vdw_lemmaE:
  fixes a b :: int
  assumes "vdw_lemma k m l n"
    "b + 1 ≥ a + int n" "col ∈ {a..b} → {..<k}"
  obtains start steps where
    "⋀i. i < m ⟹ steps i > 0"
    "is_multi_arith_prog_on (l+1) m start steps a b"
    "let f = multi_arith_prog m start steps
     in  ∀c ∈ {0..<m} → {0..l}. ∀s<m. (∀ j ≤ s. c j < l) ⟶
           col (f c) = col (f (λi. if i ≤ s then 0 else c i))"
  using assms that unfolding vdw_lemma_def by blast
```

To simplify the following proof, we show the following formula.

```
lemma sum_mod_poly:
  assumes "(k::nat)>0"
  shows "(k - 1) * (∑ n∈{..<q}. k^n) < k^q "
proof -
  have "int ((k - 1) * (∑n<q. k ^ n)) =
        (int k - 1) * (∑n<q. int k ^ n)"
    using assms by (simp add: of_nat_diff)
  also have "... = int k ^ q - 1"
    by (induction q) (auto simp: algebra_simps)
  also have "... < int (k ^ q)"
```

```
    by simp
  finally show ?thesis by linarith
qed
```

The proof of Van der Waerden's Theorem now proceeds in three steps:

- Firstly, we show that the `vdw` property for all k proves the `vdw_lemma` for fixed l but arbitrary k and m. This is done by induction over m.
- Secondly, we show that `vdw_lemma` implies the induction step of `vdw` using the pigeonhole principle.
- Lastly, we combine the previous steps in an induction over l to show Van der Waerden's Theorem in the general setting.

Firstly, we need to show that `vdw` for arbitrary k implies `vdw_lemma` for fixed l. As mentioned earlier, we use induction over m.

```
lemma vdw_imp_vdw_lemma:
  fixes l
  assumes vdw_assms: "⋀k'. k'>0 ⟹ ∃n_k'. vdw k' l n_k'"
    and "l ≥ 2"
    and "m > 0"
    and "k > 0"
  shows   "∃N. vdw_lemma k m l N"
using ‹m>0› ‹k>0› proof (induction m rule: less_induct)
  case (less m)
  consider  "m=1" | "m>1" using less.prems by linarith
  then show ?case
  proof cases
```

Case $m = 1$: Show `vdw_lemma` for arithmetic progression, Induction start.

```
    assume "m = 1"

    obtain n where vdw: "vdw k l n" using vdw_assms ‹k>0› by blast
    define N where "N = 2*n"
    have "l>0" and "l>1" using ‹l≥2› by auto

    have "vdw_lemma k m l N"
      unfolding vdw_lemma_def
    proof (safe, goal_cases)
      case (1 a b col)
```

Divide $[a..b]$ in two intervals I_1, I_2 of same length and obtain arithmetic progression of length l in I_1.

```
      have col_restr: "col ∈ {a..a + int n - 1} → {..<k}"
        using 1 by (auto simp: N_def)
      then obtain j start step where prog:
```

```
    "j < k" "step > 0"
    "is_arith_prog_on l start step a (a + int n -1)"
    "arith_prog start step ‘ {..<l} ⊆
      col -‘ {j} ∩ {a..a + int n - 1}"
    using vdw 1 unfolding N_def by (elim vdwE)(auto simp:is_arith_prog_on_def)
  have range_prog_lessThan_l:
    "arith_prog start step i ∈ {a..a + int n -1}" if "i < l" for i
    using that prog by auto

  have "{a..a + int n-1}⊆{a..b}" using N_def "1"(1) by auto
  then have "a + 2* int n - 1 ≤ b" using 1(1) unfolding N_def
    by auto
```

Show that `arith_prog start step` is an arithmetic progression of length $l+1$ in $[a..b]$.

