Verified Checker for Sorting Network Size Bounds

Jannis Harder

December 7, 2019

Contents

theory *Sorting-Network-Bound*
  imports *Main*
begin

Due to the 0-1-principle we're only concerned with Boolean vectors. While we're interested in sorting vectors of a given fixed width, it is advantageous to represent them as a function from the naturals to Boolens.

type-synonym *vect* = ‹*nat* ⇒ *bool*›

To represent vectors of a fixed width, we extend them with True to infinity. This way monotonicity of a fixed width vector corresponds to monotonicity of our representation.

definition *fixed-width-vect* :: ‹*nat* ⇒ *vect* ⇒ *bool*› **where**
  ‹*fixed-width-vect n v* = (∀ *i* ≥ *n*. *v i* = *True*)›

A comparator is represented as an ordered pair of channel indices. Applying a comparator to a vector will order the values of the two channels so that the channel corresponding to the first index receives the smaller value.

type-synonym *cmp* = ‹*nat* × *nat*›

definition *apply-cmp* :: ‹*cmp* ⇒ *vect* ⇒ *vect*› **where**
```
  ‹apply-cmp c v = (
    let (a, b) = c
    in v(
      a := min (v a) (v b),
      b := max (v a) (v b)
    )
  )›
```

A lower size bound for a sorting network on a given set of input vectors is a number of comparators so that any network that is able to sort every vector of the input set has at least that number of comparators.

definition *lower-size-bound* :: ‹*vect set* ⇒ *nat* ⇒ *bool*› **where**

⟨*lower-size-bound V b* = (∀ *cn*. (∀ *v* ∈ *V*. *mono* (*fold apply-cmp cn v*)) ⟶ *length cn* ≥ *b*)⟩

We are interested in lower size bounds for sorting networks that sort all vectors of a given width.

definition *lower-size-bound-for-width* :: ⟨*nat* ⇒ *nat* ⇒ *bool*⟩ **where**
⟨*lower-size-bound-for-width w b* = *lower-size-bound* {*v*. *fixed-width-vect w v*} *b*⟩

end
theory *Huffman*
imports *Main HOL−Library.Multiset*
begin

class *huffman-algebra* =
fixes *combine* :: ′*a*::*linorder* ⇒ ′*a* ⇒ ′*a* (**infix** ⟨◇⟩ *70*)
assumes *increasing*: ⟨*a* ≤ *a* ◇ *b*⟩
assumes *commutative*: ⟨*a* ◇ *b* = *b* ◇ *a*⟩
assumes *medial*: ⟨(*a* ◇ *b*) ◇ (*c* ◇ *d*) = (*a* ◇ *c*) ◇ (*b* ◇ *d*)⟩
assumes *mono*: ⟨*a* ≤ *b* ⟹ *a* ◇ *c* ≤ *b* ◇ *c*⟩
assumes *assoc-ineq*: ⟨*a* ≤ *c* ⟹ (*a* ◇ *b*) ◇ *c* ≤ *a* ◇ (*b* ◇ *c*)⟩

lemma *mset-tl*: ⟨*xs* ≠ [] ⟹ *mset* (*tl xs*) = *mset xs* − {#*hd xs*#}⟩
by (*cases xs*; *simp*)

lemma *hd-sorted-list-of-multiset*:
assumes ⟨*A* ≠ {#}⟩
shows ⟨*hd* (*sorted-list-of-multiset A*) = *Min-mset A*⟩
by (*metis* (*no-types*, *lifting*) *Min-in Min-le antisym assms finite-set-mset hd-Cons-tl list.set-sel*(*1*) *mset.simps*(*1*) *mset-sorted-list-of-multiset set-ConsD set-mset-eq-empty-iff set-sorted-list-of-multiset sorted.simps*(*2*) *sorted-list-of-multiset-mset sorted-sort*)

lemma *mset-tl-sorted-list-of-multiset*:
assumes ⟨*A* ≠ {#}⟩
shows ⟨*mset* (*tl* (*sorted-list-of-multiset A*)) = *A* − {#*Min-mset A*#}⟩
by (*metis assms hd-sorted-list-of-multiset mset.simps*(*1*) *mset-sorted-list-of-multiset mset-tl*)

lemma *unique-sorted-list-of-multiset*:
assumes ⟨*mset xs* = *A*⟩ ⟨*sorted xs*⟩
shows ⟨*xs* = *sorted-list-of-multiset A*⟩
using *assms*(*1*) *assms*(*2*) *sorted-sort-id* **by** *fastforce*

lemma *tl-sorted-list-of-multiset*:
assumes ⟨*A* ≠ {#}⟩
shows ⟨*tl* (*sorted-list-of-multiset A*) = *sorted-list-of-multiset* (*A* − {#*Min-mset A*#})⟩
proof −
have ⟨*sorted* (*tl* (*sorted-list-of-multiset A*))⟩

by (*metis mset-sorted-list-of-multiset sorted-list-of-multiset-mset sorted-sort sorted-tl*)
thus *?thesis*
by (*simp add*: *assms mset-tl-sorted-list-of-multiset unique-sorted-list-of-multiset*)
qed

datatype *'a expr* =
Val (*the-Val*: *'a*) (⟨-⟩) |
Op (*left-subexpr*: ‹*'a expr*›) (*right-subexpr*: ‹*'a expr*›) (**infix** ⋆ *70*)

fun (**in** *huffman-algebra*) *value-expr* :: ‹*'a expr* ⇒ *'a*› **where**
‹*value-expr* ⟨*a*⟩ = *a*› |
‹*value-expr* (*E* ⋆ *F*) = *value-expr E* ⋄ *value-expr F*›

abbreviation *is-Val* :: ‹*'a expr* ⇒ *bool*› **where**
‹*is-Val E* ≡ ∃ *a*. *E* = ⟨*a*⟩›

abbreviation *is-Op* :: ‹*'a expr* ⇒ *bool*› **where**
‹*is-Op E* ≡ ∃ *L R*. *E* = *L* ⋆ *R*›

lemma *set-expr-nonempty*[*simp*]: ‹*set-expr E* ≠ {}›
by (*induction E*; *auto*)

lemma *set-expr-finite*[*simp*]: ‹*finite* (*set-expr E*)›
by (*induction E*; *auto*)

abbreviation *list-expr* :: ‹*'a expr* ⇒ *'a list*› **where**
‹*list-expr* ≡ *rec-expr* (λ*a*. [*a*]) (λ- -. (@))›

lemma *list-expr-nonempty*[*simp*]: ‹*list-expr E* ≠ []›
by (*induction E*; *auto*)

abbreviation *count-expr* :: ‹*'a expr* ⇒ *nat*› **where**
‹*count-expr E* ≡ *length* (*list-expr E*)›

lemma *count-expr-size*: ‹*2* ∗ *count-expr E* = *Suc* (*size E*)›
by (*induction E*; *auto*)

lemma *count-expr-ge1*[*simp*]: ‹*count-expr E* ≥ *1*›
by (*simp add*: *Suc-leI*)

lemma *count-expr-Op*: ‹*count-expr* (*E* ⋆ *F*) ≥ *2*›
using *count-expr-ge1*[*of E*] *count-expr-ge1*[*of F*]
by (*simp*; *linarith*)

lemma *is-Op-by-count*: ‹*is-Op E* = (*count-expr E* ≥ *2*)›

by (*cases* E; *simp*; *insert count-expr-Op*; *auto*)

lemma *expr-from-list*: ‹*list-expr* $E = [e] \Longrightarrow E = \langle e \rangle$›
by (*cases* E; *simp add*: *append-eq-Cons-conv*)

abbreviation *mset-expr* :: ‹$'a$ *expr* $\Rightarrow$ $'a$ *multiset*› **where**
‹*mset-expr* $E \equiv$ *mset* (*list-expr* E)›

lemma *expr-from-mset*: ‹*mset-expr* $E = \{\#\ a\ \#\} \Longrightarrow E = \langle a \rangle$›
by (*simp add*: *expr-from-list*)

lemma *set-mset-expr*: ‹*set-mset* (*mset-expr* E) = *set-expr* E›
by (*induction* E; *simp*)

abbreviation *hd-expr* :: ‹$'a$ *expr* $\Rightarrow$ $'a$› **where**
‹*hd-expr* $E \equiv$ *hd* (*list-expr* E)›

definition *Min-expr* :: ‹$'a$::*linorder expr* $\Rightarrow$ $'a$› **where**
‹*Min-expr* $E \equiv$ *Min* (*set-expr* E)›

lemma *Min-expr-Val*[*simp*]: ‹*Min-expr* $\langle a \rangle = a$›
unfolding *Min-expr-def*
by *simp*

lemma *Min-expr-Op*: ‹*Min-expr* ($L \star R$) = *min* (*Min-expr* L) (*Min-expr* R)›
unfolding *Min-expr-def*
by (*simp add*: *Min-Un min-def*)

lemma (**in** *huffman-algebra*) *Min-expr-bound*:
‹*Min-expr* $E \leq$ *value-expr* E›
by (*induction* E; *simp add*: *Min-expr-Op*; *insert increasing min.coboundedI1 order-trans*; *blast*)

lemma *Min-expr-mset-cong*: ‹*mset-expr* E = *mset-expr* $F \Longrightarrow$ *Min-expr* E = *Min-expr* F›
unfolding *Min-expr-def set-mset-expr*[*symmetric*] **by** *simp*

lemma *Min-expr-from-mset*: ‹*Min-expr* E = *Min-mset* (*mset-expr* E)›
unfolding *Min-expr-def*
by (*fold set-mset-expr*; *simp*)

fun *tl-expr* :: ‹$'a$ *expr* $\Rightarrow$ $'a$ *expr*› **where**
‹*tl-expr* $\langle a \rangle = \langle a \rangle$› |
‹*tl-expr* ($\langle l \rangle \star R$) = R› |
‹*tl-expr* (($L \star M$) $\star R$) = *tl-expr* ($L \star M$) $\star R$›

lemma *list-tl-expr*: ‹*is-Op* $E \Longrightarrow$ *list-expr* (*tl-expr* E) = *tl* (*list-expr* E)›
by (*induction* E *rule*: *tl-expr.induct*; *simp*)

lemma *same-mset-tl-from-same-mset-mset-hd*:
  assumes ‹*hd-expr* E = *hd-expr* F› ‹*mset-expr* E = *mset-expr* F›
  shows ‹*mset-expr* (*tl-expr* E) = *mset-expr* (*tl-expr* F)›
proof (*cases* ‹*is-Op* E›)
  case *True*
  hence ‹*is-Op* F›
    using *mset-eq-length*[*of* ‹*list-expr* E› ‹*list-expr* F›]
      is-Op-by-count[*of* E] *is-Op-by-count*[*of* F] *assms*(*2*)
    by *auto*
  thus *?thesis*
    using *assms True*
    by (*subst* (*1 2*) *list-tl-expr*; *simp*; *subst* (*1 2*) *mset-tl*; *simp*)
next
  case *False*
  then obtain e **where** ‹$E = \langle e \rangle$›
    using *expr.exhaust-sel* **by** *blast*
  hence ‹$F = E$›
    using *expr.exhaust-sel assms*(*2*) *expr-from-list* **by** *fastforce*
  then show *?thesis*
    by *simp*
qed

inductive *all-subexpr* :: ‹($'a$ *expr* ⇒ *bool*) ⇒ $'a$ *expr* ⇒ *bool*› **where**
  val: ‹P ⟨a⟩ ⟹ *all-subexpr* P ⟨a⟩› |
  op: ‹⟦P ($L \star R$); *all-subexpr* P L; *all-subexpr* P R⟧ ⟹ *all-subexpr* P ($L \star R$)›

declare *all-subexpr.intros*[*intro*] *all-subexpr.cases*[*elim*]

lemma *all-subexpr-top*: ‹*all-subexpr* P E ⟹ P E›
  by *auto*

lemma *all-subexpr-expand*: ‹*all-subexpr* P ($L \star R$) = (P ($L \star R$) ∧ *all-subexpr* P L ∧ *all-subexpr* P R)›
  by *auto*

abbreviation *Min-hd-expr* :: ‹$'a$::*linorder expr* ⇒ *bool*› **where**
  ‹*Min-hd-expr* E ≡ *hd-expr* E = *Min-expr* E›

lemma *min-as-logic*:
  ‹*min* (a::$'a$::*linorder*) $b = c \longleftrightarrow (a = c \wedge a \le b) \vee (b = c \wedge b \le a)$›
  ‹c = *min* (a::$'a$::*linorder*) $b \longleftrightarrow (a = c \wedge a \le b) \vee (b = c \wedge b \le a)$›
  unfolding *min-def* **by** *auto*

lemma *Min-hd-expr-left-subexpr*: ‹*Min-hd-expr* ($L \star R$) ⟹ *Min-hd-expr* L›

by (*induction L*; *auto simp add*: *Min-expr-Op min-as-logic*)

lemma *Min-hd-expr-subexpr-ord*: ‹*Min-hd-expr* $(L \star R) \Longrightarrow$ *Min-expr* $L \leq$ *Min-expr* R›
using *Min-hd-expr-left-subexpr min.orderI* **by** (*fastforce simp add*: *Min-expr-Op*)

lemma *Min-hd-expr-left-subexpr-Min*: ‹*Min-hd-expr* $(L \star R) \Longrightarrow$ *Min-expr* $(L \star R) =$ *Min-expr* L›
by (*induction L*; *auto simp add*: *Min-expr-Op min-as-logic*)

lemma *Min-hd-expr-Min-from-hd-cong*:
assumes ‹*Min-hd-expr E*› ‹*Min-hd-expr F*› ‹*hd-expr E* = *hd-expr F*›
shows ‹*Min-expr E* = *Min-expr F*›
using *assms* **by** *simp*

function *Min-to-hd-subexpr* :: ‹$'a$::*linorder expr* $\Rightarrow$ $'a$::*linorder expr* $\Rightarrow$ $'a$ *expr*›
where
‹*Min-expr* $L \leq$ *Min-expr* $R \Longrightarrow$ *Min-to-hd-subexpr* $L\ R = L \star R$› |
‹$\neg$(*Min-expr* $L \leq$ *Min-expr* R) $\Longrightarrow$ *Min-to-hd-subexpr* $L\ R = R \star L$›
by *auto*
termination by *lexicographic-order*

lemma *Min-to-hd-subexpr-mset*: ‹*mset-expr* (*Min-to-hd-subexpr* $L\ R$) = *mset-expr* $(L \star R)$›
by (*cases* ‹(L, R)› *rule*: *Min-to-hd-subexpr.cases*; *auto*)

lemma *Min-to-hd-subexpr-spec*:
assumes ‹*all-subexpr Min-hd-expr L*› ‹*all-subexpr Min-hd-expr R*›
shows ‹*all-subexpr Min-hd-expr* (*Min-to-hd-subexpr* $L\ R$)›
proof (*cases* ‹*Min-expr* $L \leq$ *Min-expr* R›)
case *True*
have ‹*Min-expr* $(L \star R) =$ *Min-expr* $L \land$ *hd-expr* $(L \star R) =$ *hd-expr* L›
by (*simp add*: *True Min-expr-Op min-def*)
hence ‹*Min-hd-expr* $(L \star R)$›
using *assms* **by** *auto*
thus *?thesis*
using *assms True* **by** *auto*
next
case *False*
hence *False'*: ‹*Min-expr* $R \leq$ *Min-expr* L›
using *linear* **by** *blast*
have ‹*Min-expr* $(R \star L) =$ *Min-expr* $R \land$ *hd-expr* $(R \star L) =$ *hd-expr* R›
by (*auto simp add*: *False' Min-expr-Op min-def*)
hence ‹*Min-hd-expr* $(R \star L)$›
using *assms* **by** *auto*
thus *?thesis*
using *assms False* **by** *auto*
qed

fun *Min-to-hd-expr* :: ‹'a::linorder expr ⇒ 'a expr› **where**
‹*Min-to-hd-expr* ⟨a⟩ = ⟨a⟩› |
‹*Min-to-hd-expr* (L ⋆ R) = *Min-to-hd-subexpr* (*Min-to-hd-expr* L) (*Min-to-hd-expr* R)›

lemma *Min-to-hd-expr-spec*:
‹*all-subexpr Min-hd-expr* (*Min-to-hd-expr* E)›
by (*induction E rule*: *Min-to-hd-expr.induct*;
(*subst Min-to-hd-expr.simps*; *rule Min-to-hd-subexpr-spec*)?;
auto)

lemma *Min-to-hd-expr-mset*: ‹*mset-expr* (*Min-to-hd-expr* E) = *mset-expr* E›
by (*induction E rule*: *Min-to-hd-expr.induct*; *simp add*: *Min-to-hd-subexpr-mset*)

lemma (**in** *huffman-algebra*) *value-Min-to-hd-subexpr*:
‹*value-expr* (*Min-to-hd-subexpr* L R) = *value-expr* L ⋄ *value-expr* R›
by (*metis Min-to-hd-subexpr.simps commutative value-expr.simps*(*2*))

lemma (**in** *huffman-algebra*) *value-Min-to-hd-expr*:
‹*value-expr* (*Min-to-hd-expr* E) = *value-expr* E›
by (*induction E rule*: *Min-to-hd-expr.induct*; *simp add*: *value-Min-to-hd-subexpr*)

abbreviation *tl-Min-hd-expr* :: ‹'a::linorder expr ⇒ bool› **where**
‹*tl-Min-hd-expr* E ≡ *Min-hd-expr* (*tl-expr* E)›

lemma *tl-Min-hd-expr-list-expr-cong*:
assumes ‹*list-expr* E = *list-expr* F›
shows ‹*tl-Min-hd-expr* E = *tl-Min-hd-expr* F›
proof −
have ‹⋀E. *tl* (*list-expr* E) = *list-expr* (*tl-expr* E) ∨ ⟨*the-Val* E⟩ = E›
by (*metis expr.exhaust-sel list-tl-expr*)
then have ‹*list-expr* (*tl-expr* E) = *list-expr* (*tl-expr* F)›
using *assms* **by** (*metis* (*no-types*) *expr.simps*(*7*) *expr-from-list*)
then show *?thesis*
by (*metis Min-expr-mset-cong*)
qed

function *tl-Min-to-hd-subexpr* :: ‹'a::linorder expr ⇒ 'a expr› **where**
‹*tl-Min-to-hd-subexpr* ⟨a⟩ = ⟨a⟩› |
‹*tl-Min-to-hd-subexpr* (⟨l⟩ ⋆ R) = ⟨l⟩ ⋆ R› |
‹*Min-expr* M ≤ r ⟹
tl-Min-to-hd-subexpr ((L ⋆ M) ⋆ ⟨r⟩) = (L ⋆ M) ⋆ ⟨r⟩› |
‹¬(*Min-expr* M ≤ r) ⟹
tl-Min-to-hd-subexpr ((L ⋆ M) ⋆ ⟨r⟩) = (L ⋆ ⟨r⟩) ⋆ M› |
‹*Min-expr* LM ≤ *Min-expr* RM ⟹
tl-Min-to-hd-subexpr ((L ⋆ LM) ⋆ (RM ⋆ R)) = (L ⋆ LM) ⋆ (RM ⋆ R)› |
‹¬(*Min-expr* LM ≤ *Min-expr* RM) ⟹

tl-Min-to-hd-subexpr $((L \star LM) \star (RM \star R)) = (L \star RM) \star (LM \star R)$›
  by (*auto*, *metis tl-expr.cases*)
termination by *lexicographic-order*

lemma *tl-Min-to-hd-subexpr-size*[*simp*]:
  ‹*size* (*tl-Min-to-hd-subexpr E*) = *size E*›
  by (*induction E rule*: *tl-Min-to-hd-subexpr.induct*; *simp*)

fun *tl-Min-to-hd-expr* :: ‹′*a*::*linorder expr* ⇒ ′*a expr*›
  and *helper-tl-Min-to-hd-expr* :: ‹′*a*::*linorder expr* ⇒ ′*a expr*› **where**
    ‹*tl-Min-to-hd-expr E* = *helper-tl-Min-to-hd-expr* (*tl-Min-to-hd-subexpr E*) › |
    ‹*helper-tl-Min-to-hd-expr* ⟨*a*⟩ = ⟨*a*⟩› |
    ‹*helper-tl-Min-to-hd-expr* $(L \star R)$ = *tl-Min-to-hd-expr* $L \star R$›