```
  have prog_in_ivl: "arith_prog start step i ∈ {a..b}"
    if "i ≤ l" for i
  proof (cases "i=l")
    case False
    have "i<l" using that False by auto
    then show ?thesis
      using range_prog_lessThan_l ‹{a..a + int n-1}⊆{a..b}› by force
  next
    case True
```

Show `step` $\leq |I_1|$ then have `arith_prog start step (l+1)∈[a..b]` as `arith_prog start step (l+1) = arith_prog start step l + step`

```
    have "start ∈ {a..a + int n -1}"
      using range_prog_lessThan_l[of 0]
      unfolding arith_prog_def by (simp add: ‹0 < l›)
    moreover have "start + int step ∈ {a..a + int n -1}"
      using range_prog_lessThan_l[of 1]
      unfolding arith_prog_def by (metis ‹1 < l› mult.left_neutral)
    ultimately have "step ≤ n" by auto
    have "arith_prog start step (l-1) ∈ {a..a + int n -1}"
      using range_prog_lessThan_l[of "l-1"] unfolding arith_prog_def
      using ‹0 < l› diff_less less_numeral_extra(1) by blast
    moreover have "arith_prog start step l =
                  arith_prog start step (l-1) + int step"
      unfolding arith_prog_def using ‹0 < l› mult_eq_if by force
    ultimately have "arith_prog start step l ∈ {a..b}"
      using ‹step≤n› N_def ‹a + 2* int n -1 ≤ b› by auto
    then show ?thesis using range_prog_lessThan_l using True
      by force
  qed

  have col_prog_eq: "col (arith_prog start step k) = j"
    if "k < l" for k
    using prog that by blast
```

```
      define steps :: "nat ⇒ nat" where steps_def: "steps = (λi. step)"
      define f where "f = multi_arith_prog 1 start steps"

      have rel_prop_1:
        "col (f c) = col (f (λi. if i < s then 0 else c i))"
        if "c ∈ {0..<1} → {0..l}" "s<1" "∀j≤s. c j < l" for c s
        using that by auto

      have arith_prog_on:
        "is_multi_arith_prog_on (l+1) m start steps a b"
        using prog(3) unfolding is_arith_prog_on_def is_multi_arith_prog_on_def
        using ‹m=1› arith_prog_to_multi steps_def prog_in_ivl by auto

      show ?case
        by (rule exI[of _ start], rule exI[of _ steps])
           (use rel_prop_1 ‹step > 0› ‹m = 1› arith_prog_on col_prog_eq
             multi_to_arith_prog in ‹auto simp: f_def Let_def steps_def›)
    qed
    then show ?case ..

  next
```

Case $m > 1$: Show `vdw_lemma` for m-fold arithmetic progression, Induction step $(m-1) \longrightarrow m$.

```
    assume "m>1"

    obtain q where vdw_lemma_IH:"vdw_lemma k (m-1) l q"
      using ‹1 < m› less by force
    have "k^q>0" using ‹k>0› by auto
    obtain n_kq where vdw: "vdw (k^q) l n_kq"
      using vdw_assms ‹k^q>0› by blast
    define N where "N = q + 2 * n_kq"
```

Idea: $[a..b] = I_1 \cup I_2$ where $|I_1| = 2 * n_{k,q}$ and $|I_2| = q$. Divide I_1 into blocks of length q and define a new colouring on the set of q-blocks where the colour of the block is the k-basis representation where the i-th digit corresponds to the colour of the i-th element in the block. Get an arithmetic progression of q-blocks of length $l+1$ in I_1, such that the first l q-blocks have the same colour. The step of the block-arithmetic progression is going to be the additional step in the induction over m.