lemma *tl-Min-to-hd-expr-mset*: ‹*mset-expr* (*tl-Min-to-hd-expr E*) = *mset-expr E*›
proof (*induction* ‹*size E*› *arbitrary*: *E rule*: *less-induct*)
  case *less*
  then show *?case*
    by (*cases E rule*: *tl-Min-to-hd-subexpr.cases*; *simp*)
qed

lemma *tl-Min-to-hd-expr-Min*: ‹*Min-expr* (*tl-Min-to-hd-expr E*) = *Min-expr E*›
  using *tl-Min-to-hd-expr-mset*[*of E*]
  unfolding *Min-expr-def set-mset-expr*[*symmetric*]
  by *simp*

lemma *tl-Min-to-hd-expr-hd*: ‹*hd-expr* (*tl-Min-to-hd-expr E*) = *hd-expr E*›
proof (*induction* ‹*size E*› *arbitrary*: *E rule*: *less-induct*)
  case *less*
  then show *?case*
    by (*cases E rule*: *tl-Min-to-hd-subexpr.cases*; *simp*)
qed

lemma *tl-Min-to-hd-expr-mset-tl*: ‹*mset-expr* (*tl-expr* (*tl-Min-to-hd-expr E*)) = *mset-expr* (*tl-expr E*)›
  by (*subst same-mset-tl-from-same-mset-mset-hd*[*of E* ‹*tl-Min-to-hd-expr E*›];
    simp add: *tl-Min-to-hd-expr-hd tl-Min-to-hd-expr-mset del*: *tl-Min-to-hd-expr.simps*)

lemma *tl-Min-to-hd-expr-Min-tl*: ‹*Min-expr* (*tl-expr* (*tl-Min-to-hd-expr E*)) = *Min-expr* (*tl-expr E*)›
  using *Min-expr-mset-cong tl-Min-to-hd-expr-mset-tl* **by** *blast*

lemma *Min-hd-expr-rewrite-left*:
  assumes ‹*Min-hd-expr* $(L \star R)$› ‹*Min-expr L* = *Min-expr L*′› ‹*Min-hd-expr L*′›
  shows ‹*Min-hd-expr* $(L' \star R)$›
  by (*metis* (*mono-tags*, *lifting*)
    Min-expr-Op Min-hd-expr-left-subexpr assms expr.simps(*8*) *hd-append2 list-expr-nonempty*)

lemma *Min-hd-expr-exchange-right*:

assumes ‹*Min-hd-expr* $((L \star M) \star R)$›
shows ‹*Min-hd-expr* $((L \star R) \star M)$›
using *assms*
by (*simp add*: *Min-expr-Op*; *metis min.commute min.assoc*)

lemma *all-subexpr-Min-hd-expr-exchange-right*:
assumes ‹*all-subexpr Min-hd-expr* $((L \star M) \star R)$›
shows ‹*all-subexpr Min-hd-expr* $((L \star R) \star M)$›
by (*intro all-subexpr.op*; *insert assms Min-hd-expr-exchange-right Min-hd-expr-left-subexpr*; *blast*)

lemma *tl-Min-hd-expr-right-Val*:
assumes ‹*tl-Min-hd-expr L*› ‹*Min-expr* (*tl-expr L*) $\leq r$›
shows ‹*tl-Min-hd-expr* $(L \star \langle r \rangle)$›
using *assms*
by (*cases L*; *simp add*: *Min-expr-Op min-absorb1 dual-order.trans min-def-raw*)

lemma *Min-expr-tl-bound*:
assumes ‹*Min-expr M* $\leq r$›
shows ‹*Min-expr* (*tl-expr* $(L \star M)$) $\leq r$›
using *assms*
by (*cases L*; *simp add*: *Min-expr-Op min-le-iff-disj*)

lemma *tl-Min-hd-expr-right*:
assumes ‹*is-Op L*› ‹*tl-Min-hd-expr L*› ‹*Min-expr* (*tl-expr L*) $\leq$ *Min-expr R*›
shows ‹*tl-Min-hd-expr* $(L \star R)$›
using *assms*
by (*cases L*; *simp add*: *Min-expr-Op min-absorb1 dual-order.trans min-def-raw*)

lemma *is-Op-tl-Min-to-hd-expr*: ‹*is-Op* (*tl-Min-to-hd-expr* $(L \star R)$)›
unfolding *is-Op-by-count*
by (*metis* (*mono-tags*, *lifting*) *count-expr-Op mset-eq-length tl-Min-to-hd-expr-mset*)

lemma *tl-Min-to-hd-expr-spec*:
‹*all-subexpr Min-hd-expr E* $\Longrightarrow$ *tl-Min-hd-expr* (*tl-Min-to-hd-expr E*)›
proof (*induction* ‹*size E*› *arbitrary*: *E rule*: *less-induct*)
case *less*
then show *?case*
proof (*cases E rule*: *tl-Min-to-hd-subexpr.cases*; (*auto*; *fail*)*?*)
case (*3 M r L*)

have *A*: ‹*all-subexpr Min-hd-expr* $(L \star M)$›
using *3 less.prems* **by** *blast*

have *B*: ‹*Min-expr* (*tl-expr* (*tl-Min-to-hd-expr* $(L \star M)$)) $\leq r$›
by (*subst tl-Min-to-hd-expr-Min-tl*; *rule Min-expr-tl-bound*; *simp add*: *3*)

show *?thesis*
by (*simp add*: *3*; *fold tl-Min-to-hd-expr.simps*;

rule tl-Min-hd-expr-right-Val; *insert 3 A B less*; *auto*)
next
case (*4 M r L*)

have ‹*all-subexpr Min-hd-expr* ($L \star \langle r \rangle$)›
using *4 less.prems all-subexpr-Min-hd-expr-exchange-right* **by** *fastforce*
hence *A*: ‹*tl-Min-hd-expr* (*tl-Min-to-hd-expr* ($L \star \langle r \rangle$))›
using *4 less.hyps* **by** *auto*

have *B*: ‹*Min-expr* (*tl-expr* (*tl-Min-to-hd-expr* ($L \star \langle r \rangle$))) $\leq$ *Min-expr M*›
by (*subst tl-Min-to-hd-expr-Min-tl*; *metis 4(1) Min-expr-Val Min-expr-tl-bound linear*)

show *?thesis*
by (*simp add*: *4*; *fold tl-Min-to-hd-expr.simps*; *rule tl-Min-hd-expr-right*;
insert is-Op-tl-Min-to-hd-expr A B; *simp*)
next
case (*5 LM RM L R*)

have *A*: ‹*tl-Min-hd-expr* (*tl-Min-to-hd-expr* ($L \star LM$))›
using *5 less* **by** *auto*

have $*$: ‹*Min-expr LM* $\leq$ *Min-expr RM* $\wedge$ *Min-expr LM* $\leq$ *Min-expr R*›
using *less.prems* **unfolding** *5*
by (*simp add*: *all-subexpr-expand Min-expr-Op*;
insert 5 all-subexpr-top order-trans; *auto simp add*: *min-as-logic*)

have *B*: ‹*Min-expr* (*tl-expr* (*tl-Min-to-hd-expr* ($L \star LM$))) $\leq$ *Min-expr* ($RM \star R$)›
by (*subst tl-Min-to-hd-expr-Min-tl*; *rule Min-expr-tl-bound*; *simp add*: $*$ *Min-expr-Op*)

show *?thesis*
by (*simp add*: *5*; *fold tl-Min-to-hd-expr.simps*; *rule tl-Min-hd-expr-right*;
insert is-Op-tl-Min-to-hd-expr A B; *simp*)
next
case (*6 LM RM L R*)

have $*$: ‹*Min-expr L* $\leq$ *Min-expr RM*›
using *less.prems* **unfolding** *6*
by (*simp add*: *all-subexpr-expand Min-expr-Op*; *insert all-subexpr-top min.orderI*; *fastforce*)

have $**$: ‹*all-subexpr Min-hd-expr* ($L \star RM$)›
by (*rule all-subexpr.op*; *insert* $*$ *6 less.prems*; *auto simp add*: *Min-expr-Op min-def*)

have *A*: ‹*tl-Min-hd-expr* (*tl-Min-to-hd-expr* ($L \star RM$))›
using $**$ *less 6* **by** *auto*

have ∗∗∗: ‹*Min-expr RM* ≤ *Min-expr LM* ∧ *Min-expr RM* ≤ *Min-expr R*›
using *less.prems* **unfolding** *6*
by (*simp add*: *all-subexpr-expand Min-expr-Op*;
insert 6 all-subexpr-top min.orderI; *force*)

have *B*: ‹*Min-expr* (*tl-expr* (*tl-Min-to-hd-expr* (*L* ⋆ *RM*))) ≤ *Min-expr* (*LM* ⋆
R)›
by (*subst tl-Min-to-hd-expr-Min-tl*; *rule Min-expr-tl-bound*; *simp add*: ∗∗∗
Min-expr-Op)

show *?thesis*
by (*simp add*: *6*; *fold tl-Min-to-hd-expr.simps*; *rule tl-Min-hd-expr-right*;
insert is-Op-tl-Min-to-hd-expr A B; *auto*)
qed
qed

lemma (**in** *huffman-algebra*) *value-tl-Min-to-hd-expr*:
‹*all-subexpr Min-hd-expr E* ⟹ *value-expr* (*tl-Min-to-hd-expr E*) ≤ *value-expr E*›
proof (*induction* ‹*size E*› *arbitrary*: *E rule*: *less-induct*)
case *less*
then show *?case*
proof (*cases E rule*: *tl-Min-to-hd-subexpr.cases*; (*auto*; *fail*)*?*)
case (*3 M r L*)
show *?thesis*
by (*simp add*: *3*; *fold tl-Min-to-hd-expr.simps*; *metis* (*no-types*, *lifting*) *3*(*2*)
add-Suc-right
all-subexpr-expand dual-order.strict-trans2 expr.size(*4*) *huffman-algebra.mono*
huffman-algebra-axioms le-add1 less.hyps less.prems lessI value-expr.simps(*2*))
next
case (*4 M r L*)

have ∗: ‹*value-expr* ((*L* ⋆ ⟨*r*⟩) ⋆ *M*) ≤ *value-expr* ((*L* ⋆ *M*) ⋆ ⟨*r*⟩)›
by (*simp*; *metis 4*(*1*) *assoc-ineq commutative huffman-algebra.Min-expr-bound*
huffman-algebra-axioms linear order-trans)

have ∗∗: ‹*value-expr* (*tl-Min-to-hd-expr* (*L* ⋆ ⟨*r*⟩)) ≤ *value-expr* (*L* ⋆ ⟨*r*⟩)›
by (*metis* (*mono-tags*, *lifting*) *4*(*1*) *4*(*2*) *add.right-neutral add-Suc-right*
all-subexpr-Min-hd-expr-exchange-right all-subexpr-expand expr.size(*4*)
le-add1
le-imp-less-Suc less.hyps less.prems tl-Min-to-hd-subexpr.simps(*4*)
tl-Min-to-hd-subexpr-size)

show *?thesis*
by (*simp add*: *4*; *fold tl-Min-to-hd-expr.simps*; *insert* ∗ ∗∗; *simp add*: *dual-order.trans*
mono)
next
case (*5 LM RM L R*)
show *?thesis*

by (*simp add*: *5*; *fold tl-Min-to-hd-expr.simps*; *metis* (*no-types*, *lifting*) *5*(*2*) *Suc-le-eq*
add.right-neutral add-Suc-right all-subexpr-expand expr.size(*4*) *le-add1 less.hyps*
less.prems mono order.strict-iff-order value-expr.simps(*2*))
next
case (*6 LM RM L R*)

have ∗: ‹*value-expr* ((*L* ⋆ *LM*) ⋆ (*RM* ⋆ *R*)) = *value-expr* ((*L* ⋆ *RM*) ⋆ (*LM* ⋆ *R*))›
by (*simp add*: *medial*)

have ‹*all-subexpr Min-hd-expr* (*L* ⋆ *RM*)›
using *less* **unfolding** *6 all-subexpr-expand*
by (*metis* (*mono-tags*, *lifting*) *Min-expr-Op Min-hd-expr-left-subexpr-Min expr.simps*(*8*)
hd-append2 list-expr-nonempty)

hence ∗∗: ‹*value-expr* (*tl-Min-to-hd-expr* (*L* ⋆ *RM*)) ≤ *value-expr* (*L* ⋆ *RM*)›
by (*metis* (*no-types*, *lifting*) *6*(*1*) *6*(*2*) *add.right-neutral add-Suc-right expr.size*(*4*)
le-add1 le-imp-less-Suc less.hyps tl-Min-to-hd-subexpr.simps(*6*) *tl-Min-to-hd-subexpr-size*)

show *?thesis*
by (*simp add*: *6*; *fold tl-Min-to-hd-expr.simps*; *insert* ∗ ∗∗; *simp add*: *mono*)
qed
qed

fun *nest-left-subexpr* :: ‹*'a*::*linorder expr* ⇒ *'a expr*› **where**
‹*nest-left-subexpr* ⟨*a*⟩ = ⟨*a*⟩› |
‹*nest-left-subexpr* (⟨*l*⟩ ⋆ ⟨*r*⟩) = (⟨*l*⟩ ⋆ ⟨*r*⟩)› |
‹*nest-left-subexpr* (⟨*l*⟩ ⋆ (*M* ⋆ *R*)) = (⟨*l*⟩ ⋆ *M*) ⋆ *R*› |
‹*nest-left-subexpr* ((*L* ⋆ *M*) ⋆ *R*) = ((*L* ⋆ *M*) ⋆ *R*)›

lemma *nest-left-subexpr-size*[*simp*]:
‹*size* (*nest-left-subexpr E*) = *size E*›
by (*induction E rule*: *nest-left-subexpr.induct*; *simp*)

lemma *nest-left-subexpr-mset*[*simp*]:
‹*mset-expr* (*nest-left-subexpr E*) = *mset-expr E*›
by (*induction E rule*: *nest-left-subexpr.induct*; *simp*)

fun *nest-left-expr* :: ‹*'a*::*linorder expr* ⇒ *'a expr*›
and *helper-nest-left-expr* :: ‹*'a*::*linorder expr* ⇒ *'a expr*› **where**
‹*nest-left-expr E* = *helper-nest-left-expr* (*nest-left-subexpr E*) › |
‹*helper-nest-left-expr* ⟨*a*⟩ = ⟨*a*⟩› |
‹*helper-nest-left-expr* (*L* ⋆ *R*) = *nest-left-expr L* ⋆ *R*›

lemma *nest-left-expr-list*: ‹*list-expr* (*nest-left-expr E*) = *list-expr E*›
proof (*induction* ‹*size E*› *arbitrary*: *E rule*: *less-induct*)
  case *less*
  then show *?case*
    by (*cases E rule*: *nest-left-subexpr.cases*; *simp*)
qed

inductive *left-nested-expr* :: ‹*'a expr* ⇒ *bool*› **where**
  pair: ‹*left-nested-expr* (⟨*l*⟩ ⋆ ⟨*r*⟩)› |
  nested: ‹*left-nested-expr L* ⟹ *left-nested-expr* (*L* ⋆ *R*)›

declare *left-nested-expr.intros*[*intro*] *left-nested-expr.cases*[*elim*]

lemma *left-nested-nest-left-expr*:
  ‹*is-Op E* ⟹ *left-nested-expr* (*nest-left-expr E*)›
proof (*induction* ‹*size E*› *arbitrary*: *E rule*: *less-induct*)
  case *less*
  then show *?case*
    by (*cases E rule*: *nest-left-subexpr.cases*; *auto*)
qed

lemma (**in** *huffman-algebra*) *value-nest-left-expr*:
  ‹⟦*Min-hd-expr E*⟧ ⟹ *value-expr* (*nest-left-expr E*) ≤ *value-expr E*›
proof (*induction* ‹*size E*› *arbitrary*: *E rule*: *less-induct*)
  case *less*
  then show *?case*
  proof (*cases E rule*: *nest-left-subexpr.cases*; (*auto*; *fail*)*?*)
    case (*3 l M R*)

    have *A*: ‹*l* ≤ *Min-expr M*›
      by (*metis 3 Min-expr-Op Min-expr-Val Min-hd-expr-subexpr-ord less.prems min.bounded-iff*)
    hence ‹*Min-hd-expr* (⟨*l*⟩ ⋆ *M*)› ‹*size* (⟨*l*⟩ ⋆ *M*) < *size E*›
      by (*auto simp add*: *Min-expr-Op min.absorb1 3*)
    hence ‹*value-expr* (*nest-left-expr* (⟨*l*⟩ ⋆ *M*)) ≤ *value-expr* (⟨*l*⟩ ⋆ *M*)›
      using *less.hyps* **by** *fastforce*
    hence ∗: ‹*value-expr* (*nest-left-expr* (⟨*l*⟩ ⋆ *M*) ⋆ *R*) ≤ *value-expr* ((⟨*l*⟩ ⋆ *M*) ⋆ *R*)›
      by (*simp add*: *mono*)

    have ‹*l* ≤ *Min-expr R*›
    by (*metis 3 Min-expr-Op Min-expr-Val Min-hd-expr-left-subexpr-Min less.prems min.cobounded2 min-le-iff-disj*)
    hence ∗∗: ‹*value-expr* ((⟨*l*⟩ ⋆ *M*) ⋆ *R*) ≤ *value-expr* (⟨*l*⟩ ⋆ (*M* ⋆ *R*))›
      using *A*
    by (*metis Min-expr-bound assoc-ineq order-trans value-expr.simps*(*1*) *value-expr.simps*(*2*))

    show *?thesis*
      by (*simp add*: *3*; *fold nest-left-expr.simps*; *insert* ∗ ∗∗; *auto*)

next
case (*4 L M R*)
show *?thesis*
by (*simp add*: *4*; *fold nest-left-expr.simps*; *metis 4 Min-hd-expr-left-subexpr*
Suc-le-eq
add.right-neutral add-Suc-right dual-order.strict-iff-order expr.size(*4*)
le-add1
less.hyps less.prems mono value-expr.simps(*2*))
qed
qed

lemma *Min-hd-expr-sorted-1*:
⟨*Min-hd-expr E* ⟹ *hd-expr E* = *hd* (*sorted-list-of-multiset* (*mset-expr E*))⟩
by (*metis Min-expr-from-mset hd-sorted-list-of-multiset length-0-conv list-expr-nonempty*
mset.simps(*1*) *size-mset*)

lemma *Min-hd-expr-sorted-2*:
assumes ⟨*is-Op E*⟩ ⟨*Min-hd-expr E*⟩ ⟨*tl-Min-hd-expr E*⟩
shows ⟨*hd-expr* (*tl-expr E*) = *hd* (*tl* (*sorted-list-of-multiset* (*mset-expr E*)))⟩
by (*metis Min-expr-from-mset Min-hd-expr-sorted-1 assms list-expr-nonempty*
list-tl-expr mset-tl mset-zero-iff tl-sorted-list-of-multiset)

definition *rearrange-expr* :: ⟨*'a*::*linorder expr* ⇒ *'a expr*⟩ **where**
⟨*rearrange-expr E* = *nest-left-expr* (*tl-Min-to-hd-expr* (*Min-to-hd-expr E*))⟩

lemma *rearrange-expr-mset*: ⟨*mset-expr* (*rearrange-expr E*) = *mset-expr E*⟩
by (*metis Min-to-hd-expr-mset nest-left-expr-list rearrange-expr-def tl-Min-to-hd-expr-mset*)

lemma *Min-hd-rearrange-expr*: ⟨*Min-hd-expr* (*rearrange-expr E*)⟩
by (*metis* (*mono-tags*, *lifting*) *Min-expr-mset-cong Min-to-hd-expr-spec all-subexpr-top*
nest-left-expr-list rearrange-expr-def tl-Min-to-hd-expr-Min tl-Min-to-hd-expr-hd)

lemma *tl-Min-hd-rearrange-expr*: ⟨*tl-Min-hd-expr* (*rearrange-expr E*)⟩
unfolding *rearrange-expr-def*
using *tl-Min-to-hd-expr-spec Min-to-hd-expr-spec nest-left-expr-list tl-Min-hd-expr-list-expr-cong*
by *blast*

lemma *left-nested-rearrange-expr*:
assumes ⟨*is-Op E*⟩
shows ⟨*left-nested-expr* (*rearrange-expr E*)⟩
proof −
have ⟨*is-Op* (*tl-Min-to-hd-expr* (*Min-to-hd-expr E*))⟩
using *assms* **unfolding** *is-Op-by-count*
by (*metis* (*mono-tags*, *lifting*) *Min-to-hd-expr-mset mset-eq-length tl-Min-to-hd-expr-mset*)
thus *?thesis*

unfolding *rearrange-expr-def*
using *left-nested-nest-left-expr* **by** *blast*
qed

lemma (**in** *huffman-algebra*) *value-rearrange-expr*:
⟨*value-expr* (*rearrange-expr E*) ≤ *value-expr E*⟩
unfolding *rearrange-expr-def*
by (*metis* (*mono-tags*, *lifting*) *Min-to-hd-expr-spec all-subexpr-top order-trans tl-Min-to-hd-expr-Min tl-Min-to-hd-expr-hd value-Min-to-hd-expr value-nest-left-expr value-tl-Min-to-hd-expr*)

lemma *hd-list-rearrange-expr*:
⟨*hd* (*list-expr* (*rearrange-expr E*)) = *hd* (*sorted-list-of-multiset* (*mset-expr E*))⟩
by (*metis Min-expr-from-mset Min-hd-rearrange-expr hd-sorted-list-of-multiset list-expr-nonempty*
mset-zero-iff rearrange-expr-mset)

lemma *hd-tl-list-rearrange-expr*:
⟨*hd* (*tl* (*list-expr* (*rearrange-expr E*))) = *hd* (*tl* (*sorted-list-of-multiset* (*mset-expr E*)))⟩
by (*cases E*; (*simp add*: *rearrange-expr-def*; *fail*)?;
metis (*mono-tags*, *lifting*) *Min-hd-expr-sorted-2 Min-hd-rearrange-expr is-Op-by-count list-tl-expr rearrange-expr-mset size-mset tl-Min-hd-rearrange-expr*)

lemma *take-2-from-hds*:
assumes ⟨*length xs* = *length ys*⟩ ⟨*hd xs* = *hd ys*⟩ ⟨*hd* (*tl xs*) = *hd* (*tl ys*)⟩
shows ⟨*take 2 xs* = *take 2 ys*⟩
using *assms*
by (*cases xs*; *simp*; *cases ys*; *simp*; *cases* ⟨*tl xs*⟩; *simp*; *cases* ⟨*tl ys*⟩; *simp*)

lemma *take-2-list-rearrange-expr*:
⟨*take 2* (*list-expr* (*rearrange-expr E*)) = *take 2* (*sorted-list-of-multiset* (*mset-expr E*))⟩
by (*rule take-2-from-hds*; (*simp add*: *hd-list-rearrange-expr hd-tl-list-rearrange-expr*; *fail*)?;
metis mset-sorted-list-of-multiset rearrange-expr-mset size-mset)

inductive *has-subexpr* :: ⟨′*a expr* ⇒ ′*a expr* ⇒ *bool*⟩ **where**
here: ⟨*has-subexpr X X*⟩ |
left: ⟨*has-subexpr X L* ⟹ *has-subexpr X* (*L* ⋆ *R*)⟩ |
right: ⟨*has-subexpr X R* ⟹ *has-subexpr X* (*L* ⋆ *R*)⟩

declare *has-subexpr.intros*[*intro*] *has-subexpr.cases*[*elim*]

lemma *has-subexpr-simp-Op*:
⟨*has-subexpr E* (*L* ⋆ *R*) = (*E* = *L* ⋆ *R* ∨ *has-subexpr E L* ∨ *has-subexpr E R*)⟩

by *blast*

lemma *has-subexpr-Val*: ‹$a \in$ *set-expr E* $=$ *has-subexpr* $\langle a \rangle$ *E*›
by (*induction E*; *auto*)

lemma *mset-has-subexpr*: ‹*has-subexpr X E* $\Longrightarrow$ *mset-expr X* $\subseteq\#$ *mset-expr E*›
by (*induction E*; *auto*; *insert subset-mset.add-increasing subset-mset.add-increasing2*; *fastforce*)

lemma *left-nested-expr-has-hd2-subexpr*:
assumes ‹*left-nested-expr E*› ‹*hd* (*list-expr E*) $=$ *a1*› ‹*hd* (*tl* (*list-expr E*)) $=$ *a2*›
shows ‹*has-subexpr* ($\langle a1 \rangle \star \langle a2 \rangle$) *E*›
using *assms*
proof (*induction E rule*: *left-nested-expr.induct*)
case (*pair l r*)
then show *?case*
by *auto*
next
case (*nested L R*)
then show *?case*
by (*simp*; *metis* (*mono-tags*, *lifting*) *count-expr-ge1 expr.distinct*(*1*) *expr-from-list hd-append2*
left left-nested-expr.cases list.collapse list.size(*3*) *not-one-le-zero*)
qed

function *replace-subexpr* :: ‹$'a$ *expr* $\Rightarrow$ $'a$ *expr* $\Rightarrow$ $'a$ *expr* $\Rightarrow$ $'a$ *expr*› **where**
‹$\neg$*has-subexpr X E* $\Longrightarrow$ *replace-subexpr X Y E* $=$ *E*› |
‹$X = E \Longrightarrow$ *replace-subexpr X Y E* $=$ *Y*› |
‹$[\![X \neq L \star R$; *has-subexpr X L*$]\!] \Longrightarrow$ *replace-subexpr X Y* ($L \star R$) $=$ *replace-subexpr X Y L* $\star$ *R*› |
‹$[\![X \neq L \star R$; $\neg$*has-subexpr X L*; *has-subexpr X R*$]\!] \Longrightarrow$
replace-subexpr X Y ($L \star R$) $=$ *L* $\star$ *replace-subexpr X Y R*›
by (*auto*, *metis has-subexpr.cases*)
termination by *lexicographic-order*

lemma *mset-replace-subexpr*:
‹*has-subexpr X E* $\Longrightarrow$ *mset-expr* (*replace-subexpr X Y E*) $=$ *mset-expr E* $-$ *mset-expr X* $+$ *mset-expr Y*›
by (*induction X Y E rule*: *replace-subexpr.induct*; *auto*;
unfold has-subexpr-simp-Op; *auto simp add*: *mset-has-subexpr*)

lemma (**in** *huffman-algebra*) *value-replace-subexpr*:
‹*value-expr X* $=$ *value-expr Y* $\Longrightarrow$ *value-expr* (*replace-subexpr X Y E*) $=$ *value-expr E*›
by (*induction X Y E rule*: *replace-subexpr.induct*; *auto*)

lemma (**in** *huffman-algebra*) *value-replace-subexpr-increasing*:
‹*value-expr X* $\leq$ *value-expr Y* $\Longrightarrow$ *value-expr E* $\leq$ *value-expr* (*replace-subexpr X Y E*)›

by (*induction X Y E rule*: *replace-subexpr.induct*; *simp add*: *mono*;
metis commutative mono value-expr.simps(*2*))

lemma (**in** *huffman-algebra*) *value-replace-subexpr-decreasing*:
‹*value-expr Y* $\leq$ *value-expr X* $\Longrightarrow$ *value-expr* (*replace-subexpr X Y E*) $\leq$ *value-expr E*›
by (*induction X Y E rule*: *replace-subexpr.induct*; *simp add*: *mono*;
metis commutative mono value-expr.simps(*2*))

lemma *finite-expr-of-size*:
assumes ‹*finite U*›
shows ‹*finite* $\{E.$ *set-expr E* $\subseteq U \wedge$ *size E* $< n\}$›
proof (*induction n*)
case *0*
then show *?case*
by *simp*
next
case (*Suc n*)
have ‹$\{E.$ *set-expr E* $\subseteq U \wedge$ *size E* $<$ *Suc n*$\} \subseteq$
(*Val* ‘ *U*) $\cup$ ($\bigcup L \in \{E.$ *set-expr E* $\subseteq U \wedge$ *size E* $< n\}$.
($\star$) *L* ‘ $\{E.$ *set-expr E* $\subseteq U \wedge$ *size E* $< n\}$)›
proof
fix *E* **assume** *E*: ‹$E \in \{E.$ *set-expr E* $\subseteq U \wedge$ *size E* $<$ *Suc n*$\}$›
hence *PE*: ‹*size E* $<$ *Suc n*› ‹*set-expr E* $\subseteq U$›
by *auto*
show ‹$E \in$ *Val* ‘ *U* $\cup$
($\bigcup L \in \{E.$ *set-expr E* $\subseteq U \wedge$ *size E* $< n\}$.
($\star$) *L* ‘ $\{E.$ *set-expr E* $\subseteq U \wedge$ *size E* $< n\}$)›
by (*cases E*; *insert PE*; *auto*)
qed
then show *?case*
by (*metis* (*no-types*, *lifting*) *Suc.IH assms finite-UN-I finite-Un finite-imageI finite-subset*)
qed

lemma *finite-expr-for-mset*:
‹*finite* $\{E.$ *mset-expr E* $= A\}$›
proof −
have ‹$\{E.$ *mset-expr E* $= A\} \subseteq \{E.$ *set-expr E* $\subseteq$ *set-mset A* $\wedge$ *size E* $<$ *2* $*$ *size A*$\}$›
by (*intro Collect-mono impI*; *fold set-mset-expr*; *auto simp add*: *count-expr-size*)
thus *?thesis*
using *finite-expr-of-size finite-subset* **by** *fastforce*
qed

lemma *ex-expr-for-mset*:
assumes ‹$V \neq \{\#\}$›

shows ‹∃ *E*. *mset-expr E* = *V*›
proof −
obtain *v* **where** *v*: ‹*v* ∈# *V*› **using** *assms*
by *blast*
obtain *L* **where** ‹*mset L* = (*V* − {#*v*#})›
using *ex-mset* **by** *blast*
hence *Lv-mset*: ‹*mset* (*L* @ [*v*]) = *V*›
by (*simp add*: *v*)
obtain *E* **where** *E*: ‹*E* = *foldr* (λ *a b*. ⟨*a*⟩ ⋆ *b*) *L* ⟨*v*⟩›
by *simp*
hence ‹*list-expr E* = *L* @ [*v*]›
unfolding *E* **by** (*induction L*; *simp*)
hence ‹*mset-expr E* = *V*›
using *Lv-mset* **by** *auto*
thus *?thesis*
by *blast*
qed

context *huffman-algebra*
begin

abbreviation *value-bound-mset* :: ‹′*a multiset* ⇒ ′*a*› **where**
‹*value-bound-mset A* ≡ *Min* (*value-expr* ‘ {*E*. *mset-expr E* = *A*})›

lemma *value-bound-singleton*:
‹*value-bound-mset* {# *a* #} = *a*›
proof −
have ‹{*E*. *mset-expr E* = {# *a* #}} = {⟨*a*⟩}›
using *expr-from-mset* **by** *force*
thus *?thesis*
by *simp*
qed

lemma ‹*value-expr E* ≥ *value-bound-mset* (*mset-expr E*)›
by (*intro Min-le*; *insert finite-expr-for-mset*; *blast*)

fun *huffman-step-sorted-list* :: ‹′*a list* ⇒ ′*a multiset*› **where**
‹*huffman-step-sorted-list* (*a1* # *a2* # *as*) = *mset* (*a1* ⋄ *a2* # *as*)› |
‹*huffman-step-sorted-list as* = *mset as*›

abbreviation *huffman-step* :: ‹′*a multiset* ⇒ ′*a multiset*› **where**
‹*huffman-step A* ≡ *huffman-step-sorted-list* (*sorted-list-of-multiset A*)›

lemma *huffman-step-sorted-list-size*:
‹*length as* ≥ *2* ⟹ *Suc* (*size* (*huffman-step-sorted-list as*)) = *length as*›
by (*metis One-nat-def Suc-1 Suc-leD Suc-n-not-le-n huffman-step-sorted-list.elims*

length-Cons
list.size(3) size-mset)

lemma *huffman-step-size[simp]*:
‹*size A* $\geq$ *2* $\Longrightarrow$ *size (huffman-step A)* $<$ *size A*›
by (*metis Suc-n-not-le-n huffman-step-sorted-list-size leI mset-sorted-list-of-multiset size-mset*)

lemma *huffman-step-as-mset-ops*:
assumes ‹*size A* $\geq$ *2*› ‹*a1 # a2 # as = sorted-list-of-multiset A*›
shows ‹*huffman-step A = A* $-$ {# *a1*, *a2* #} + {# *a1* $\diamond$ *a2* #}›
by (*metis add-mset-add-single add-mset-diff-bothsides add-mset-remove-trivial assms(2)*
huffman-step-sorted-list.simps(1) mset.simps(2) mset-sorted-list-of-multiset)

lemma *Min-image-corr-le*:
assumes ‹*finite A*› ‹*A* $\neq$ {}› ‹*finite B*› ‹$\bigwedge a.\ a \in A \Longrightarrow \exists\, b \in B.\ f\ b \leq f\ a$›
shows ‹*Min (f ‘ B)* $\leq$ *Min (f ‘ A)*›
proof −
have ‹$\bigwedge a.\ a \in A \Longrightarrow$ *Min (f ‘ B)* $\leq$ *f a*›
by (*meson Min-le assms(3) assms(4) finite-imageI imageI le-less-trans not-le*)
thus *?thesis*
by (*simp add*: *assms(1) assms(2)*)
qed

lemma *value-bound-via-correspondence*:
assumes ‹*V1* $\neq$ {#}›
‹$\bigwedge$*E1*. *mset-expr E1 = V1* $\Longrightarrow \exists$ *E2*. *mset-expr E2 = V2* $\wedge$ *value-expr E2* $\leq$ *value-expr E1*›
shows ‹*value-bound-mset V2* $\leq$ *value-bound-mset V1*›
by (*intro Min-image-corr-le*; *auto simp add*: *assms finite-expr-for-mset ex-expr-for-mset*)

lemma *combine-step-lower-bound*:
assumes ‹{# *a1*, *a2* #} $\subseteq$# *A*›
shows ‹*value-bound-mset A* $\leq$ *value-bound-mset* (*A* $-$ {# *a1*, *a2* #} + {# *a1* $\diamond$ *a2* #})›
proof (*intro value-bound-via-correspondence*; (*simp*; *fail*)?)
fix *E1* **assume** *E1*: ‹*mset-expr E1 = A* $-$ {#*a1*, *a2*#} + {#*a1* $\diamond$ *a2*#}›
hence ‹*has-subexpr* ⟨*a1* $\diamond$ *a2*⟩ *E1*›
by (*metis add-mset-add-single has-subexpr-Val set-mset-expr union-single-eq-member*)
hence ‹*mset-expr* (*replace-subexpr* ⟨*a1* $\diamond$ *a2*⟩ (⟨*a1*⟩ $\star$ ⟨*a2*⟩) *E1*) = *A*›
by (*simp add*: *mset-replace-subexpr*; *insert E1 assms subset-mset.diff-add*; *fast-force*)
moreover have ‹*value-expr* (*replace-subexpr* ⟨*a1* $\diamond$ *a2*⟩ (⟨*a1*⟩ $\star$ ⟨*a2*⟩) *E1*) = *value-expr E1*›
by (*simp add*: *value-replace-subexpr*)
ultimately show ‹$\exists$ *E2*. *mset-expr E2 = A* $\wedge$ *value-expr E2* $\leq$ *value-expr E1*›
by *auto*
qed

lemma (**in** *huffman-algebra*) *huffman-step-lower-bound*:
  assumes ‹$A \neq \{\#\}$›
  shows ‹*value-bound-mset A* ≤ *value-bound-mset* (*huffman-step A*)›
proof (*cases* ‹*size A* < *2*›)
  case *True*
  then obtain *a* **where** ‹$A = \{\#\ a\ \#\}$›
    using *assms less-2-cases size-1-singleton-mset* **by** *auto*
  then show *?thesis*
    by *auto*
next
  case *False*
  then obtain *a1 a2 as* **where** *V*: ‹*a1* # *a2* # *as* = *sorted-list-of-multiset A*›
    by (*metis One-nat-def Suc-1 assms length-Cons lessI list.size*(*3*) *mset.simps*(*1*)
      mset-sorted-list-of-multiset remdups-adj.cases size-mset)
  hence *a1a2-in-A*: ‹{# *a1*, *a2* #} ⊆# *A*›
   by (*metis empty-le mset.simps*(*2*) *mset-sorted-list-of-multiset mset-subset-eq-add-mset-cancel*)

  show *?thesis*
    using *huffman-step-as-mset-ops*[*of A a1 a2 as*] *False V a1a2-in-A*
      combine-step-lower-bound huffman-algebra-axioms **by** *auto*
qed

lemma *huffman-step-upper-bound*:
  assumes ‹$A \neq \{\#\}$›
  shows ‹*value-bound-mset* (*huffman-step A*) ≤ *value-bound-mset A*›
proof (*intro value-bound-via-correspondence*)
  show ‹$A \neq \{\#\}$›
    by (*simp add*: *assms*)
next
  fix *E1* **assume** *E1*: ‹*mset-expr E1* = *A*›
  show ‹∃ *E2*. *mset-expr E2* = *huffman-step A* ∧ *value-expr E2* ≤ *value-expr E1*›
  proof (*cases* ‹*size A* < *2*›)
    case *True*
    then obtain *a* **where** *A*: ‹$A = \{\#a\#\}$›
      using *assms less-2-cases size-1-singleton-mset* **by** *auto*
    then show *?thesis*
      using *E1* **by** *auto*
  next
    case *False*
    then obtain *a1 a2 as* **where** *V*: ‹*a1* # *a2* # *as* = *sorted-list-of-multiset A*›
        by (*metis Suc-le-length-iff leI mset-sorted-list-of-multiset numeral-2-eq-2*
size-mset)
    obtain *H* **where** *H*: ‹*H* = *rearrange-expr E1*›
      by *simp*

    have *H-is-Op*: ‹*is-Op H*›
      by (*metis E1 False H is-Op-by-count leI rearrange-expr-mset size-mset*)
    have *H-bound*: ‹*value-expr H* ≤ *value-expr E1*›

by (*simp add*: *H value-rearrange-expr*)

have ‹*left-nested-expr H*›
by (*metis E1 False H is-Op-by-count le-less-linear left-nested-rearrange-expr size-mset*)
moreover have ‹*hd* (*list-expr H*) = *a1*›
by (*metis H E1 V hd-list-rearrange-expr list.sel*(*1*))
moreover have ‹*hd* (*tl* (*list-expr H*)) = *a2*›
by (*metis E1 H V hd-tl-list-rearrange-expr list.sel*(*1*) *list.sel*(*3*))
ultimately have *H-subexpr*: ‹*has-subexpr* (⟨*a1*⟩ ⋆ ⟨*a2*⟩) *H*›
by (*simp add*: *left-nested-expr-has-hd2-subexpr*)

then obtain *E2* **where** *E2*: ‹*E2* = *replace-subexpr* (⟨*a1*⟩ ⋆ ⟨*a2*⟩) ⟨*a1* ⋄ *a2*⟩ *H*›
by *simp*
hence ‹*value-expr E2* ≤ *value-expr E1*›
by (*simp add*: *H-bound value-replace-subexpr*)

moreover have ‹*mset-expr E2* = *A* − {# *a1*, *a2* #} + {# *a1* ⋄ *a2* #}›
by (*metis* (*mono-tags*, *lifting*) *E1 E2 H H-subexpr append.simps*(*2*) *append-self-conv2 expr.simps*(*7*) *expr.simps*(*8*) *mset.simps*(*1*) *mset.simps*(*2*) *mset-replace-subexpr rearrange-expr-mset*)
hence ‹*mset-expr E2* = *huffman-step A*›
using *False V huffman-step-as-mset-ops* **by** *auto*