```
    have "vdw_lemma k m l N"
      unfolding vdw_lemma_def
    proof (safe, goal_cases)
      case (1 a b col)
      have "n_kq>0" using vdw_imp_pos vdw ‹l≥2› by auto
      then have "N>0" by (simp add:N_def)
      then have "a≤b" using 1 by auto
```

```
then have "k>0" using 1 by (intro Nat.gr0I) force
have "l>0" and "l>1" using ‹l≥2› by auto
interpret digits k by (simp add: ‹0 < k› digits_def)
define col1 where "col1 = (λ x. from_digits q (λy. col (x + y)))"

have range_col1: "col1∈{a..a + int n_kq - 1} → {..<k^q}"
unfolding Pi_def
proof safe
  fix x assume "x∈{a..a + int n_kq - 1}"
  then have col_xn:"col (x + int n)∈{..<k}" if "n<q" for n :: nat
    using that 1 PiE N_def by auto
  have col_xn_upper_bound:"col (x + int n) ≤ k - 1"
    if "n<q" for n ::nat
    using that col_xn[of n] ‹k>0› by (auto)
  have "(∑n<q. col (x + int n) * k ^ n)≤
        (∑n<q. (k-1) *  k ^ n)"
    using col_xn_upper_bound by (intro sum_mono mult_right_mono)

      auto
  also have "... = (k-1) * (∑n<q. k ^ n)"
    by (rule sum_distrib_left[symmetric])
  also have "... < k^q" using sum_mod_poly ‹k>0› by auto
  finally show "col1 x <k^q" unfolding col1_def from_digits_altdef

    by auto
qed

obtain j start step where prog:
  "j < k^q" "step > 0"
  "is_arith_prog_on l start step a (a + int n_kq - 1)"
  "arith_prog start step ‘ {..<l} ⊆
    col1 -‘ {j} ∩ {a..a + int n_kq -1}"
  using vdw range_col1 by (elim vdwE) (auto simp: ‹k>0›)

have range_prog_lessThan_l:
  "arith_prog start step i ∈ {a..a + int n_kq -1}"
  if "i < l" for i
  using that prog by auto

have prog_in_ivl:
  "arith_prog start step i ∈ {a..a + 2 * int n_kq -1}"
  if "i ≤ l" for i
proof (cases "i=l")
  case False
  then have "i<l" using that by auto
  then show ?thesis using prog by auto
next
  case True
  have "start ∈ {a..a + int n_kq -1}"
```

```
    using range_prog_lessThan_l[of 0] unfolding arith_prog_def
    by (simp add: ‹0 < l›)
  moreover have "start + step ∈ {a..a + int n_kq -1}"
    using range_prog_lessThan_l[of 1] unfolding arith_prog_def
    by (metis ‹1 < l› mult.left_neutral)
  ultimately have "step ≤ n_kq" by auto
  have "arith_prog start step (l-1) ∈ {a..a + int n_kq -1}"
    using range_prog_lessThan_l[of "l-1"] unfolding arith_prog_def
    using ‹0 < l› diff_less less_numeral_extra(1) by blast
  moreover have "arith_prog start step l =
      arith_prog start step (l-1) + step"
    unfolding arith_prog_def using ‹0 < l› mult_eq_if by force
  ultimately have "arith_prog start step l ∈
      {a..a + 2 * int n_kq - 1}"
    using ‹step≤n_kq› by auto
  then show ?thesis using range_prog_lessThan_l using True
    by force
qed

have col_prog_eq: "col1 (arith_prog start step k) = j"
  if "k < l" for k
  using prog that by blast

have digit_col1:"digit (col1 x) y = col (x+int y)"
  if "x∈{a..<a + 2*int n_kq}" "y∈{..<q}"
  for x::int and y::nat unfolding col1_def using that
proof -
  have "⋀j'. j'<q ⟹ x+j'∈{a..b}"
    using "1"(1) N_def that(1) by force
  then have "⋀j'. j'<q ⟹ (λy. col (x+int y)) j' < k"
    using 1 that by auto
  then show "digit (from_digits q (λxa. col (x + int xa))) y =

           col (x + int y)"
    using digit_from_digits that 1 by auto
qed
```

Impact on the colour when taking the block-step.

```
have one_step_more:
  "col (arith_prog start' step i) = digit j (nat (start'-start))"

  if "start'∈{start..<start+q}" "i∈{..<l}" for start' i
proof -
  have "start ≤ start'" using that by simp
  have shift_arith_prog:
    "arith_prog start step i + (start' - start) =
      arith_prog start' step i"
    unfolding arith_prog_def by simp
  define diff where "diff = nat (start'-start)"
```

```
    have "diff ∈{..<q}" using that unfolding diff_def by auto
    have "col (arith_prog start step i + int diff) = digit j diff"
    proof -
      have "col1 (arith_prog start step i) = j"
        using col1_def prog that by blast
      moreover have " arith_prog start step i∈{a..a + 2 * int n_kq-1}"
        using prog(4) that by auto
      ultimately show ?thesis
        using digit_col1[where x = "arith_prog start step i"
            and y = "diff"]
          prog 1 ‹diff ∈{..<q}› by auto
    qed
    then show ?thesis unfolding diff_def 1
      by (auto simp: ‹start≤start'› shift_arith_prog)
  qed

  have one_step_more': "col (arith_prog start' step i) =
    col (arith_prog start' step 0)"
    if "start'∈{start..<start+q}" "i∈{..<l}" for start' i
    using that one_step_more[of start' 0]
      one_step_more[of start' i] by auto

  have start_q: "start + int q ≤ start + int q - 1 + 1" by linarith
  have "{start..start + int q-1} ⊆ {a..b}"
    using prog N_def 1(1) by (force simp: arith_prog_def is_arith_prog_on_def)

  then have col': "col ∈ {start..start + int q-1} → {..<k}"
    using 1 prog(4) by auto
```

Obtain an $(m-1)$-fold arithmetic progression in the starting q-bolck of the block arithmetic progression.

```
  obtain start_m steps_m where
    step_m_pos: "⋀i. i < m - 1 ⟹ 0 < steps_m i" and
    is_multi_arith_prog: "is_multi_arith_prog_on (l+1) (m - 1)
      start_m steps_m start (start + int q - 1)" and
    g_aux: "let g = multi_arith_prog (m - 1) start_m steps_m
      in  ∀c∈{0..<m - 1} → {0..l}. ∀s<m - 1. (∀j≤s. c j < l) ⟶
      col (g c) = col (g (λi. if i ≤ s then 0 else c i))"
    by (rule vdw_lemmaE[OF vdw_lemma_IH start_q col']) blast

  define g where "g = multi_arith_prog (m-1) start_m steps_m"
  have g: "col (g c) = col (g (λi. if i ≤ s then 0 else c i))"
    if "c ∈ {0..<(m-1)} → {0..l}" "s < m - 1" "∀j ≤ s. c j < l"
    for c s using g_aux that unfolding g_def Let_def by blast

  have range_g: "g c ∈ {start..start + int q - 1}"
    if "c ∈ {0..<m - 1} → {0..<(l+1)}" for c
    using is_multi_arith_prog_onD[OF is_multi_arith_prog that]
    by (auto simp: g_def)
```

Obtain an m-fold arithmetic progression by adding the block-step.

```
define steps :: "nat ⇒ nat" where steps_def:
  "steps = (λi.  (if i=0 then step else steps_m (i-1)))"
define f where "f = multi_arith_prog m start_m steps"
have f_step_g: "f c = int (c 0*step) + g (c ∘ Suc)" for c
proof -
  have "f c = start_m + int (∑i<Suc (m-1). c i * steps i)"
    using f_def unfolding multi_arith_prog_def
    using less.prems by auto
  also have "... = start_m + int (c 0 * steps 0) +
               int (∑i<m-1. c (Suc i) * steps (Suc i))"
    using sum.lessThan_Suc_shift[where n = "m-1"] by auto
  also have "... = start_m + int (c 0 * step) +
               int (∑i<m-1. c (Suc i) * steps_m i)"
    using steps_def by (auto split:if_splits)
  finally show ?thesis unfolding multi_arith_prog_def g_def
    by simp
qed
```

Show that this m-fold arithmetic progression fulfills all needed properties.