ultimately show *?thesis*
by *blast*
qed
qed

lemma *value-huffman-step*:
‹*value-bound-mset* (*huffman-step A*) = *value-bound-mset A*›
by (*cases* ‹*A* = {#}›; *insert huffman-step-lower-bound huffman-step-upper-bound*; *force*)

function *value-bound-huffman* :: ‹*'a multiset* ⇒ *'a*› **where**
‹*value-bound-huffman A* = (*case size A of*
0 ⇒ *Min* {} |
Suc 0 ⇒ *the-elem* (*set-mset A*) |
Suc (*Suc* -) ⇒ *value-bound-huffman* (*huffman-step A*)
)›
by *pat-completeness auto*
termination
by (*relation* ‹*measure size*›; *simp*;
metis Suc-1 Suc-le-eq less-add-Suc1 local.huffman-step-size plus-1-eq-Suc)

lemma *value-bound-huffman-singleton*:
‹*value-bound-mset* {#*a*#} = *value-bound-huffman* {#*a*#}›
by (*subst value-bound-singleton*; *simp*)

lemma *value-bound-huffman-nonsingleton*:
‹*size A* = *Suc n* ⟹ *value-bound-mset A* = *value-bound-huffman A*›
proof (*induction n arbitrary*: *A*)
case *0*
then obtain *a* **where** ‹*A* = {# *a* #}›
by (*metis One-nat-def size-1-singleton-mset*)
then show *?case*
using *value-bound-huffman-singleton* **by** *blast*
next
case (*Suc n*)
have ‹*size* (*huffman-step A*) = *Suc n*›
by (*metis Suc.prems Suc-1 Suc-le-eq add-diff-cancel-left′ less-add-Suc1 local.huffman-step-sorted-list-size mset-sorted-list-of-multiset plus-1-eq-Suc size-mset*)
hence ‹*value-bound-huffman* (*huffman-step A*) = *value-bound-mset* (*huffman-step A*)›
using *Suc.IH* **by** *auto*
then show *?case*
by (*subst value-bound-huffman.simps*; *simp add*: *Suc.prems value-huffman-step*)
qed

lemma *value-bound-huffman-mset*:
‹*value-bound-mset A* = *value-bound-huffman A*›
by (*cases* ‹*size A*›; *insert value-bound-huffman-nonsingleton*; *auto*)

lemma *value-expr-homo*:
assumes ‹$\bigwedge a\ b.\ f\ (a \diamond b) = f\ a \diamond f\ b$›
shows ‹*value-expr* (*map-expr f E*) = *f* (*value-expr E*)›
using *assms*
by (*induction E*; *auto*)

lemma *value-expr-mono*:
assumes ‹$\bigwedge a\ b.\ f\ (a \diamond b) \le f\ a \diamond f\ b$›
shows ‹*f* (*value-expr E*) ≤ *value-expr* (*map-expr f E*)›
using *assms*
proof (*induction E*; (*simp*; *fail*)*?*)
case (*Op L R*)

have *L*: ‹*f* (*value-expr L*) ≤ *value-expr* (*map-expr f L*)›
and *R*: ‹*f* (*value-expr R*) ≤ *value-expr* (*map-expr f R*)›
using *Op.IH assms* **by** *auto*
hence ‹*f* (*value-expr L*) ⋄ *f* (*value-expr R*) ≤ *f* (*value-expr L*) ⋄ *value-expr* (*map-expr f R*)›
using *local.commutative local.mono* **by** *fastforce*
hence ‹*f* (*value-expr L*) ⋄ *f* (*value-expr R*) ≤ *value-expr* (*map-expr f L*) ⋄ *value-expr* (*map-expr f R*)›

by (*metis L local.mono min.absorb2 min.coboundedI1*)
hence ‹*f* (*value-expr L* $\diamond$ *value-expr R*) $\leq$ *value-expr* (*map-expr f L*) $\diamond$ *value-expr* (*map-expr f R*)›
using *assms dual-order.trans* **by** *blast*
then show *?case*
by *simp*
qed

lemma *mset-expr-map-expr*:
‹*list-expr* (*map-expr f E*) = *map f* (*list-expr E*)›
by (*induction E*; *auto*)

lemma *unmap-list-expr*:
‹*list-expr E* = *map f as* $\Longrightarrow$ $\exists$ E'. *E* = *map-expr f* E' $\wedge$ *list-expr* E' = *as*›
proof (*induction E arbitrary*: *as*)
case (*Val b*)
then obtain *a* **where** ‹*as* = [*a*]›
by *auto*
then show *?case*
by (*metis Val.prems expr.simps*(*7*) *expr.simps*(*9*) *list.sel*(*1*) *list.simps*(*9*))
next
case (*Op L R*)
obtain *ls* **where** *ls*: ‹*ls* = *take* (*length* (*list-expr L*)) *as*›
by *blast*
obtain *rs* **where** *rs*: ‹*rs* = *drop* (*length* (*list-expr L*)) *as*›
by *blast*
have ‹*list-expr L* = *map f ls*›
by (*metis* (*mono-tags*, *lifting*) *Op.prems append-eq-conv-conj expr.simps*(*8*) *ls take-map*)
then obtain L' **where** L': ‹*L* = *map-expr f* L' $\wedge$ *list-expr* L' = *ls*›
using *Op.IH*(*1*) **by** *blast*

have ‹*list-expr R* = *map f rs*›
by (*metis* (*mono-tags*, *lifting*) *Op.prems append-eq-conv-conj drop-map expr.simps*(*8*) *rs*)
then obtain R' **where** R': ‹*R* = *map-expr f* R' $\wedge$ *list-expr* R' = *rs*›
using *Op.IH*(*2*) **by** *blast*

have ‹$L \star R$ = *map-expr f* ($L' \star R'$) $\wedge$ *list-expr* ($L' \star R'$) = *as*›
by (*simp add*: L' R' *ls rs*)
thus *?case*
by *blast*
qed

lemma *unmap-image-mset*:
‹*mset as* = *image-mset f B* $\Longrightarrow$ $\exists$ *bs*. *as* = *map f bs* $\wedge$ *B* = *mset bs*›
proof (*induction as arbitrary*: *B*)
case *Nil*
then show *?case*

by *simp*
next
case (*Cons a as*)
obtain B' *b* **where** $*$: ‹*mset as* = *image-mset f* B' ∧ *a* = *f b* ∧ *B* = *add-mset b* B'›
by (*metis Cons.prems msed-map-invR mset.simps*(*2*))
then obtain *bs* **where** $**$: ‹*as* = *map f bs* ∧ B' = *mset bs*›
using *Cons.IH* **by** *blast*

have ‹*a* # *as* = *map f* (*b* # *bs*) ∧ *B* = *mset* (*b* # *bs*)›
by (*simp add*: $*$ $**$)
then show *?case*
by *metis*
qed

lemma *unmap-mset-expr*:
assumes ‹*mset-expr E* = *image-mset f A*›
shows ‹∃ E'. *E* = *map-expr f* E' ∧ *mset-expr* E' = *A*›
proof −
obtain *es* **where** *es*: ‹*es* = *list-expr E*›
by *simp*
then obtain *as* **where** ‹*es* = *map f as* ∧ *A* = *mset as*›
using *unmap-image-mset*[*of es f A*] *assms*
by *blast*
thus *?thesis*
using *es unmap-list-expr* **by** *fastforce*
qed

lemma *map-expr-inv*: ‹*set-expr E* ⊆ *range f* ⟹ *map-expr f* (*map-expr* (*inv f*) *E*) = *E*›
by (*induction E*; *simp add*: *f-inv-into-f*)

lemma *value-expr-map-expr-inv-homo*:
assumes ‹⋀*a b*. *f* (*a* ⋄ *b*) = *f a* ⋄ *f b*› ‹*set-expr E* ⊆ *range f*›
shows ‹*f* (*value-expr* (*map-expr* (*inv f*) *E*)) = *value-expr E*›
using *assms*
by (*induction E*; *simp add*: *f-inv-into-f*)

lemma *map-expr-inv-homo-image-mset*:
assumes ‹⋀*a b*. *f* (*a* ⋄ *b*) = *f a* ⋄ *f b*› ‹*mset-expr E* = *image-mset f A*›
shows ‹(*map-expr f* (*map-expr* (*inv f*) *E*) = *E*) ∧ (*f* (*value-expr* (*map-expr* (*inv f*) *E*)) = *value-expr E*)›
proof −
have ‹*set-expr E* ⊆ *range f*›
unfolding *set-mset-expr*[*symmetric*]
using *assms* **by** *auto*
thus *?thesis*
by (*simp add*: *assms*(*1*) *map-expr-inv value-expr-map-expr-inv-homo*)
qed

lemma *map-exprs-for-mset*:
‹$\{E.$ *mset-expr* $E =$ *image-mset* f $A\} =$ *map-expr* f ‘ $\{E.$ *mset-expr* $E = A\}$›
proof (*rule*; *rule*)
fix x **assume** ‹$x \in \{E.$ *mset-expr* $E =$ *image-mset* f $A\}$›
thus ‹$x \in$ *map-expr* f ‘ $\{E.$ *mset-expr* $E = A\}$›
using *unmap-mset-expr* **by** *fastforce*
next
fix x **assume** ‹$x \in$ *map-expr* f ‘ $\{E.$ *mset-expr* $E = A\}$›
thus ‹$x \in \{E.$ *mset-expr* $E =$ *image-mset* f $A\}$›
by (*metis* (*mono-tags*, *lifting*) *imageE* *mem-Collect-eq* *mset-expr-map-expr* *mset-map*)
qed

lemma *value-bound-homo*:
assumes ‹$\bigwedge a\ b.\ f\ (a \diamond b) = f\ a \diamond f\ b$› ‹*mono* f› ‹$A \neq \{\#\}$›
shows ‹*value-bound-mset* (*image-mset* f A) $= f$ (*value-bound-mset* A)›
proof −
have ‹*value-expr* ‘ $\{E.$ *mset-expr* $E =$ *image-mset* f $A\} =$
(*value-expr* $\circ$ *map-expr* f) ‘ $\{E.$ *mset-expr* $E = A\}$›
by (*simp add*: *image-comp* *map-exprs-for-mset*)
moreover have ‹(f $\circ$ *value-expr*) ‘ $\{E.$ *mset-expr* $E = A\} =$
(*value-expr* $\circ$ *map-expr* f) ‘ $\{E.$ *mset-expr* $E = A\}$›
using *assms*(*1*) *value-expr-homo* **by** *auto*
ultimately have ‹*value-expr* ‘ $\{E.$ *mset-expr* $E =$ *image-mset* f $A\} =$
f ‘ *value-expr* ‘ $\{E.$ *mset-expr* $E = A\}$›
by (*simp add*: *image-comp*)
hence ‹*value-bound-mset* (*image-mset* f A) $=$ *Min* (f ‘ *value-expr* ‘ $\{E.$ *mset-expr* $E = A\}$)›
by *simp*
moreover have ‹*finite* (*value-expr* ‘ $\{E.$ *mset-expr* $E = A\}$)›
using *finite-expr-for-mset* **by** *blast*
ultimately show *?thesis*
using *mono-Min-commute*[*of* f ‹*value-expr* ‘ $\{E.$ *mset-expr* $E = A\}$›]
by (*simp add*: *assms* *ex-expr-for-mset*)
qed

lemma *Min-corr-image-le*:
assumes ‹*finite* A› ‹$A \neq \{\}$› ‹$\bigwedge a.\ a \in A \Longrightarrow f\ a \leq g\ a$›
shows ‹*Min* (f ‘ A) $\leq$ *Min* (g ‘ A)›
proof −
have ‹$\bigwedge a.\ a \in A \Longrightarrow$ *Min* (f ‘ A) $\leq g\ a$›
using *Min-le-iff* *assms*(*1*) *assms*(*3*) **by** *auto*
thus *?thesis*
by (*simp add*: *assms*(*1*) *assms*(*2*))
qed

lemma *value-bound-mono*:
assumes ‹$\bigwedge a\ b.\ f\ (a \diamond b) \leq f\ a \diamond f\ b$› ‹*mono* f› ‹$A \neq \{\#\}$›

shows ‹f (*value-bound-mset* A) $\leq$ *value-bound-mset* (*image-mset* f A)›
proof −
have ‹*value-expr* ‘ {E. *mset-expr* E = *image-mset* f A} =
(*value-expr* ∘ *map-expr* f) ‘ {E. *mset-expr* E = A}›
by (*simp add*: *image-comp map-exprs-for-mset*)
moreover have ‹*Min* ((f ∘ *value-expr*) ‘ {E. *mset-expr* E = A}) $\leq$
Min ((*value-expr* ∘ *map-expr* f) ‘ {E. *mset-expr* E = A})›
by (*intro Min-corr-image-le*;
simp add: *assms finite-expr-for-mset ex-expr-for-mset value-expr-mono*)
ultimately show *?thesis*
by (*simp add*: *assms ex-expr-for-mset finite-expr-for-mset image-comp mono-Min-commute*)
qed

lemma *value-bound-increasing*:
assumes ‹a ∈# A› ‹$b \geq a$›
shows ‹*value-bound-mset* $A \leq$ *value-bound-mset* (A − {# a #} + {# b #})›
proof (*intro value-bound-via-correspondence*; (*simp*; *fail*)*?*)
fix *E1* **assume** *E1*: ‹*mset-expr E1* = A − {#a#} + {#b#}›

hence ‹*has-subexpr* ⟨b⟩ *E1*›
by (*metis add-mset-add-single has-subexpr-Val set-mset-expr union-single-eq-member*)

hence ‹*mset-expr* (*replace-subexpr* ⟨b⟩ ⟨a⟩ *E1*) = A›
by (*simp add*: *E1 assms*(*1*) *mset-replace-subexpr*)

moreover have ‹*value-expr* (*replace-subexpr* ⟨b⟩ ⟨a⟩ *E1*) $\leq$ *value-expr E1*›
by (*simp add*: *assms*(*2*) *value-replace-subexpr-decreasing*)

ultimately show ‹∃ *E2*. *mset-expr E2* = A ∧ *value-expr E2* $\leq$ *value-expr E1*›
by *blast*
qed

end

end
theory *Sorting-Network*
imports *Main Sorting-Network-Bound HOL−Library.Permutations HOL−Library.Multiset Huffman*
begin

lemma *bool-min-is-conj*[*simp*]: ‹*min* a b = (a ∧ b)›
unfolding *min-def* **by** *auto*

lemma *bool-max-is-disj*[*simp*]: ‹*max* a b = (a ∨ b)›
unfolding *max-def* **by** *auto*

lemma *apply-cmp-logic*:
‹*apply-cmp* c v i = (v i ∧ (i ≠ *fst* c ∨ v (*snd* c)) ∨ (i = *snd* c ∧ v (*fst* c)))›
unfolding *apply-cmp-def Let-def case-prod-unfold*

by *auto*

lemma *apply-cmp-swap-or-id*:
‹*apply-cmp c v* = *v* ∨ *apply-cmp c v* = *Fun.swap* (*fst c*) (*snd c*) *v*›
proof (*cases* ‹*v* (*fst c*) ∧ ¬*v* (*snd c*)›)
case *True*
hence ‹*apply-cmp c v* = *Fun.swap* (*fst c*) (*snd c*) *v*›
by (*simp add*: *apply-cmp-def case-prod-beta′ swap-def*)
thus *?thesis*..
next
case *False*
hence ‹*apply-cmp c v* = *v*›
unfolding *apply-cmp-logic*
by *blast*
thus *?thesis*..
qed

lemma *apply-cmp-same-channels*:
‹*fst c* = *snd c* ⟹ *apply-cmp c v* = *v*›
using *apply-cmp-swap-or-id* **by** *fastforce*

lemma *apply-cmp-fixed-width-snd-oob*:
assumes ‹*fixed-width-vect n v*› ‹*snd c* ≥ *n*›
shows ‹*apply-cmp c v* = *v*›
using *assms*
unfolding *fixed-width-vect-def apply-cmp-logic*
proof (*intro impI ext*)
fix *i*
assume *fixed-width*: ‹∀ *i*≥*n*. *v i* = *True*›
assume ‹*n* ≤ *snd c*›
show ‹(*v i* ∧ (*i* ≠ *fst c* ∨ *v* (*snd c*)) ∨ *i* = *snd c* ∧ *v* (*fst c*)) = *v i*›
proof (*cases* ‹*i* ≥ *n*›)
case *True*
thus *?thesis*
by (*simp add*: *assms*(*2*) *fixed-width*)
next
case *False*
thus *?thesis*
using *assms*(*2*) *fixed-width* **by** *blast*
qed
qed

definition *weight* :: ‹*vect* ⇒ *nat*› **where**
‹*weight v* = *card* (*v* −‘ {*False*})›

lemma ‹*weight* (*apply-cmp c v*) = *weight v*›
proof (*cases* ‹*apply-cmp c v* = *v*›)

  case *True*
  thus *?thesis*
    by *simp*
next
  case *False*
  hence ‹*apply-cmp c v* = *Fun.swap* (*fst c*) (*snd c*) *v*›
    using *apply-cmp-swap-or-id* **by** *blast*
  hence ‹*apply-cmp c v* = *v* ∘ *Fun.swap* (*fst c*) (*snd c*) *id*›
    by (*simp add*: *comp-swap*)
  hence ‹*apply-cmp c v* −‘ {*False*} = (*v* ∘ *Fun.swap* (*fst c*) (*snd c*) *id*) −‘ {*False*}›
    by *simp*
  also have ‹... = *Fun.swap* (*fst c*) (*snd c*) *id* −‘ *v* −‘ {*False*}›
    by *fastforce*
  finally show *?thesis*
    using *card-vimage-inj*[*of* ‹*Fun.swap* (*fst c*) (*snd c*) *id*›] *weight-def* **by** *auto*
qed

lemma *fixed-width-false-set*:
  ‹*fixed-width-vect n v* ⟹ (*v* −‘ {*False*}) ⊆ {..<*n*}›
  unfolding *fixed-width-vect-def*
  using *leI* **by** *blast*

lemma *fixed-width-weight-bound*:
  ‹*fixed-width-vect n v* ⟹ *weight v* ≤ *n*›
  by (*metis fixed-width-false-set card-lessThan weight-def card-mono finite-lessThan*)

lemma *fixed-width-mono-at-weight*:
  assumes ‹*fixed-width-vect n v*› ‹*mono v*› ‹*i* = *weight v*›
  shows ‹*v i* = *True*›
proof (*rule ccontr*)
  assume ‹*v i* ≠ *True*›
  hence ‹(*v* −‘ {*False*}) ⊇ {..*i*}›
    using *assms*(*2*) *monoD* **by** *fastforce*
  hence ‹*weight v* > *i*›
    by (*metis Suc-le-eq assms*(*1*) *card-atMost card-mono finite-lessThan finite-subset fixed-width-false-set weight-def*)
  thus *False*
    using *assms*(*3*) **by** *simp*
qed

lemma *fixed-width-mono-from-weight*:
  assumes ‹*fixed-width-vect n v*› ‹*mono v*›
  shows ‹*v i* = (*i* ≥ *weight v*)›
proof (*cases* ‹*i* ≥ *weight v*›)
  case *True*
  thus *?thesis*
    by (*meson assms*(*1*) *assms*(*2*) *fixed-width-mono-at-weight le-boolD mono-def*)
next
  case *False*

thus *?thesis*
by (*metis* (*full-types*) *assms*(*2*) *fixed-width-vect-def fixed-width-weight-bound le-boolD monoD*)
qed

lemma *weight-inj-on-fixed-width-mono*:
assumes ‹$\bigwedge v.\ v \in V \Longrightarrow$ *mono* $v \wedge$ *fixed-width-vect* $n\ v$›
shows ‹*inj-on weight* V›
proof (*intro inj-onI ext*)
fix $v\ w\ i$ **assume** *vw*: ‹$v \in V$› ‹$w \in V$› ‹*weight* $v =$ *weight* w›
show ‹$v\ i = w\ i$›
by (*metis assms fixed-width-mono-from-weight vw*)
qed