```
have steps_gr_0: "∀i<m. 0 < steps i"
  unfolding steps_def using step_m_pos prog by auto

have is_multi_on_f:
  "is_multi_arith_prog_on (l+1) m start_m steps a b"
proof -
  have "a ≤ start_m" using is_multi_arith_prog
    unfolding is_multi_arith_prog_on_def
    using is_arith_prog_on_def prog(3) by force
  moreover {
    have "f (λ_. l) = arith_prog (g ((λ_. l) ∘ Suc)) step l"
      using f_step_g unfolding arith_prog_def by auto
    also have "g ((λ_. l) ∘ Suc) ≤ start + q"
      using range_g[of "(λ_. l) ∘ Suc"] by auto
    then have "arith_prog (g ((λ_. l) ∘ Suc)) step l ≤
      arith_prog start step l + q"
      unfolding arith_prog_def by auto
    also have "...≤ b" using prog_in_ivl[of l]
      using is_multi_arith_prog unfolding is_multi_arith_prog_on_def
      using "1"(1) N_def by auto
    finally have "f (λ_. l) ≤ b" by auto
   }
   ultimately show ?thesis
     unfolding is_multi_arith_prog_on_def f_def by auto
qed
```

Show the relational property for all s.

```
have rel_prop_1:
  "col (f c) = col (f (λi. if i ≤ s then 0 else c i))"
```

```
  if "c ∈ {0..<m} → {0..1}" "s<m" "∀j≤s. c j < 1" for c s
proof (cases "s = 0")
  case True
  have "c 0 < 1" using that(3) True by auto
  have range_c_Suc: "c ∘ Suc ∈ {0..<m-1} → {0..1}"
    using that(1) by auto
  have "f c = arith_prog (g (c ∘ Suc)) step (c 0)"
    using f_step_g unfolding arith_prog_def by auto
  then have "col (f c) = col (arith_prog (g (c ∘ Suc)) step 0)"
    using one_step_more'[of "g (c ∘ Suc)" "c 0"] ‹c 0 < 1›
      range_g[of "c ∘ Suc"] range_c_Suc
      atLeastLessThanSuc_atLeastAtMost by auto
  also {
    have "(∑x<m - 1. int (c (Suc x)) * int (steps_m x)) =
              (∑x=1..<m. int(c x) * int (steps x))"
      by(rule sum.reindex_bij_witness[of _ "(λx. x-1)" "Suc"])
        (auto simp: steps_def split:if_splits)
    also have "... = (∑x<m. int (if x = 0 then 0 else c x) *
      int (steps x))"
      by (rule sum.mono_neutral_cong_left) auto
    finally have "arith_prog (g (c ∘ Suc)) step 0 =
      f (λi. if i ≤ s then 0 else c i)"
      unfolding f_def g_def multi_arith_prog_def arith_prog_def
      using True by auto
}
  finally show ?thesis by auto
next
  case False
  hence s_greater_0: "s > 0" by auto
  have range_c_Suc: "c ∘ Suc ∈ {0..<m-1} → {0..1}"
    using that(1) by auto
  have "c 0 < 1" using ‹s>0› that by auto
  have g_IH:
    "col (g c') = col (g (λi. if i ≤ s' then 0 else c' i))"
    if "c' ∈ {0..<m-1} → {0..1}" "s'<m-1" "∀j≤s'. c' j < 1"
    for c' s'
    using g_aux that unfolding multi_arith_prog_def g_def
    by (auto simp: Let_def)
  have g_shift_IH: "col (g (c ∘ Suc)) =
    col (g ((λi. if i∈{1..t} then 0 else c i) ∘ Suc))"
    if "c ∈ {1..<m} → {0..1}" "t∈{1..<m}" "∀j∈{1..t}. c j < 1"
    for c t
  proof -
    have "(λi. (if i ≤ t - 1 then 0 else (c ∘ Suc) i)) =
           (λi. (if i ∈ {1..t} then 0 else c i)) ∘ Suc"
      using that by (auto split: if_splits simp:fun_eq_iff)
    then have right:
      "g (λi. if i ≤ (t-1) then 0 else (c ∘ Suc) i) =
       g ((λi. if i∈{1..t} then 0 else c i) ∘ Suc)" by auto
```

```
        have "(c ∘ Suc)∈ {0..<m-1} → {0..1}" using that(1) by auto
        moreover have "t-1<m-1" using that(2) by auto
        moreover have"∀j≤t-1. (c ∘ Suc) j < 1" using that by auto
        ultimately have "col (g (c ∘ Suc)) =
          col (g (λi. (if i ≤ t-1 then 0 else (c ∘ Suc) i)))"
          using g_IH[of "(c ∘ Suc)" "t-1"] by auto
        with right show ?thesis by auto
      qed

      have "col (f c) = col (int (c 0 * step) + g (c ∘ Suc))"
        using f_step_g by simp
      also have "int (c 0 * step) + g (c ∘ Suc) =
        arith_prog (g (c ∘ Suc)) step (c 0)"
        by (simp add: arith_prog_def)
      also have "col ... = col (arith_prog (g (c ∘ Suc)) step 0)"
        using one_step_more'[of "g (c ∘ Suc)" "c 0"] ‹c 0 < 1›
          range_g[of "c ∘ Suc"] range_c_Suc
          atLeastLessThanSuc_atLeastAtMost by auto
      also have "... = col (g (c ∘ Suc))"
        unfolding arith_prog_def by auto
      also have "... = col (g ((λi. if  i∈{1..s} then 0 else c i) ∘
        Suc))" using g_shift_IH[of "c" s] ‹s>0› that by force
      also have "... = col ((λc. int (c 0 * step) +
        g (c ∘ Suc))(λi. if i≤s then 0 else c i))"
        by (auto simp: g_def multi_arith_prog_def)
      also have "... = col (f (λi. if i ≤ s then 0 else c i))"
        unfolding f_step_g by auto
      finally show ?thesis by simp
    qed

    show ?case
      by (rule exI[of _ start_m], rule exI[of _ steps])
         (use steps_gr_0 is_multi_on_f rel_prop_1 in
           ‹auto simp: f_def Let_def steps_def›)
  qed
  then show ?case ..
 qed
qed
```

Secondly, we show that `vdw_lemma` implies the induction step of Van der Waerden's Theorem using the pigeonhole principle.

```
lemma vdw_lemma_imp_vdw:
  assumes "vdw_lemma k k 1 N"
  shows   "vdw k (Suc 1) N"
unfolding vdw_def proof (safe, goal_cases)
```

Idea: Proof uses pigeonhole principle to guarantee the existence of an arithmetic progression of length $l+1$ with the same colour.

```
  case (1 a b col)
```

```
obtain start steps where prog:
  "⋀i. i < k ⟹ steps i > 0"
  "is_multi_arith_prog_on (l+1) k start steps a b"
  "let f = multi_arith_prog k start steps
   in  ∀c ∈ {0..<k} → {0..l}. ∀s<k. (∀ j ≤ s. c j < l) ⟶
         col (f c) = col (f (λi. if i ≤ s then 0 else c i))"
  using assms 1
  by (elim vdw_lemmaE[where a=a and b=b and col=col and m=k
        and k=k and l=l and n=N]) auto
```

Obtain a k-fold arithmetic progression f of length l from assumptions.

```
define f where "f = multi_arith_prog k start steps"
have rel_propE: "col (f c) = col (f (λi. if i ≤ s then 0 else c i))"
  if "c ∈ {0..<k} → {0..l}" "s<k" "∀ j ≤ s. c j < l"
  for c s
  using prog(3) that unfolding f_def Let_def by auto
```

There are $k+1$ values $a_r = f(0,\dots,0,l,\dots,l)$ with $0 \leq r \leq k$ zeros.