lemma *apply-cmp-fixed-width*:
assumes ‹*fixed-width-vect* $n\ v$›
shows ‹*fixed-width-vect* (*Suc* (*max* n (*max* (*fst* c) (*snd* c)))) (*apply-cmp* $c\ v$)›
unfolding *apply-cmp-logic*
using *assms fixed-width-vect-def* **by** *auto*

lemma *apply-cmp-fixed-width-in-bounds*:
assumes ‹*fixed-width-vect* $n\ v$› ‹*fst* $c < n$› ‹*snd* $c < n$›
shows ‹*fixed-width-vect* n (*apply-cmp* $c\ v$)›
unfolding *apply-cmp-logic*
using *assms fixed-width-vect-def* **by** *auto*

lemma *apply-cn-fixed-width*:
‹*fixed-width-vect* $n\ v \Longrightarrow \exists\, n'.$ *fixed-width-vect* n' (*fold apply-cmp cn* v)›
proof (*induction cn arbitrary*: $n\ v$)
case *Nil*
thus *?case*
by *auto*
next
case (*Cons c cn*)
thus *?case*
by (*metis apply-cmp-fixed-width fold-simps*(*2*))
qed

lemma *weight-one*:
‹*weight* (($\neq$) i) $=$ *1*›
proof $-$
have ‹($\neq$) i $-$‘ {*False*} $= \{i\}$›
by (*rule set-eqI*; *auto*)
thus *?thesis*
by (*simp add*: *weight-def*)
qed

definition *pls-bound* :: ‹*vect set* ⇒ *nat* ⇒ *bool*› **where**
‹*pls-bound V b* = (∀ *cn*. *inj-on weight* (*fold apply-cmp cn* ‘ *V*) ⟶ *length cn* ≥ *b*)›

lemma *pls-bound-implies-lower-size-bound*:
assumes ‹⋀*v*. *v* ∈ *V* ⟹ *fixed-width-vect n v*› ‹*pls-bound V b*›
shows ‹*lower-size-bound V b*›
unfolding *lower-size-bound-def*
proof (*intro allI impI*)
fix *cn* **assume** *cn-sorts*: ‹∀ *v* ∈ *V*. *mono* (*fold apply-cmp cn v*)›

have ‹*inj-on weight* (*fold apply-cmp cn* ‘ *V*)›
proof
fix *v w* **assume** *v-asm*: ‹*v* ∈ *fold apply-cmp cn* ‘ *V*›
and *w-asm*: ‹*w* ∈ *fold apply-cmp cn* ‘ *V*›
and *vw-weight*: ‹*weight v* = *weight w*›
hence ‹*mono v* ∧ *mono w*›
using *cn-sorts* **by** *auto*

moreover obtain *n-v* **where** ‹*fixed-width-vect n-v v*›
by (*metis v-asm apply-cn-fixed-width assms*(*1*) *imageE*)
moreover obtain *n-w* **where** ‹*fixed-width-vect n-w w*›
by (*metis w-asm apply-cn-fixed-width assms*(*1*) *imageE*)
ultimately have ‹⋀*i*. *v i* = (*i* ≥ *weight v*)› ‹⋀*i*. *w i* = (*i* ≥ *weight w*)›
using *fixed-width-mono-from-weight*
by *auto*
thus ‹*v* = *w*›
using *vw-weight* **by** *auto*
qed
thus ‹*b* ≤ *length cn*›
using *assms*(*2*) *pls-bound-def* **by** *blast*
qed

lemma *trivial-bound*: ‹*pls-bound V 0*›
by (*simp add*: *pls-bound-def*)

lemma *unsorted-bound*: ‹¬*inj-on weight V* ⟹ *pls-bound V 1*›
using *pls-bound-def not-less-eq-eq* **by** *fastforce*

lemma *suc-bound*:
assumes *unsorted*: ‹¬*inj-on weight V*› **and** *suc-bounds*: ‹⋀*c*. *pls-bound* (*apply-cmp c* ‘ *V*) *b*›
shows ‹*pls-bound V* (*Suc b*)›
proof (*subst pls-bound-def*; *intro allI impI*)
fix *cn* **assume** *sorts*: ‹*inj-on weight* (*fold apply-cmp cn* ‘ *V*)›
from *this* **and** *unsorted* **have** ‹*cn* ≠ []›
using *pls-bound-def* **by** *auto*
then obtain *c cn′* **where** *cn-cons*: ‹*cn* = *c* # *cn′*›
using *list.exhaust* **by** *blast*

from *this* **and** *sorts* **have** ⟨*inj-on weight* (*fold apply-cmp cn′* ‘ (*apply-cmp c* ‘ *V*))⟩
  by (*simp add*: *image-comp*)
 from *this* **and** *suc-bounds* **have** ⟨*length cn′* ≥ *b*⟩
  using *pls-bound-def cn-cons* **by** *auto*
 thus ⟨*length cn* ≥ *Suc b*⟩
  using *cn-cons* **by** *simp*
qed

lemma *bound-suc*:
 assumes ⟨*pls-bound V* (*Suc b*)⟩
 shows ⟨*pls-bound* (*apply-cmp c* ‘ *V*) *b*⟩
 using *assms*
 unfolding *pls-bound-def*
 by (*metis One-nat-def Suc-eq-plus1 Suc-le-mono fold.simps*(*2*) *image-comp list.size*(*4*))

lemma *bound-unsorted*:
 assumes ⟨*pls-bound V 1*⟩
 shows ⟨¬*inj-on weight V*⟩
 using *assms*
 unfolding *pls-bound-def*
 by (*metis One-nat-def Suc-n-not-le-n fold.simps*(*1*) *id-apply image-subsetI inj-on-subset list.size*(*3*))

lemma *bound-mono-subset*:
 assumes ⟨*pls-bound V b*⟩ ⟨*V* ⊆ *W*⟩
 shows ⟨*pls-bound W b*⟩
 using *pls-bound-def inj-on-subset image-mono*
 by (*metis assms*)

lemma *bound-weaken*:
 ⟨*pls-bound V* (*b* + *d*) ⟹ *pls-bound V b*⟩
 using *pls-bound-def* **by** *auto*

lemma *unsorted-by-card-bound*:
 assumes ⟨⋀*v*. *v* ∈ *V* ⟹ *fixed-width-vect n v*⟩ ⟨*card V* > *n* + *1*⟩
 shows ⟨*pls-bound V 1*⟩
proof(*rule unsorted-bound*; *rule*)
 assume ⟨*inj-on weight V*⟩
 hence ⟨*card* (*weight* ‘ *V*) > *n* + *1*⟩
  using *assms*(*2*) *card-image* **by** *fastforce*
 moreover have ⟨*weight* ‘ *V* ⊆ {..*n*}⟩
  using *assms*(*1*) *fixed-width-weight-bound* **by** *auto*
 hence ⟨*card* (*weight* ‘ *V*) ≤ *n* + *1*⟩
  by (*metis Suc-eq-plus1 card-atMost card-mono finite-atMost*)
 ultimately show *False*
  by *simp*

```
qed

lemma inj-on-invariant-bij-image:
  assumes ‹bij g› ‹⋀a. f (g a) = f a›
  shows ‹inj-on f A = inj-on f (g ‘ A)›
  by (metis assms bij-betw-def inj-on-image-iff inj-on-subset subset-UNIV)

definition apply-perm :: ‹(nat ⇒ nat) ⇒ vect ⇒ vect› where
  ‹apply-perm p v i = v (p i)›

lemma apply-perm-bij:
  assumes ‹bij p›
  shows ‹bij (apply-perm p)›
proof –
  have ‹⋀v i. apply-perm (inv p) (apply-perm p v) i = v i›
    unfolding apply-perm-def
    by (metis assms bij-inv-eq-iff)
  hence ‹apply-perm (inv p) ∘ apply-perm p = id›
    by (auto 0 3)
  moreover have ‹⋀v i. apply-perm p (apply-perm (inv p) v) i = v i›
    unfolding apply-perm-def
    by (metis assms bij-inv-eq-iff)
  hence ‹apply-perm p ∘ apply-perm (inv p) = id›
    by (auto 0 3)
  ultimately show ?thesis
    using o-bij by blast
qed

lemma apply-perm-comp:
  shows ‹apply-perm f ∘ apply-perm g = apply-perm (g ∘ f)›
  unfolding apply-perm-def by auto

lemma apply-perm-weight:
  assumes ‹bij p›
  shows ‹weight (apply-perm p v) = weight v›
  unfolding weight-def apply-perm-def
proof –
  have ‹(λi. v (p i)) –‘ {False} = (v ∘ p) –‘ {False}›
    by fastforce
  also have ‹... = p –‘ v –‘ {False}›
    by (simp add: vimage-comp)
  also have ‹... = (inv p) ‘ v –‘ {False}›
    by (simp add: assms bij-vimage-eq-inv-image)
  finally have ‹card ((λi. v (p i)) –‘ {False}) = card (inv p ‘ v –‘ {False})›
```

by *simp*
also have ‹... = *card* (*v* −‘ {*False*})›
using *card-vimage-inj*[*of* ‹*inv p*› ‹*v* −‘ {*False*}›]
by (*metis assms bij-betw-imp-surj-on card-image inj-on-inv-into top-greatest*)
finally show ‹*card* ((λ*i*. *v* (*p i*)) −‘ {*False*}) = *card* (*v* −‘ {*False*})›**.**
qed

lemma *apply-perm-cmp*:
assumes ‹*bij p*›
shows ‹*apply-cmp c* (*apply-perm p v*) = *apply-perm p* (*apply-cmp* (*p* (*fst c*), *p* (*snd c*)) *v*)›
unfolding *apply-cmp-logic apply-perm-def fst-conv snd-conv*
by (*metis assms bij-pointE*)

lemma *apply-perm-cmp-comp*:
assumes ‹*bij p*›
shows ‹*apply-cmp c* ∘ *apply-perm p* = *apply-perm p* ∘ *apply-cmp* (*p* (*fst c*), *p* (*snd c*))›
by (*rule ext*; *insert assms*; *simp add*: *apply-perm-cmp*)

lemma *permuted-bound*:
assumes ‹*pls-bound V b*› ‹*bij p*›
shows ‹*pls-bound* (*apply-perm p* ‘ *V*) *b*›
using *assms*
proof (*induction b arbitrary*: *p V*)
case *0*
have ‹*pls-bound* (*apply-perm p* ‘ *V*) *0*›
using *trivial-bound***.**
thus *?case***.**
next
case (*Suc b*)

have *V-unsorted*: ‹¬*inj-on weight V*›
by (*metis Suc.prems*(*1*) *bound-unsorted bound-weaken plus-1-eq-Suc*)
have *weight-invariant*: ‹⋀*v*. *weight* (*apply-perm p v*) = *weight v*›
using *Suc.prems*(*2*) *apply-perm-weight* **by** *auto*
have *p-bij*: ‹*bij* (*apply-perm p*)›
by (*simp add*: *Suc.prems*(*2*) *apply-perm-bij*)

show *?case*
proof (*rule suc-bound*)
show ‹¬*inj-on weight* (*apply-perm p* ‘ *V*)›
using *V-unsorted weight-invariant p-bij inj-on-invariant-bij-image* **by** *blast*
next
fix *c*
have ‹*pls-bound* (*apply-perm p* ‘ *apply-cmp* (*p* (*fst c*), *p* (*snd c*)) ‘ *V*) *b*›
using *Suc.IH Suc.prems*(*1*) *Suc.prems*(*2*) *bound-suc* **by** *blast*
thus ‹*pls-bound* (*apply-cmp c* ‘ *apply-perm p* ‘ *V*) *b*›
using *apply-perm-cmp-comp*[*of p c*]

by (*metis Suc.prems*(*2*) *image-comp*)
qed
qed

lemma *permuted-bound-iff*:
assumes ‹*bij p*›
shows ‹*pls-bound V b* = *pls-bound* (*apply-perm p* ‘ *V*) *b*›
proof
show ‹*pls-bound V b* ⟹ *pls-bound* (*apply-perm p* ‘ *V*) *b*›
using *assms permuted-bound* **by** *auto*
next
assume ‹*pls-bound* (*apply-perm p* ‘ *V*) *b*›
hence ‹*pls-bound* (*apply-perm* (*inv p*) ‘ *apply-perm p* ‘ *V*) *b*›
by (*simp add*: *assms bij-betw-inv-into permuted-bound*)
thus ‹*pls-bound V b*›
using *apply-perm-comp*
by (*metis* (*no-types*, *lifting*) *apply-perm-bij assms bij-id bij-is-inj bijection.intro bijection.inv-comp-right image-comp inj-vimage-image-eq inv-id*)
qed

lemma *permuted-bounds-iff*:
assumes ‹*bij p*›
shows ‹*pls-bound V* = *pls-bound* (*apply-perm p* ‘ *V*)›
proof
fix *x*
show ‹*pls-bound V x* = *pls-bound* (*apply-perm p* ‘ *V*) *x*›
using *assms permuted-bound-iff* **by** *blast*
qed

lemma *apply-perm-fixed-width*:
assumes ‹*p permutes* {..<*n*}› ‹*fixed-width-vect n v*›
shows ‹*fixed-width-vect n* (*apply-perm p v*)›
using *assms* **unfolding** *fixed-width-vect-def apply-perm-def permutes-def*
by *simp*

lemma *apply-perm-fixed-width-image*:
assumes ‹*p permutes* {..<*n*}› ‹$\bigwedge$*v*. *v* ∈ *V* ⟹ *fixed-width-vect n v*›
shows ‹$\bigwedge$*v*. *v* ∈ *apply-perm p* ‘ *V* ⟹ *fixed-width-vect n v*›
using *apply-perm-fixed-width assms* **by** *blast*

lemma *apply-cmp-swap*:
‹*apply-cmp* (*prod.swap c*) *v* = *apply-perm* (*Fun.swap* (*fst c*) (*snd c*) *id*) (*apply-cmp c v*)›
unfolding *apply-cmp-logic apply-perm-def fst-swap snd-swap*
by (*metis* (*no-types*, *hide-lams*) *swap-id-eq*)

lemma *apply-cmp-swap-comp*:

⟨*apply-cmp* (*prod.swap* *c*) = *apply-perm* (*Fun.swap* (*fst* *c*) (*snd* *c*) *id*) ∘ *apply-cmp* *c*⟩
by (*rule ext*; *auto simp add*: *apply-cmp-swap*)

lemma *apply-cmp-swap-bound*:
⟨*pls-bound* (*apply-cmp* (*prod.swap* *c*) ‘ *V*) *b* = *pls-bound* (*apply-cmp* *c* ‘ *V*) *b*⟩
proof −
have ⟨*pls-bound* (*apply-cmp* (*prod.swap* *c*) ‘ *V*) *b* =
pls-bound ((*apply-perm* (*Fun.swap* (*fst* *c*) (*snd* *c*) *id*) ∘ *apply-cmp* *c*) ‘ *V*) *b*⟩
using *apply-cmp-swap-comp* **by** *simp*
also have ⟨. . . = *pls-bound* (*apply-cmp* *c* ‘ *V*) *b*⟩
by (*metis image-comp o-bij permuted-bound-iff swap-id-idempotent*)
finally show *?thesis***.**
qed

lemma *suc-bound-noop*:
assumes *unsorted*: ⟨¬*inj-on weight V*⟩
and *suc-bounds*: ⟨⋀*c*. *pls-bound* (*apply-cmp* *c* ‘ *V*) *b* ∨ *pls-bound* (*apply-cmp* *c* ‘ *V*) = *pls-bound* *V*⟩
shows ⟨*pls-bound* *V* (*Suc* *b*)⟩
using *assms*
proof (*induction* *b*)
case *0*
thus *?case*
using *unsorted-bound* **by** *auto*
next
case (*Suc* *b*)
thus *?case*
by (*metis Suc-eq-plus1 bound-weaken suc-bound*)
qed

lemma *ocmp-suc-bound*:
assumes *unsorted*: ⟨¬*inj-on weight V*⟩
and *fixed-width*: ⟨⋀*v*. *v* ∈ *V* ⟹ *fixed-width-vect* *n* *v*⟩
and *suc-bounds*:
⟨⋀*c*. *fst* *c* < *snd* *c* ∧ *snd* *c* < *n* ⟹
pls-bound (*apply-cmp* *c* ‘ *V*) *b* ∨ *pls-bound* (*apply-cmp* *c* ‘ *V*) = *pls-bound* *V*⟩
shows ⟨*pls-bound* *V* (*Suc* *b*)⟩
proof −
have ⟨⋀*c*. *pls-bound* (*apply-cmp* *c* ‘ *V*) *b* ∨ *pls-bound* (*apply-cmp* *c* ‘ *V*) = *pls-bound* *V*⟩
proof −
fix *c*
have ⟨*snd* *c* ≥ *n* ⟹ ∀ *v* ∈ *V*. *apply-cmp* *c* *v* = *v*⟩
using *apply-cmp-fixed-width-snd-oob fixed-width not-less* **by** *blast*
hence ⟨*snd* *c* ≥ *n* ⟹ *pls-bound* (*apply-cmp* *c* ‘ *V*) = *pls-bound* *V*⟩
by *simp*

moreover have
⟨*fst c* $\geq$ *n* $\Longrightarrow$ $\forall$ *v* $\in$ *V*. *apply-cmp c v* = *apply-perm* (*Fun.swap* (*fst c*) (*snd c*) *id*) *v*⟩
by (*metis apply-cmp-fixed-width-snd-oob apply-cmp-swap fixed-width fst-swap swap-commute swap-swap*)
hence ⟨*fst c* $\geq$ *n* $\Longrightarrow$
pls-bound (*apply-cmp c* ‘ *V*) = *pls-bound* (*apply-perm* (*Fun.swap* (*fst c*) (*snd c*) *id*) ‘ *V*)⟩
by *auto*
hence ⟨*fst c* $\geq$ *n* $\Longrightarrow$ *pls-bound* (*apply-cmp c* ‘ *V*) = *pls-bound V*⟩
by (*metis o-bij permuted-bounds-iff swap-id-idempotent*)

moreover have
⟨*fst c* < *snd c* $\wedge$ *snd c* < *n* $\Longrightarrow$
pls-bound (*apply-cmp c* ‘ *V*) *b* $\vee$ *pls-bound* (*apply-cmp c* ‘ *V*) = *pls-bound V*⟩
using *suc-bounds* **by** *simp*

moreover have
⟨*snd c* < *fst c* $\wedge$ *fst c* < *n* $\Longrightarrow$
pls-bound (*apply-cmp c* ‘ *V*) *b* $\vee$ *pls-bound* (*apply-cmp c* ‘ *V*) = *pls-bound V*⟩
using *apply-cmp-swap-bound suc-bounds*
by (*metis apply-cmp-swap-comp image-comp o-bij permuted-bounds-iff prod.exhaust-sel snd-swap swap-id-idempotent swap-simp*)

moreover have ⟨*fst c* = *snd c* $\Longrightarrow$ *pls-bound* (*apply-cmp c* ‘ *V*) = *pls-bound V*⟩
by (*simp add*: *apply-cmp-same-channels*)

ultimately show ⟨*pls-bound* (*apply-cmp c* ‘ *V*) *b* $\vee$ *pls-bound* (*apply-cmp c* ‘ *V*) = *pls-bound V*⟩
using *nat-neq-iff not-less* **by** *blast*
qed
thus *?thesis*
using *suc-bound-noop unsorted* **by** *blast*
qed

definition *redundant-cmp* :: ⟨*cmp* $\Rightarrow$ *vect set* $\Rightarrow$ *bool*⟩ **where**
⟨*redundant-cmp c V* = ($\neg$(($\exists$ *v* $\in$ *V*. *v* (*fst c*) $\wedge$ $\neg$*v* (*snd c*)) $\wedge$ ($\exists$ *v* $\in$ *V*. $\neg$*v* (*fst c*) $\wedge$ *v* (*snd c*))))⟩