```
define a_r where "a_r = (λr. f (λi. (if i<r then 0 else l)))"
have range_col_a_r: "col (a_r x) < k" if "x < k+1" for x
proof -
  have "a_r x ∈ {a..b}" unfolding a_r_def f_def
    by (intro is_multi_arith_prog_onD[OF prog(2)]) auto
  thus ?thesis using 1 by blast
qed
then have "(col ∘ a_r) ‘ {..<k + 1} ⊆ {..<k}" using 1(2) by auto
then have "card ((col ∘ a_r) ‘ {..<k + 1}) ≤ card {..<k}"
  by (intro card_mono) auto
then have "¬ inj_on (col ∘ a_r) {..<k+1}"
  using pigeonhole[of "col ∘ a_r" "{..<k+1}"] by auto
```

Using the pigeonhole principle get r_1 and r_2 where a_{r_1} and a_{r_2} have the same colour.

```
then obtain r1 r2 where pigeon_cols:
    "r1∈{..<k+1}"
    "r2∈{..<k+1}"
    "r1 < r2"
    "(col ∘ a_r) r1 = (col ∘ a_r) r2"
  by (metis (mono_tags, lifting) linear linorder_inj_onI)
```

Show that the following function h is an arithmetic progression which fulfills all properties for Van der Waerden's Theorem.

```
define h where
  "h = (λx. f (λi. (if i<r1 then 0 else (if i<r2 then x else l))))"
have "h 0 = a_r r2" unfolding h_def a_r_def using ‹r1<r2›
  by (intro arg_cong[where f = f]) auto
moreover have "h l = a_r r1"  unfolding h_def a_r_def using ‹r1<r2›
  by (metis le_eq_less_or_eq less_le_trans)
```

```
ultimately have "col (h 0) = col (h 1)" using pigeon_cols(4) by auto
have h_col: "col (h 0) = col (h i)" if "i∈{..<l+1}" for i
proof (cases "i=l")
  case True
  then show ?thesis using ‹col (h 0) = col (h 1)› by auto
next
  case False
  then have "i<l" using that by auto
  let ?c = "(λidx. if idx < r1 then 0 else if idx < r2 then i else l)"
  have "?c∈{0..<k} → {0..l}"
    using that by auto
  moreover have "(∀j≤r2-1. ?c j < l)"
    using ‹i<l› pigeon_cols(3) by force
  ultimately have "col (f ?c) =
    col (f (λi. if i ≤ r2-1 then 0 else ?c i))"
    using rel_propE[of ?c "r2-1"] pigeon_cols by simp
  then show ?thesis unfolding h_def f_def
    by (smt (verit) Nat.lessE One_nat_def add_diff_cancel_left'
        le_less less_Suc_eq_le multi_arith_prog_mono plus_1_eq_Suc)
qed

define h_start where "h_start = start + l*(∑i∈{r2..<k}. steps i)"
define h_step where "h_step = (∑i∈{r1..<r2}. steps i)"
have h_arith_prog: "h = arith_prog h_start h_step"
proof -
  have "(∑x<k. int (if x < r1 then 0 else if x < r2 then y else l)
      * int (steps x)) =
    int l * (∑x = r2..<k. int (steps x)) +
      int y * (∑x = r1..<r2. int (steps x))"
    for y
  proof (cases "r2 = k")
    case True
    then have "r1<k" using pigeon_cols by auto
    with True have
      "(∑x<k. int (if x < r1 then 0 else if x < r2 then y else l)
        * int (steps x)) =
       (∑x<k. int (if x < r1 then 0 else y) * int (steps x))"
      by (intro sum.cong) auto
    also have "... = (∑x<r1. int (if x < r1 then 0 else y) *
        int (steps x)) + (∑x=r1..<k. int (if x < r1 then 0 else y)
        * int (steps x))"
      using split_sum_mid_less[of r1 k
          "(λx. int (if x < r1 then 0 else y) * int (steps x))"]
          ‹r1<k› by auto
    also have "... = (∑x=r1..<k. int y * int (steps x))" by auto
    also have "... = int y * (∑x=r1..<k. int (steps x))"
      by (auto simp: sum_distrib_left[of "int y"])
    finally show ?thesis using True by auto
  next
```

```
      case False
      then have "r2<k" using pigeon_cols by auto
      define aux_left where "aux_left =
        (λx. int (if x < r1 then 0 else if x < r2 then y else l)
          * int (steps x))"
      have "(∑x<k. aux_left x) = (∑x=r1..<k. aux_left x)"
        by (intro sum.mono_neutral_right) (auto simp: aux_left_def)
      also have "{r1..<k} = {r1..<r2} ∪ {r2..<k}"
        using ‹r1 < r2› ‹r2 < k› by auto
      also have "(∑x∈.... aux_left x) = (∑x=r1..<r2. aux_left x) +

        (∑x=r2..<k. aux_left x)"
        by (intro sum.union_disjoint) auto
      also have "(∑x=r1..<r2. aux_left x) =
        (∑x=r1..<r2. int y * int (steps x))"
        by (intro sum.cong) (auto simp: aux_left_def)
      also have "(∑x=r2..<k. aux_left x) =
        (∑x=r2..<k. int l * int (steps x))"
        using ‹r1 < r2› by (intro sum.cong) (auto simp: aux_left_def)
      finally show ?thesis
        by (simp add: aux_left_def sum_distrib_left)
    qed
    then show ?thesis
      unfolding arith_prog_def h_start_def h_step_def h_def f_def
        multi_arith_prog_def by (auto split:if_splits)
  qed

  define j where "j = col (h 0)"
  have case_j: "j<k" using 1 range_col_a_r ‹col (h 0) = col (h 1)›
      ‹h l = a_r r1› j_def pigeon_cols(1) by auto
  have case_step: "h_step > 0" unfolding h_step_def
    using pigeon_cols by (intro sum_pos prog(1)) auto

  have range_h: "h i ∈ {a..b}" if "i < l + 1" for i
    unfolding h_def f_def by (rule is_multi_arith_prog_onD[OF prog(2)])
      (use that in auto)

  have case_on: "is_arith_prog_on (l+1) h_start h_step a b"
    unfolding is_arith_prog_on_def h_arith_prog
    using range_h[of 0] range_h[of l]
    by (auto simp: Max_ge[of "{a..b}"] Min_le[of "{a..b}"]
        h_arith_prog arith_prog_def)

  have case_col: "h ‘ {..<Suc l} ⊆ col -‘ {j} ∩ {a..b}"
    using h_col range_h unfolding j_def by auto

  show ?case using case_j case_step case_on case_col
    by (auto simp: h_arith_prog)
qed
```

Lastly, we assemble all lemmas to finally prove Van der Waerden's Theorem by induction on l. The cases $l = 1$ and the induction start $l = 2$ are treated separately and have been shown earlier.

```
theorem van_der_Waerden: assumes "l>0" "k>0" shows "∃n. vdw k l n"
using assms proof (induction l arbitrary: k rule: less_induct)
  case (less l)
  consider  "l=1" | "l=2" | "l>2" using less.prems by linarith
  then show ?case
  proof (cases)
    assume "l=1"
    then show ?thesis using vdw_1_right by auto
  next
    assume "l=2"
    then show ?thesis using vdw_2_right by auto
  next
    assume "l > 2"
    then have "2≤l-1" by auto
    from less.IH[of "l-1"] ‹l>2›
    have "⋀k'. k'>0 ⟹ ∃n. vdw k' (l-1) n" by auto
    with vdw_imp_vdw_lemma[of "l-1" k k] ‹l-1≥2› ‹k>0›
      obtain N where "vdw_lemma k k (l-1) N" by auto
    then have "vdw k l N" using vdw_lemma_imp_vdw[of k "l-1" N]
      by (simp add: less.prems(1))
    then show ?thesis by auto
  qed
qed

end
```

References

[1] R. G. Swan. Van der Waerden's theorem on arithmetic progressions. Technical report, Department of Mathematics, University of Chicago. Online at http://www.math.uchicago.edu/~swan/expo/vdW.pdf.