lemma *redundant-cmp-id*:
assumes ⟨$\neg$($\exists$ *v* $\in$ *V*. *v* (*fst c*) $\wedge$ $\neg$*v* (*snd c*))⟩
shows ⟨*apply-cmp c* ‘ *V* = *V*⟩
proof –

have ‹$\bigwedge v.\ v \in V \Longrightarrow$ *apply-cmp* $c\ v = v$›
proof
fix $v\ i$ **assume** ‹$v \in V$›
hence ‹$\neg v$ (*fst* c) $\vee$ v (*snd* c)›
using *assms* **by** *blast*
thus ‹*apply-cmp* $c\ v\ i = v\ i$›
using *apply-cmp-logic* **by** *auto*
qed
thus *?thesis*
by *simp*
qed

lemma *redundant-cmp-swap*:
assumes ‹$\neg(\exists\, v \in V.\ \neg v$ (*fst* c) $\wedge$ v (*snd* c))›
shows ‹*apply-cmp* c ‘ V = *Fun.swap* (*fst* c) (*snd* c) ‘ V›
proof −
have ‹$\bigwedge v.\ v \in V \Longrightarrow$ *apply-cmp* $c\ v$ = *Fun.swap* (*fst* c) (*snd* c) v›
proof
fix $v\ i$ **assume** ‹$v \in V$›
hence ‹v (*fst* c) $\vee$ $\neg v$ (*snd* c)›
using *assms* **by** *blast*
thus ‹*apply-cmp* $c\ v\ i$ = *Fun.swap* (*fst* c) (*snd* c) $v\ i$›
using *apply-cmp-logic*
by (*metis swap-apply*(*1*, *3*) *swap-commute*)
qed
thus *?thesis*
by *simp*
qed

lemma *redundant-cmp-id-bound*:
assumes ‹$\neg(\exists\, v \in V.\ v$ (*fst* c) $\wedge$ $\neg v$ (*snd* c))›
shows ‹*pls-bound* (*apply-cmp* c ‘ V) = *pls-bound* V›
by (*simp add*: *assms redundant-cmp-id*)

lemma *redundant-cmp-swap-bound*:
assumes ‹$\neg(\exists\, v \in V.\ \neg v$ (*fst* c) $\wedge$ v (*snd* c))›
shows ‹*pls-bound* (*apply-cmp* c ‘ V) = *pls-bound* V›
by (*metis* (*no-types*, *lifting*) *apply-cmp-swap-comp assms fst-swap image-comp*
o-bij
permuted-bounds-iff redundant-cmp-id snd-swap swap-id-idempotent)

lemma *redundant-cmp-bound*:
assumes ‹*redundant-cmp* $c\ V$›
shows ‹*pls-bound* (*apply-cmp* c ‘ V) = *pls-bound* V›
by (*metis assms redundant-cmp-def redundant-cmp-id-bound redundant-cmp-swap-bound*)

definition *invert-vect* :: ‹*nat* ⇒ *vect* ⇒ *vect*› **where**

‹*invert-vect* n v i = (v i ≠ (i < n))›

lemma *invert-vect-invol*: ‹*invert-vect* n ∘ *invert-vect* n = *id*›
unfolding *invert-vect-def comp-apply* **by** *fastforce*

lemma *invert-vect-bij*: ‹*bij* (*invert-vect* n)›
using *invert-vect-invol o-bij* **by** *blast*

lemma *invert-vect-fixed-width*:
assumes ‹*fixed-width-vect* n v›
shows ‹*fixed-width-vect* n (*invert-vect* n v)›
using *assms fixed-width-vect-def invert-vect-def* **by** *auto*

lemma *invert-false-set*:
assumes ‹*fixed-width-vect* n v›
shows ‹*invert-vect* n v −‘ {*False*} = {..<n} − (v −‘ {*False*})›
proof (*rule set-eqI*)
fix i
have ‹(i ∈ {..<n} − v −‘ {*False*}) = (i ∈ {..<n} ∧ i ∉ v −‘ {*False*})›
by *simp*
also have ‹. . . = (i < n ∧ v i)›
by *simp*
also have ‹. . . = (v i = (i < n))›
using *assms fixed-width-vect-def* **by** *auto*
also have ‹. . . = (¬*invert-vect* n v i)›
by (*simp add*: *invert-vect-def*)
also have ‹. . . = (i ∈ *invert-vect* n v −‘ {*False*})›
by *simp*
finally show ‹(i ∈ *invert-vect* n v −‘ {*False*}) = (i ∈ {..<n} − v −‘ {*False*})›
by *simp*
qed

lemma *invert-vect-weight*:
assumes ‹*fixed-width-vect* n v›
shows ‹*weight* (*invert-vect* n v) = n − *weight* v›
unfolding *weight-def*
by (*metis assms card-Diff-subset card-lessThan finite-lessThan finite-subset fixed-width-false-set invert-false-set*)

lemma *inj-on-inj-inj-image*:
assumes ‹*inj-on* g A› ‹*inj-on* h (f ‘ A)› ‹⋀a. a ∈ A ⟹ f (g a) = h (f a)›
shows ‹*inj-on* f A = *inj-on* f (g ‘ A)›
proof
assume *fwd*: ‹*inj-on* f A›
show ‹*inj-on* f (g ‘ A)›
proof
fix x y **assume** x: ‹x ∈ g ‘ A› **and** y: ‹y ∈ g ‘ A› **and** *f-eq*: ‹f x = f y›
thus ‹x = y›
by (*metis* (*full-types*) *fwd assms*(*2*) *assms*(*3*) *image-iff inj-on-eq-iff*)

```
  qed
next
  assume bwd: ‹inj-on f (g ` A)›
  show ‹inj-on f A›
  proof
    fix x y assume x: ‹x ∈ A› and y: ‹y ∈ A› and f-eq: ‹f x = f y›
    thus ‹x = y›
      by (metis assms(1) assms(3) bwd imageI inj-on-contraD)
  qed
qed

lemma sub-inj-on: ‹inj-on ((-) n) {0..n :: nat}›
  by (metis atLeast0AtMost atMost-iff diff-diff-cancel inj-onI)

lemma invert-vect-cmp:
  assumes ‹fst c < n› ‹snd c < n›
  shows ‹apply-cmp c (invert-vect n v) = invert-vect n (apply-cmp (prod.swap c)
v)›
  unfolding apply-cmp-logic invert-vect-def fst-swap snd-swap
  using assms by auto

lemma invert-vect-cmp-comp:
  assumes ‹fst c < n› ‹snd c < n›
  shows ‹apply-cmp c ∘ invert-vect n = invert-vect n ∘ apply-cmp (prod.swap c)›
  by (rule ext; insert assms; simp add: invert-vect-cmp)

lemma inverted-bound:
  assumes ‹⋀v. v ∈ V ⟹ fixed-width-vect n v› ‹pls-bound V b›
  shows ‹pls-bound (invert-vect n ` V) b›
  using assms
proof (induction b arbitrary: n V)
  case 0
  have ‹pls-bound (invert-vect n ` V) 0›
    using trivial-bound.
  thus ?case.
next
  case (Suc b)

  show ?case
  proof (rule ocmp-suc-bound[where n=n])
    have ‹weight ` V ⊆ {0..n}›
      by (simp add: Suc.prems(1) fixed-width-weight-bound image-subsetI)
    hence ‹inj-on ((-) n) (weight ` V)›
      by (meson inj-on-subset sub-inj-on)
    thus ‹¬inj-on weight (invert-vect n ` V)›
      using inj-on-inj-inj-image[of ‹invert-vect n› V ‹λx. n - x› weight]
    by (metis Suc.prems(1) Suc.prems(2) bij-betw-def bound-unsorted bound-weaken
inj-on-subset
          invert-vect-bij invert-vect-weight plus-1-eq-Suc subset-UNIV)
```

next
fix v
show ‹$v \in$ *invert-vect* n ‘ $V \Longrightarrow$ *fixed-width-vect* n v›
using *Suc.prems*(*1*) *invert-vect-fixed-width* **by** *auto*
next
fix c **assume** ‹*fst* $c <$ *snd* $c \wedge$ *snd* $c < n$›
hence *bound-c*: ‹*fst* $c < n$› ‹*snd* $c < n$›
using *less-trans* **by** *auto*

have ‹*pls-bound* (*apply-cmp* (*prod.swap* c) ‘ V) b›
using *Suc.prems*(*2*) *bound-suc*
by *simp*
hence ‹*pls-bound* (*invert-vect* n ‘ *apply-cmp* (*prod.swap* c) ‘ V) b›
using *Suc.prems*(*1*) *Suc.IH apply-cmp-fixed-width-in-bounds bound-c*
by (*metis* (*no-types*, *lifting*) *fst-swap imageE snd-swap*)

thus ‹*pls-bound* (*apply-cmp* c ‘ *invert-vect* n ‘ V) $b \vee$
pls-bound (*apply-cmp* c ‘ *invert-vect* n ‘ V) = *pls-bound* (*invert-vect* n ‘ V)›
by (*simp add*: *bound-c image-comp invert-vect-cmp-comp*)
qed
qed

definition *pruned-bound* :: ‹*vect set* ⇒ *nat* ⇒ *nat* ⇒ *bool*› **where**
‹*pruned-bound* V i b = (((≠) i) $\in V \wedge$ *pls-bound* $\{v \in V. \neg v\ i\}$ b)›

lemma *pls-bound-from-pruned-bound*:
assumes ‹*pruned-bound* V i b›
shows ‹*pls-bound* V b›
by (*rule bound-mono-subset*[*of* ‹$\{v \in V. \neg v\ i\}$›]; *insert assms pruned-bound-def*; *blast*)

lemma *apply-cmp-sorted*:
assumes ‹$\neg v$ (*fst* c)›
shows ‹*apply-cmp* c v = v›
unfolding *apply-cmp-logic*
using *assms* **by** *auto*

lemma *apply-cmp-sorted-bound*:
assumes ‹$\bigwedge v.\ v \in V \Longrightarrow \neg v$ (*fst* c)›
shows ‹*pls-bound* (*apply-cmp* c ‘ V) = *pls-bound* V›
using *apply-cmp-sorted* **by** (*simp add*: *assms*)

lemma *apply-cmp-sorted-pruned-bound*:
assumes ‹*pruned-bound* V (*fst* c) b›
shows ‹*pruned-bound* (*apply-cmp* c ‘ V) (*fst* c) b›
unfolding *pruned-bound-def*
proof

show ‹(≠) (*fst c*) ∈ *apply-cmp c* ‘ *V*›
  using *assms* **unfolding** *pruned-bound-def apply-cmp-logic* **by** *auto*
next
  have ‹*pls-bound* {*v* ∈ *V*. ¬*v* (*fst c*)} *b*›
    using *assms pruned-bound-def* **by** *blast*
  hence ‹*pls-bound* (*apply-cmp c* ‘ {*v* ∈ *V*. ¬*v* (*fst c*)}) *b*›
    by (*metis* (*mono-tags*, *lifting*) *apply-cmp-sorted-bound mem-Collect-eq*)
  moreover have ‹(*apply-cmp c* ‘ {*v* ∈ *V*. ¬*v* (*fst c*)}) ⊆ {*v* ∈ *apply-cmp c* ‘ *V*. ¬*v* (*fst c*)}›
    by (*metis* (*mono-tags*, *lifting*) *apply-cmp-sorted image-eqI image-subsetI mem-Collect-eq*)
  ultimately show ‹*pls-bound* {*v* ∈ *apply-cmp c* ‘ *V*. ¬*v* (*fst c*)} *b*›
    using *bound-mono-subset* **by** *auto*
qed

lemma *apply-cmp-rev-sorted*:
  assumes ‹¬*v* (*snd c*)›
  shows ‹*apply-cmp c v* = *v* ∘ (*Fun.swap* (*fst c*) (*snd c*) *id*)›
  unfolding *apply-cmp-logic comp-apply*
  using *assms* **by** (*auto simp add*: *swap-id-eq*)

lemma *apply-cmp-rev-sorted-bound*:
  assumes ‹⋀*v*. *v* ∈ *V* ⟹ ¬*v* (*snd c*)›
  shows ‹*pls-bound* (*apply-cmp c* ‘ *V*) = *pls-bound V*›
  using *apply-cmp-rev-sorted*
  by (*metis* (*no-types*, *hide-lams*) *UNIV-I apply-cmp-sorted-bound apply-cmp-swap-comp assms*
    bij-betw-id bij-betw-swap-iff fst-swap image-comp permuted-bounds-iff)

lemma *apply-cmp-rev-sorted-pruned-bound*:
  assumes ‹*pruned-bound V* (*snd c*) *b*›
  shows ‹*pruned-bound* (*apply-cmp c* ‘ *V*) (*fst c*) *b*›
  unfolding *pruned-bound-def*
proof
  have ‹(≠) (*snd c*) ∈ *V*›
    using *assms* **unfolding** *pruned-bound-def* **by** *simp*
  moreover have ‹*apply-cmp c* ((≠) (*snd c*)) = ((≠) (*fst c*))›
    unfolding *apply-cmp-logic* **by** *auto*
  ultimately show ‹(≠) (*fst c*) ∈ *apply-cmp c* ‘ *V*›
    using *image-iff*[*of* ‹(≠) (*fst c*)› ‹*apply-cmp c*› *V*] **by** *fastforce*
next
  have ‹*pls-bound* {*v* ∈ *V*. ¬*v* (*snd c*)} *b*›
    using *assms pruned-bound-def* **by** *blast*
  hence ‹*pls-bound* (*apply-cmp c* ‘ {*v* ∈ *V*. ¬*v* (*snd c*)}) *b*›
    by (*metis* (*mono-tags*, *lifting*) *apply-cmp-rev-sorted-bound mem-Collect-eq*)
  moreover have ‹(*apply-cmp c* ‘ {*v* ∈ *V*. ¬*v* (*snd c*)}) ⊆ {*v* ∈ *apply-cmp c* ‘ *V*. ¬*v* (*fst c*)}›
    by (*metis* (*mono-tags*, *lifting*) *apply-cmp-logic imageI image-Collect-subsetI mem-Collect-eq*)
  ultimately show ‹*pls-bound* {*v* ∈ *apply-cmp c* ‘ *V*. ¬*v* (*fst c*)} *b*›

using *bound-mono-subset* **by** *auto*
qed

lemma *apply-cmp-other-pruned-bound*:
assumes ‹$i \neq$ *fst* c› ‹$i \neq$ *snd* c› ‹*pruned-bound* V i (*Suc* b)›
shows ‹*pruned-bound* (*apply-cmp* c ‘ V) i b›
unfolding *pruned-bound-def*
proof
have ‹$(\neq)$ $i \in V$›
using *assms* **unfolding** *pruned-bound-def* **by** *simp*
moreover have ‹*apply-cmp* c $((\neq)$ $i) = ((\neq)$ $i)$›
unfolding *apply-cmp-logic* **using** *assms* **by** *auto*
ultimately show ‹$(\neq)$ $i \in$ *apply-cmp* c ‘ V›
using *image-iff*[*of* ‹$(\neq)$ i› ‹*apply-cmp* c› V] **by** *fastforce*
next
have ‹*pls-bound* $\{v \in V.\ \neg v\ i\}$ (*Suc* b)›
using *assms pruned-bound-def* **by** *blast*
hence ‹*pls-bound* (*apply-cmp* c ‘ $\{v \in V.\ \neg v\ i\}$) b›
using *bound-suc* **by** *blast*
moreover have ‹$\bigwedge v.$ *apply-cmp* c v $i = v$ i›
unfolding *apply-cmp-logic*
by (*simp add*: *assms*)
hence ‹(*apply-cmp* c ‘ $\{v \in V.\ \neg v\ i\}$) $= \{v \in$ *apply-cmp* c ‘ $V.\ \neg v\ i\}$›
by *blast*
ultimately show ‹*pls-bound* $\{v \in$ *apply-cmp* c ‘ $V.\ \neg v\ i\}$ b›
by *auto*
qed

lemma *apply-cmp-other-pruned-zero-bound*:
assumes ‹$i \neq$ *fst* c› ‹$i \neq$ *snd* c› ‹*pruned-bound* V i *0*›
shows ‹*pruned-bound* (*apply-cmp* c ‘ V) i *0*›
unfolding *pruned-bound-def*
proof
have ‹$(\neq)$ $i \in V$›
using *assms* **unfolding** *pruned-bound-def* **by** *simp*
moreover have ‹*apply-cmp* c $((\neq)$ $i) = ((\neq)$ $i)$›
unfolding *apply-cmp-logic* **using** *assms* **by** *auto*
ultimately show ‹$(\neq)$ $i \in$ *apply-cmp* c ‘ V›
using *image-iff*[*of* ‹$(\neq)$ i› ‹*apply-cmp* c› V] **by** *fastforce*
next
show ‹*pls-bound* $\{v \in$ *apply-cmp* c ‘ $V.\ \neg v\ i\}$ *0*›
using *trivial-bound*.
qed

definition *pruned-bounds* :: ‹*vect set* $\Rightarrow$ (*nat* $\rightharpoonup$ *nat*) $\Rightarrow$ *bool*› **where**
‹*pruned-bounds* V B = ($\forall i \in$ *dom* B. *pruned-bound* V i (*the* (B i)))›

definition *combine-bounds* :: ‹*nat option* ⇒ *nat option* ⇒ *nat option*› **where**
‹*combine-bounds a b* = (*case a of*
None ⇒ *b* | *Some a'* ⇒ (*case b of*
None ⇒ *Some a'* | *Some b'* ⇒ *Some* (*max a' b'*)))›

lemma *combine-bounds-either*:
‹*combine-bounds a b* = *a* ∨ *combine-bounds a b* = *b*›
by (*simp add*: *combine-bounds-def max-def option.case-eq-if*)

definition *pruned-bounds-suc* :: ‹*cmp* ⇒ (*nat* ⇀ *nat*) ⇒ (*nat* ⇀ *nat*)› **where**
‹*pruned-bounds-suc c bs* =
(*map-option nat.pred* ∘ *bs*)(
snd c := *None*,
fst c := *combine-bounds* (*bs* (*fst c*)) (*bs* (*snd c*)))›

lemma *the-map-option-dom*[*simp*]:
‹*x* ∈ *dom f* ⟹ *the* (*map-option g* (*f x*)) = *g* (*the* (*f x*))›
by (*simp add*: *domIff option.map-sel*)

lemma *apply-cmp-pruned-bounds*:
assumes ‹*pruned-bounds V B*›
shows ‹*pruned-bounds* (*apply-cmp c* ‘ *V*) (*pruned-bounds-suc c B*)›
unfolding *pruned-bounds-def*
proof
fix *i* **assume** *in-dom*: ‹*i* ∈ *dom* (*pruned-bounds-suc c B*)›
show ‹*pruned-bound* (*apply-cmp c* ‘ *V*) *i* (*the* (*pruned-bounds-suc c B i*))›
proof (*cases* ‹*i* = *fst c* ∨ *i* = *snd c*›)
case *True*
hence *i-fst*: ‹*i* = *fst c*›
by (*metis domIff fun-upd-def in-dom pruned-bounds-suc-def*)

show *?thesis*
proof (*cases* ‹*fst c* = *snd c*›)
case *True*
then show *?thesis*
unfolding *pruned-bounds-suc-def*
by (*simp add*: *i-fst True*;*metis True apply-cmp-rev-sorted-pruned-bound assms*
combine-bounds-either domD domI fun-upd-same i-fst in-dom pruned-bounds-def
pruned-bounds-suc-def)
next
case *False*
show *?thesis*
unfolding *pruned-bounds-suc-def*
by (*metis apply-cmp-rev-sorted-pruned-bound apply-cmp-sorted-pruned-bound assms*(*1*)
combine-bounds-either domIff fun-upd-same i-fst in-dom pruned-bounds-def
pruned-bounds-suc-def)
qed

next
  case *False*
  hence *in-dom'*: ‹$i \in dom\ B$›
   by (*metis in-dom comp-apply domIff fun-upd-apply option.simps*(*8*) *pruned-bounds-suc-def*)
  show *?thesis*
    unfolding *pruned-bounds-suc-def*
    using *False in-dom'*
   by (*simp*; *metis apply-cmp-other-pruned-bound apply-cmp-other-pruned-zero-bound assms*(*1*)
      nat.split-sels(*2*) *pred-def pruned-bounds-def*)
qed
qed

definition *mset-ran* :: ‹$('a \rightharpoonup 'b) \Rightarrow 'b\ multiset$› **where**
‹$mset\text{-}ran\ m = \{\#the\ (m\ a).\ a \in\#\ mset\text{-}set\ (dom\ m)\#\}$›

lemma *size-mset-ran*:
‹$finite\ (dom\ m) \Longrightarrow size\ (mset\text{-}ran\ m) = card\ (dom\ m)$›
unfolding *mset-ran-def*
by *simp*

lemma *image-mset-set-cong*:
assumes ‹$\bigwedge x.\ x \in A \Longrightarrow f\ x = g\ x$›
shows ‹$image\text{-}mset\ f\ (mset\text{-}set\ A) = image\text{-}mset\ g\ (mset\text{-}set\ A)$›
by (*metis assms finite-set-mset-mset-set image-mset-cong image-mset-empty mset-set.infinite*)

lemma *comp-push-lambda*: ‹$f \circ (\lambda x.\ g\ x) = (\lambda x.\ f\ (g\ x))$›
by *auto*

lemma *mset-ran-map-option*:
‹$mset\text{-}ran\ (map\text{-}option\ f \circ m) = image\text{-}mset\ f\ (mset\text{-}ran\ m)$›
proof −
have ‹$\bigwedge a.\ a \in dom\ m \Longrightarrow the\ (map\text{-}option\ f\ (m\ a)) = f\ (the\ (m\ a))$›
  by *simp*
hence
  ‹$\{\#the\ (map\text{-}option\ f\ (m\ a)).\ a \in\#\ mset\text{-}set\ (dom\ m)\#\} =$
    $\{\#f\ (the\ (m\ a)).\ a \in\#\ mset\text{-}set\ (dom\ m)\#\}$›
  by (*meson image-mset-set-cong*)
thus *?thesis*
  unfolding *mset-ran-def*
  by (*simp add*: *image-mset.compositionality*; *simp add*: *comp-push-lambda*)
qed

abbreviation *mset-option* :: ‹$'a\ option \Rightarrow 'a\ multiset$› **where**
‹$mset\text{-}option \equiv case\text{-}option\ \{\#\}\ (\lambda x.\ \{\#x\#\})$›

lemma *mset-option-Some*:

‹mset-option (Some x) = {#x#}›
  by *simp*

lemma *image-mset-set-diff*:
  assumes ‹finite A›
  shows ‹image-mset f (mset-set (A − B)) =
    image-mset f (mset-set A) − image-mset f (mset-set (A ∩ B))›
  using *assms*
  by (*metis Diff-Compl Diff-Diff-Int Diff-eq image-mset-Diff inf-le1 mset-set-Diff subset-imp-msubset-mset-set*)

lemma *mset-ran-upd-None*:
  assumes ‹finite (dom m)›
  shows ‹mset-ran (m(x := None)) = mset-ran m − mset-option (m x)›
proof −
  have ‹dom (m(x := None)) = dom m − {x}›
    by *simp*
  hence ‹mset-ran (m(x := None)) = {#the ((m(x := None)) a). a ∈# mset-set (dom m − {x})#}›
    unfolding *mset-ran-def*
    by *simp*
  also have ‹. . . = {#the (m a). a ∈# mset-set (dom m − {x})#}›
    using *image-mset-set-cong*[*of* ‹dom m − {x}› ‹λa. the ((m(x := None)) a)› ‹λa. the (m a)›]
    by *simp*
  also have ‹. . . = {#the (m a). a ∈# mset-set (dom m)#} − {#the (m a). a ∈# mset-set (dom m ∩ {x})#}›
    using *image-mset-set-diff*[*of* ‹dom m› - ‹{x}›] *assms*
    by *blast*
  also have ‹. . . = mset-ran m − mset-option (m x)›
    unfolding *mset-ran-def*
    by (*simp add*: *domIff option.case-eq-if*)
  finally show *?thesis*
    by *simp*
qed

lemma *mset-ran-upd-new*:
  assumes ‹finite (dom m)› ‹x ∉ dom m›
  shows ‹mset-ran (m(x ↦ y)) = mset-ran m + {#y#}›
proof −
  have ‹mset-ran (m(x ↦ y)) = {#the ((m(x ↦ y)) a). a ∈# mset-set (dom m ∪ {x})#}›
    unfolding *mset-ran-def*
    by *simp*
  also have ‹. . . = {#the ((m(x ↦ y)) a). a ∈# mset-set (dom m)#} +
      {#the ((m(x ↦ y)) a). a ∈# mset-set {x}#}›
    using *assms*(*1*) *assms*(*2*) **by** *auto*
  also have ‹. . . = mset-ran m + {#the ((m(x ↦ y)) a). a ∈# mset-set {x}#}›
    unfolding *mset-ran-def*

by (*metis* (*mono-tags*, *lifting*) *assms*(*2*) *fun-upd-other image-mset-set-cong*)
finally show *?thesis*
by *simp*
qed

lemma *mset-ran-upd-new-option*:
assumes ‹*finite* (*dom m*)› ‹$x \notin$ *dom m*›
shows ‹*mset-ran* ($m(x := y)$) = *mset-ran m* + *mset-option y*›
proof (*cases* ‹y = *None*›)
case *True*
then show *?thesis*
by (*metis add.comm-neutral assms*(*2*) *domIff fun-upd-idem-iff option.simps*(*4*))
next
case *False*
then show *?thesis*
using *assms mset-ran-upd-new* **by** *fastforce*
qed

lemma *mset-ran-upd*:
assumes ‹*finite* (*dom m*)›
shows ‹*mset-ran* ($m(x := y)$) = *mset-ran m* − *mset-option* (*m x*) + *mset-option y*›
proof −
have ‹*mset-ran* ($m(x :=$ *None*, $x := y)$) = *mset-ran m* − *mset-option* (*m x*) + *mset-option y*›
by (*metis assms domIff dom-fun-upd finite-Diff fun-upd-same mset-ran-upd-None mset-ran-upd-new-option*)
thus *?thesis*
by *simp*
qed

lemma *mset-ran-Melem*:
assumes ‹*finite* (*dom B*)› ‹$x \in$ *dom B*›
shows ‹*the* (*B x*) ∈# *mset-ran B*›
proof −
have ‹*x* ∈# *mset-set* (*dom B*)›
using *assms* **by** *simp*
thus *?thesis*
unfolding *mset-ran-def*
by *simp*
qed

lemma *mset-ran-pair*:
assumes ‹*finite* (*dom B*)› ‹$x \in$ *dom B*› ‹$y \in$ *dom B*› ‹$x \neq y$›
shows ‹{#*the* (*B x*), *the* (*B y*)#} ⊆# *mset-ran B*›
proof −
have ‹{#*x*, *y*#} ⊆# *mset-set* (*dom B*)›
using *assms mset-set.remove* **by** *fastforce*
thus *?thesis*

unfolding *mset-ran-def*
using *image-mset-subseteq-mono* **by** *fastforce*
qed

lemma *not-in-dom-None*[*simp*]: ‹$x \notin dom\ B \Longrightarrow B\ x = None$›
by *blast*

lemma *in-dom-eq-None-case*[*simp*]:
‹$x \in dom\ B \Longrightarrow (case\ B\ x\ of\ None \Rightarrow a \mid Some\ y \Rightarrow b\ y) = b\ (the\ (B\ x))$›
by *auto*

definition *sucmax* :: ‹$nat \Rightarrow nat \Rightarrow nat$› **where**
‹$sucmax\ a\ b = Suc\ (max\ a\ b)$›

global-interpretation *sucmax*: *huffman-algebra sucmax*
defines *sucmax-value-bound-huffman* = *sucmax.value-bound-huffman*
and *sucmax-huffman-step-sorted-list* = *sucmax.huffman-step-sorted-list*
by (*unfold-locales*; *unfold sucmax-def*; *auto*)

lemma *nat-pred-sucmax-mono*:
‹$nat.pred\ (sucmax\ a\ b) \leq sucmax\ (nat.pred\ a)\ (nat.pred\ b)$›
by (*cases a*; *cases b*; *simp add*: *sucmax-def*)

lemma *mono-nat-pred*: ‹*mono nat.pred*›
unfolding *mono-def*
proof (*rule*; *rule*; *rule*)
fix *x y* :: *nat* **assume** ‹$x \leq y$›
thus ‹$nat.pred\ x \leq nat.pred\ y$›
by (*cases x*; *simp*; *cases y*; *simp*)
qed

lemma *sucmax-value-bound-mset-pred*:
assumes ‹$A \neq \{\#\}$›
shows ‹$nat.pred\ (sucmax.value\text{-}bound\text{-}mset\ A) \leq sucmax.value\text{-}bound\text{-}mset\ (image\text{-}mset\ nat.pred\ A)$›
by (*rule sucmax.value-bound-mono*; *simp add*: *nat-pred-sucmax-mono mono-nat-pred assms*)

lemma *mset-ran-pruned-bounds-suc-nn*:
assumes ‹*finite* (*dom B*)› ‹$fst\ c \notin dom\ B$› ‹$snd\ c \notin dom\ B$›
shows ‹*mset-ran* (*pruned-bounds-suc c B*) = *image-mset nat.pred* (*mset-ran B*)›
unfolding *pruned-bounds-suc-def*
using *assms*
by (*simp add*: *mset-ran-upd combine-bounds-def mset-ran-map-option*)

lemma *mset-ran-pruned-bounds-suc-in*:

assumes ‹*finite* (*dom* *B*)› ‹*fst* *c* ∈ *dom* *B*› ‹*snd* *c* ∉ *dom* *B* ∨ *snd* *c* = *fst* *c*›
shows ‹*mset-ran* (*pruned-bounds-suc* *c* *B*) =
add-mset (*the* (*B* (*fst* *c*))) (*image-mset* *nat.pred* (*mset-ran* *B* − {#*the* (*B* (*fst* *c*)) #}))›
proof −
have ∗: ‹*pruned-bounds-suc* *c* *B* = (*map-option* *nat.pred* ∘ *B*)(*fst* *c* ↦ *the* (*B* (*fst* *c*)))›
unfolding *pruned-bounds-suc-def* *combine-bounds-def*
using *assms* **by** *auto*
show *?thesis*
unfolding ∗
by (*subst* *mset-ran-upd*; *simp* *add*: *assms* *mset-ran-map-option* *mset-ran-Melem* *image-mset-Diff*)
qed

lemma *mset-ran-pruned-bounds-suc-ni*:
assumes ‹*finite* (*dom* *B*)› ‹*fst* *c* ∉ *dom* *B*› ‹*snd* *c* ∈ *dom* *B*›
shows ‹*mset-ran* (*pruned-bounds-suc* *c* *B*) =
add-mset (*the* (*B* (*snd* *c*))) (*image-mset* *nat.pred* (*mset-ran* *B* − {#*the* (*B* (*snd* *c*)) #}))›
proof −
have ∗:
‹*pruned-bounds-suc* *c* *B* = (*map-option* *nat.pred* ∘ *B*)(*snd* *c* := *None*)(*fst* *c* ↦ *the* (*B* (*snd* *c*)))›
unfolding *pruned-bounds-suc-def* *combine-bounds-def*
using *assms* **by** *auto*
show *?thesis*
unfolding ∗
by (*subst* *mset-ran-upd*; *simp* *add*: *assms* *mset-ran-map-option* *mset-ran-Melem* *image-mset-Diff*;
subst *mset-ran-upd*; *simp* *add*: *assms* *mset-ran-map-option*)
qed

lemma *mset-ran-pruned-bounds-suc-ii*:
assumes ‹*finite* (*dom* *B*)› ‹*fst* *c* ∈ *dom* *B*› ‹*snd* *c* ∈ *dom* *B*› ‹*fst* *c* ≠ *snd* *c*›
shows ‹*mset-ran* (*pruned-bounds-suc* *c* *B*) =
add-mset (*max* (*the* (*B* (*fst* *c*))) (*the* (*B* (*snd* *c*))))
(*image-mset* *nat.pred* (*mset-ran* *B* − {#*the* (*B* (*fst* *c*)), *the* (*B* (*snd* *c*)) #}))›
proof −
have ∗:
‹*pruned-bounds-suc* *c* *B* =
(*map-option* *nat.pred* ∘ *B*)(*snd* *c* := *None*)(*fst* *c* ↦ *max* (*the* (*B* (*fst* *c*))) (*the* (*B* (*snd* *c*))))›
unfolding *pruned-bounds-suc-def* *combine-bounds-def*
using *assms* **by** *auto*
show *?thesis*
unfolding ∗
by (*subst* *mset-ran-upd*; *simp* *add*: *assms* *mset-ran-map-option* *mset-ran-pair* *image-mset-Diff*;

subst mset-ran-upd; *simp add*: *assms mset-ran-map-option*)
qed

lemma *sucmax-value-bound-pruned-bounds-suc-nn*:
assumes ‹*finite* (*dom B*)› ‹*dom B* ≠ {}› ‹*fst c* ∉ *dom B*› ‹*snd c* ∉ *dom B*›
shows ‹*nat.pred* (*sucmax.value-bound-mset* (*mset-ran B*)) ≤
sucmax.value-bound-mset (*mset-ran* (*pruned-bounds-suc c B*))›
by (*subst mset-ran-pruned-bounds-suc-nn*; *simp add*: *assms*;
rule sucmax-value-bound-mset-pred; *insert assms mset-ran-Melem*; *fastforce*)

lemma *sucmax-value-bound-pruned-bounds-suc-in*:
assumes ‹*finite* (*dom B*)› ‹*dom B* ≠ {}› ‹*fst c* ∈ *dom B*› ‹*snd c* ∉ *dom B* ∨ *snd c* = *fst c*›
shows ‹*nat.pred* (*sucmax.value-bound-mset* (*mset-ran B*)) ≤
sucmax.value-bound-mset (*mset-ran* (*pruned-bounds-suc c B*))›
proof (*subst mset-ran-pruned-bounds-suc-in*; (*insert assms*; *blast*; *fail*)?)
have *i1*:
‹*nat.pred* (*sucmax.value-bound-mset* (*mset-ran B*))
≤ *sucmax.value-bound-mset* (*image-mset nat.pred* (*mset-ran B*))›
by (*metis assms*(*1*, *3*) *empty-iff mset-ran-Melem set-mset-empty sucmax-value-bound-mset-pred*)

have ‹*nat.pred* (*the* (*B* (*fst c*))) ∈# *image-mset nat.pred* (*mset-ran B*)›
by (*simp add*: *assms*(*1*, *3*) *mset-ran-Melem*)

moreover have ‹*nat.pred* (*the* (*B* (*fst c*))) ≤ *the* (*B* (*fst c*))›
by (*metis Suc-n-not-le-n bot.extremum mono-def mono-nat-pred nat.split-sels*(*2*) *nat-le-linear*)

ultimately have *i2*:
‹*sucmax.value-bound-mset* (*image-mset nat.pred* (*mset-ran B*))
≤ *sucmax.value-bound-mset* (
image-mset nat.pred (*mset-ran B*)
− {#*nat.pred* (*the* (*B* (*fst c*)))#}
+ {#*the* (*B* (*fst c*))#})›
using *sucmax.value-bound-increasing* **by** *blast*

have *e1*:
‹*add-mset* (*the* (*B* (*fst c*))) (*image-mset nat.pred* (*mset-ran B* − {#*the* (*B* (*fst c*))#})) =
image-mset nat.pred (*mset-ran B*)
− {#*nat.pred* (*the* (*B* (*fst c*)))#}
+ {#*the* (*B* (*fst c*))#}›
by (*metis* (*no-types*, *lifting*) *add-mset-add-single assms*(*1*, *3*) *image-mset-Diff image-mset-single*
mset-ran-Melem mset-subset-eq-single)

show
‹*nat.pred* (*sucmax.value-bound-mset* (*mset-ran B*))
≤ *sucmax.value-bound-mset* (

add-mset (*the* (*B* (*fst c*))) (*image-mset nat.pred* (*mset-ran B* − {#*the* (*B*
(*fst c*))#})))›
using *i1 i2 e1* **by** *auto*
qed

lemma *sucmax-value-bound-pruned-bounds-suc-ni*:
assumes ‹*finite* (*dom B*)› ‹*dom B* ≠ {}› ‹*fst c* ∉ *dom B*› ‹*snd c* ∈ *dom B*›
shows ‹*nat.pred* (*sucmax.value-bound-mset* (*mset-ran B*)) ≤
sucmax.value-bound-mset (*mset-ran* (*pruned-bounds-suc c B*))›
proof (*subst mset-ran-pruned-bounds-suc-ni*; (*insert assms*; *blast*; *fail*)?)
have *i1*:
‹*nat.pred* (*sucmax.value-bound-mset* (*mset-ran B*))
≤ *sucmax.value-bound-mset* (*image-mset nat.pred* (*mset-ran B*))›
by (*metis assms*(*1*, *4*) *empty-iff mset-ran-Melem set-mset-empty sucmax-value-bound-mset-pred*)

have ‹*nat.pred* (*the* (*B* (*snd c*))) ∈# *image-mset nat.pred* (*mset-ran B*)›
by (*simp add*: *assms*(*1*, *4*) *mset-ran-Melem*)

moreover have ‹*nat.pred* (*the* (*B* (*snd c*))) ≤ *the* (*B* (*snd c*))›
by (*metis Suc-n-not-le-n bot.extremum mono-def mono-nat-pred nat.split-sels*(*2*)
nat-le-linear)

ultimately have *i2*:
‹*sucmax.value-bound-mset* (*image-mset nat.pred* (*mset-ran B*))
≤ *sucmax.value-bound-mset* (
image-mset nat.pred (*mset-ran B*)
− {#*nat.pred* (*the* (*B* (*snd c*)))#}
+ {#*the* (*B* (*snd c*))#})›
using *sucmax.value-bound-increasing* **by** *blast*

have *e1*:
‹*add-mset* (*the* (*B* (*snd c*))) (*image-mset nat.pred* (*mset-ran B* − {#*the* (*B*
(*snd c*))#})) =
image-mset nat.pred (*mset-ran B*)
− {#*nat.pred* (*the* (*B* (*snd c*)))#}
+ {#*the* (*B* (*snd c*))#}›
by (*metis* (*no-types*, *lifting*) *add-mset-add-single assms*(*1*, *4*) *image-mset-Diff*
image-mset-single
mset-ran-Melem mset-subset-eq-single)

show
‹*nat.pred* (*sucmax.value-bound-mset* (*mset-ran B*))
≤ *sucmax.value-bound-mset* (
add-mset (*the* (*B* (*snd c*))) (*image-mset nat.pred* (*mset-ran B* − {#*the* (*B*
(*snd c*))#})))›
using *i1 i2 e1* **by** *auto*
qed

lemma *sucmax-value-bound-pruned-bounds-suc-ii*:

assumes ‹*finite* (*dom* *B*)› ‹*dom* *B* ≠ {}› ‹*fst* *c* ∈ *dom* *B*› ‹*snd* *c* ∈ *dom* *B*› ‹*fst* *c* ≠ *snd* *c*›
shows ‹*nat.pred* (*sucmax.value-bound-mset* (*mset-ran* *B*)) ≤
sucmax.value-bound-mset (*mset-ran* (*pruned-bounds-suc* *c* *B*))›
proof −
define *B-suc-ran* **where** ‹
B-suc-ran = *mset-ran* *B*
− {#*the* (*B* (*fst* *c*)), *the* (*B* (*snd* *c*)) #}
+ {#*sucmax* (*the* (*B* (*fst* *c*))) (*the* (*B* (*snd* *c*)))#}›

note *B-suc-ran-def*[*simp*]

have *e1*: ‹*mset-ran* (*pruned-bounds-suc* *c* *B*) = *image-mset* *nat.pred* *B-suc-ran*›
by (*simp* *add*: *assms* *mset-ran-pruned-bounds-suc-ii* *sucmax-def*)

have *i1*: ‹*sucmax.value-bound-mset* (*mset-ran* *B*) ≤ *sucmax.value-bound-mset* *B-suc-ran*›
using *sucmax.combine-step-lower-bound*
by (*simp* *add*: *assms* *mset-ran-pair*)

have *i2*:
‹*nat.pred* (*sucmax.value-bound-mset* *B-suc-ran*)
≤ *sucmax.value-bound-mset* (*image-mset* *nat.pred* *B-suc-ran*)›
by (*metis* *add-mset-add-single* *empty-not-add-mset* *sucmax-value-bound-mset-pred* *B-suc-ran-def*)

have *i3*:
‹*nat.pred* (*sucmax.value-bound-mset* (*mset-ran* *B*))
≤ *sucmax.value-bound-mset* (*image-mset* *nat.pred* *B-suc-ran*)›
by (*rule* *order-subst2*[*of*
‹*sucmax.value-bound-mset* (*mset-ran* *B*)› ‹*sucmax.value-bound-mset* *B-suc-ran*›];
insert *i1* *i2* *mono-def* *mono-nat-pred*; *blast*)

show *?thesis*
using *e1* *i3* **by** *auto*
qed

lemma *sucmax-value-bound-pruned-bounds-suc*:
assumes ‹*finite* (*dom* *B*)› ‹*dom* *B* ≠ {}›
shows ‹*nat.pred* (*sucmax.value-bound-mset* (*mset-ran* *B*)) ≤
sucmax.value-bound-mset (*mset-ran* (*pruned-bounds-suc* *c* *B*))›
using *sucmax-value-bound-pruned-bounds-suc-nn*
sucmax-value-bound-pruned-bounds-suc-in
sucmax-value-bound-pruned-bounds-suc-ni
sucmax-value-bound-pruned-bounds-suc-ii
by (*metis* *assms*)

lemma *finite-dom-pruned-bounds-suc*:
assumes ‹*finite* (*dom* *B*)› ‹*dom* *B* ≠ {}›

```
  shows ‹finite (dom (pruned-bounds-suc c B))›
  unfolding pruned-bounds-suc-def combine-bounds-def
  using assms
  by auto

lemma nonempty-dom-pruned-bounds-suc:
  assumes ‹finite (dom B)› ‹dom B ≠ {}›
  shows ‹dom (pruned-bounds-suc c B) ≠ {}›
  unfolding pruned-bounds-suc-def combine-bounds-def
  using assms
  by (cases ‹fst c ∈ dom B›; cases ‹snd c ∈ dom B›; auto)

lemma pls-bound-1-from-sucmax-value-bound-mset:
  assumes ‹pruned-bounds V B› ‹finite (dom B)› ‹dom B ≠ {}›
    ‹sucmax.value-bound-mset (mset-ran B) ≠ 0›
  shows ‹pls-bound V 1›
proof (cases ‹size (mset-ran B) = 1›)
  case True
  then obtain b where b: ‹mset-ran B = {#b#}›
    using size-1-singleton-mset by blast
  hence b-ne-0: ‹b ≠ 0›
    using assms(4) sucmax.value-bound-singleton by auto

  obtain i where i: ‹B i = Some b›
    using b assms(2, 3) mset-ran-Melem by fastforce
  hence ‹pruned-bound V i b›
    by (metis assms(1) domI option.sel pruned-bounds-def)
  then show ?thesis
    by (metis One-nat-def Suc-le-eq b-ne-0 le-0-eq neq0-conv pls-bound-def
        pls-bound-from-pruned-bound)
next
  case False
  moreover have ‹size (mset-ran B) ≠ 0›
   by (metis assms(2) assms(3) image-mset-is-empty-iff mset-ran-def mset-set-empty-iff
        size-eq-0-iff-empty)
  ultimately have ‹size (mset-ran B) ≥ 2›
    by linarith
  hence card-dom-B: ‹card (dom B) ≥ 2›
    using assms
    by (simp add: size-mset-ran)

  obtain i1 where i1: ‹i1 ∈ dom B›
    using assms(3) by blast

  have ‹dom B − {i1} ≠ {}›
    using card-dom-B
```

by (*metis One-nat-def Suc-n-not-le-n assms*(*2*) *card.remove card-empty i1*
one-add-one plus-1-eq-Suc)
then obtain *i2* **where** *i2*: ‹*i2* ∈ *dom B* − {*i1*}›
by *blast*
hence ‹((≠) *i1*) ≠ ((≠) *i2*)›
by (*metis Diff-iff singletonI*)
moreover have ‹((≠) *i1*) ∈ *V*›
using *assms*(*1*) *i1 pruned-bounds-def pruned-bound-def* **by** *blast*
moreover have ‹((≠) *i2*) ∈ *V*›
using *assms*(*1*) *i2 pruned-bounds-def pruned-bound-def* **by** *blast*
ultimately have ‹¬*inj-on weight V*›
by (*metis* (*no-types*, *lifting*) *inj-on-contraD weight-one*)
then show *?thesis*
using *unsorted-bound* **by** *blast*
qed

lemma *pls-bound-from-pruned-bounds′*:
assumes ‹*pruned-bounds V B*› ‹*finite* (*dom B*)› ‹*dom B* ≠ {}›
‹*b* ≤ *sucmax.value-bound-mset* (*mset-ran B*)›
shows ‹*pls-bound V b*›
using *assms*
proof (*induction b arbitrary*: *B V rule*: *less-induct*)
case (*less b*)
show *?case*
proof (*cases b*)
case *0*
then show *?thesis*
using *trivial-bound* **by** *simp*
next
case (*Suc b′*)

hence ‹*pls-bound V 1*›
using *pls-bound-1-from-sucmax-value-bound-mset*[*of - B*] *less.prems*
by *auto*

moreover have ‹⋀*c*. *pls-bound* (*apply-cmp c* ‘ *V*) *b′*›
proof −
fix *c*

have ∗: ‹*pruned-bounds* (*apply-cmp c* ‘ *V*) (*pruned-bounds-suc c B*)›
by (*simp add*: *apply-cmp-pruned-bounds less.prems*(*1*))

have ∗∗:
‹*nat.pred* (*sucmax.value-bound-mset* (*mset-ran B*))
≤ *sucmax.value-bound-mset* (*mset-ran* (*pruned-bounds-suc c B*))›
using *less.prems*(*2*) *less.prems*(*3*) *sucmax-value-bound-pruned-bounds-suc*
by *blast*

have ∗∗∗:

‹$b' \leq$ *nat.pred* (*sucmax.value-bound-mset* (*mset-ran B*))›
using *Suc Suc-le-D less.prems*(*4*) **by** *fastforce*

show ‹*pls-bound* (*apply-cmp c* ‘ *V*) b'›
by (*rule less.IH*[*of* - - ‹*pruned-bounds-suc c B*›];
insert * ** *** *less.prems finite-dom-pruned-bounds-suc nonempty-dom-pruned-bounds-suc dual-order.trans*; *auto simp add*: *Suc*)
qed

ultimately show *?thesis*
by (*simp add*: *Suc suc-bound bound-unsorted*)
qed
qed

lemma *pls-bound-from-pruned-bounds*:
assumes ‹*pruned-bounds V B*› ‹*finite* (*dom B*)› ‹*dom B* $\neq$ {}›
shows ‹*pls-bound V* (*sucmax.value-bound-mset* (*mset-ran B*))›
using *assms pls-bound-from-pruned-bounds'*
by *blast*

abbreviation *apply-pol* :: ‹*nat* ⇒ *bool* ⇒ *vect* ⇒ *vect*› **where**
‹*apply-pol n pol v* ≡ (*if pol then v else invert-vect n v*)›

lemma *apply-pol-invol*: ‹*apply-pol n pol* (*apply-pol n pol v*) = *v*›
by (*cases pol*; *simp add*: *invert-vect-invol pointfree-idE*)

definition *pruned-bound-pol* :: ‹*nat* ⇒ *bool* ⇒ *vect set* ⇒ *nat* ⇒ *nat* ⇒ *bool*›
where
‹*pruned-bound-pol n pol V i b* = *pruned-bound* (*apply-pol n pol* ‘ *V*) *i b*›

definition *pruned-bounds-pol* :: ‹*nat* ⇒ *bool* ⇒ *vect set* ⇒ (*nat* ⇀ *nat*) ⇒ *bool*›
where
‹*pruned-bounds-pol n pol V B* = (∀ *i* ∈ *dom B*. *pruned-bound-pol n pol V i* (*the* (*B i*)))›

lemma *pls-bound-from-pruned-bounds-pol*:
assumes ‹$\bigwedge v.\ v \in V \Longrightarrow$ *fixed-width-vect n v*›
‹*pruned-bounds-pol n pol V B*› ‹*finite* (*dom B*)› ‹*dom B* $\neq$ {}›
shows ‹*pls-bound V* (*sucmax.value-bound-mset* (*mset-ran B*))›
proof (*cases pol*)
case *True*
hence ‹*pruned-bounds V B*›
using *assms* **unfolding** *pruned-bounds-def pruned-bounds-pol-def pruned-bound-pol-def*
by *simp*
then show *?thesis*
using *assms pls-bound-from-pruned-bounds* **by** *simp*
next

case *False*
hence ‹*pruned-bounds* (*invert-vect n* ‘ *V*) *B*›
using *assms* **unfolding** *pruned-bounds-def pruned-bounds-pol-def pruned-bound-pol-def*
by *simp*
hence ‹*pls-bound* (*invert-vect n* ‘ *V*) (*sucmax.value-bound-mset* (*mset-ran B*))›
using *assms pls-bound-from-pruned-bounds* **by** *blast*
hence ‹*pls-bound* (*invert-vect n* ‘ *invert-vect n* ‘ *V*) (*sucmax.value-bound-mset* (*mset-ran B*))›
using *assms*(*1*) *invert-vect-fixed-width*
by (*auto intro*: *inverted-bound*)
then show *?thesis*
unfolding *image-comp invert-vect-invol*
by *simp*
qed

lemma *apply-pol-bound*:
assumes ‹$\bigwedge$*v*. *v* ∈ *V* ⟹ *fixed-width-vect n v*› ‹*pls-bound V b*›
shows ‹*pls-bound* (*apply-pol n pol* ‘ *V*) *b*›
using *assms*
by (*cases pol*; *simp add*: *inverted-bound*)

lemma *apply-pol-bound-iff*:
assumes ‹$\bigwedge$*v*. *v* ∈ *V* ⟹ *fixed-width-vect n v*›
shows ‹*pls-bound* (*apply-pol n pol* ‘ *V*) *b* = *pls-bound V b*›
proof
assume ‹*pls-bound* (*apply-pol n pol* ‘ *V*) *b*›
hence ‹*pls-bound* (*apply-pol n pol* ‘ (*apply-pol n pol* ‘ *V*)) *b*›
by (*intro apply-pol-bound*[**where** *V*=‹*apply-pol n pol* ‘ *V*›];
insert assms invert-vect-fixed-width; *auto*)
thus ‹*pls-bound V b*›
using *apply-pol-invol*[**where** *n*=*n* **and** *pol*=*pol*]
unfolding *image-image* **by** *simp*
next
assume ‹*pls-bound V b*›
thus ‹*pls-bound* (*apply-pol n pol* ‘ *V*) *b*›
using *assms apply-pol-bound* **by** *simp*
qed

end
theory *Checker*
imports *Main Sorting-Network*
begin

datatype *proof-witness* =
ProofWitness (*witness-step-id*: *int*) (*witness-invert*: *bool*) (*witness-perm*: ‹*int list*›)

datatype *proof-step-witnesses* =
HuffmanWitnesses (*huffman-polarity*: *bool*) (*huffman-witnesses*: ‹*proof-witness option list*›) |

SuccessorWitnesses (*successor-witnesses*: ‹*proof-witness option list*›)

datatype *proof-step* =
ProofStep
(*step-width*: *int*)
(*step-vect-list*: ‹*bool list list*›)
(*step-bound*: *int*)
(*step-witnesses*: *proof-step-witnesses*)

datatype *proof-cert* =
ProofCert (*cert-length*: *int*) (*cert-step*: ‹*int* ⇒ *proof-step*›)

datatype *vect-trie* =
VtEmpty | *VtNode bool vect-trie vect-trie*

fun *vt-singleton* :: ‹*bool list* ⇒ *vect-trie*› **where**
‹*vt-singleton* [] = *VtNode True VtEmpty VtEmpty*› |
‹*vt-singleton* (*False* # *xs*) = *VtNode False* (*vt-singleton xs*) *VtEmpty*› |
‹*vt-singleton* (*True* # *xs*) = *VtNode False VtEmpty* (*vt-singleton xs*)›

fun *vt-union* :: ‹*vect-trie* ⇒ *vect-trie* ⇒ *vect-trie*› **where**
‹*vt-union VtEmpty VtEmpty* = *VtEmpty*› |
‹*vt-union a VtEmpty* = *a*› |
‹*vt-union VtEmpty b* = *b*› |
‹*vt-union* (*VtNode a a-lo a-hi*) (*VtNode b b-lo b-hi*) =
VtNode (*a* ∨ *b*) (*vt-union a-lo b-lo*) (*vt-union a-hi b-hi*)›

lemma *vt-union-commutative*:
‹*vt-union A B* = *vt-union B A*›
by (*induction A B rule*: *vt-union.induct*; *auto*)

definition *vt-list* :: ‹*bool list list* ⇒ *vect-trie*› **where**
‹*vt-list ls* = *fold* (*vt-union* ∘ *vt-singleton*) *ls VtEmpty*›

fun *set-vt* :: ‹*vect-trie* ⇒ *bool list set*› **where**
‹*set-vt VtEmpty* = {}› |
‹*set-vt* (*VtNode True lo hi*) = {[]} ∪ (((#) *False*) ‘ *set-vt lo*) ∪ (((#) *True*) ‘ *set-vt hi*)› |
‹*set-vt* (*VtNode False lo hi*) = (((#) *False*) ‘ *set-vt lo*) ∪ (((#) *True*) ‘ *set-vt hi*)›

lemma *set-vt-union*:
‹*set-vt* (*vt-union A B*) = *set-vt A* ∪ *set-vt B*›
proof (*induction A B rule*: *vt-union.induct*; (*simp*; *fail*)?)
case (*4 a a-lo a-hi b b-lo b-hi*)
thus *?case*
by (*subst vt-union.simps*; *cases a*; *cases b*; *auto*)

```
qed

lemma set-vt-singleton:
  ‹set-vt (vt-singleton xs) = {xs}›
proof (induction xs)
  case Nil
  thus ?case
    by auto
next
  case (Cons x xs)
  thus ?case
    by (cases x; simp)
qed

lemma homo-fold:
  assumes ‹⋀A B. f (g A B) = h (f A) (f B)›
  shows ‹f (fold g xs x) = fold h (map f xs) (f x)›
  by (induction xs arbitrary: x; simp add: assms)

lemma set-vt-list:
  ‹set-vt (vt-list ls) = set ls›
proof -
  have ‹set-vt (vt-list ls) = set-vt (fold (vt-union) (map vt-singleton ls) VtEmpty)›
    by (simp add: vt-list-def fold-map)
  also have ‹... = fold (∪) (map set-vt (map vt-singleton ls)) (set-vt VtEmpty)›
    by (simp add: homo-fold set-vt-union)
  also have ‹... = fold (∪) (map (λl. {l}) ls) {}›
    by (simp add: comp-def set-vt-singleton)
  also have ‹... = set ls›
    by (metis Sup-set-fold UN-singleton image-set)
  finally show ?thesis.
qed

fun list-vt :: ‹vect-trie ⇒ bool list list› where
  ‹list-vt VtEmpty = []› |
  ‹list-vt (VtNode True lo hi) =
    [] # map ((#) False) (list-vt lo) @ map ((#) True) (list-vt hi)› |
  ‹list-vt (VtNode False lo hi) =
    map ((#) False) (list-vt lo) @ map ((#) True) (list-vt hi)›

lemma set-list-vt:
  ‹set (list-vt A) = set-vt A›
  by (induction A rule: list-vt.induct; auto)

fun list-vt-extend ::
  ‹vect-trie ⇒ (bool list ⇒ bool list) ⇒ bool list list ⇒ bool list list› where
  ‹list-vt-extend VtEmpty el-prefix suffix = suffix› |
  ‹list-vt-extend (VtNode True lo hi) el-prefix suffix =
    el-prefix [] # list-vt-extend lo (el-prefix ∘ ((#) False)) (
```

list-vt-extend hi (*el-prefix* ∘ ((#) *True*)) *suffix*)› |
‹*list-vt-extend* (*VtNode False lo hi*) *el-prefix suffix* =
list-vt-extend lo (*el-prefix* ∘ ((#) *False*)) (
list-vt-extend hi (*el-prefix* ∘ ((#) *True*)) *suffix*)›

lemma *dlist-as-bs*[*simp*]: ‹((@) *as* ∘ (@) *bs*) = ((@) (*as* @ *bs*))›
by (*rule ext*; *simp*)

lemma *dlist-as-b*[*simp*]: ‹((@) *as* ∘ (#) *b*) = ((@) (*as* @ [*b*]))›
by (*rule ext*; *simp*)

lemma *dlist-a-bs*[*simp*]: ‹((#) *a* ∘ (@) *bs*) = ((@) (*a* # *bs*))›
by (*rule ext*; *simp*)

lemma *list-vt-extend-as-list-vt*:
‹*list-vt-extend A* ((@) *el-prefix*) *suffix* = *map* ((@) *el-prefix*) (*list-vt A*) @ *suffix*›
by (*induction A* ‹((@) *el-prefix*)› *suffix arbitrary*: *el-prefix rule*: *list-vt-extend.induct*; *simp*)

lemma *list-vt-as-list-vt-extend*[*code*]:
‹*list-vt A* = *list-vt-extend A id* []›
proof −
have ‹*list-vt A* = *list-vt-extend A* ((@) []) []›
using *list-vt-extend-as-list-vt*
by (*simp add*: *map-idI*)
also have ‹. . . = *list-vt-extend A id* []›
by (*metis append-Nil id-apply*)
finally show *?thesis*.
qed

lemma *empty-list-nmember-cons-image*:
‹[] ∉ ((#) *x*) ‘ *A*›
by *blast*

lemma *cons-image-disjoint*:
‹*x* ≠ *y* ⟹ (((#) *x*) ‘ *A*) ∩ (((#) *y*) ‘ *B*) = {}›
by *blast*

lemma *distinct-map′*:
‹⟦*distinct xs*; *inj f*⟧ ⟹ *distinct* (*map f xs*)›
using *distinct-map inj-on-subset* **by** *blast*

lemma *distinct-list-vt*:
‹*distinct* (*list-vt A*)›
by (*induction A rule*: *list-vt.induct*;
simp add: *empty-list-nmember-cons-image cons-image-disjoint distinct-map′*)

fun *is-subset-vt* :: ‹*vect-trie* ⇒ *vect-trie* ⇒ *bool*› **where**
‹*is-subset-vt VtEmpty a* = *True*› |

⟨*is-subset-vt* (*VtNode True a-lo a-hi*) *VtEmpty* = *False*⟩ |
⟨*is-subset-vt* (*VtNode False a-lo a-hi*) *VtEmpty* =
  (*is-subset-vt a-lo VtEmpty* ∧ *is-subset-vt a-hi VtEmpty*)⟩ |
⟨*is-subset-vt* (*VtNode True a-lo a-hi*) (*VtNode False b-lo b-hi*) = *False*⟩ |
⟨*is-subset-vt* (*VtNode False a-lo a-hi*) (*VtNode b b-lo b-hi*) =
  (*is-subset-vt a-lo b-lo* ∧ *is-subset-vt a-hi b-hi*)⟩ |
⟨*is-subset-vt* (*VtNode a a-lo a-hi*) (*VtNode True b-lo b-hi*) =
  (*is-subset-vt a-lo b-lo* ∧ *is-subset-vt a-hi b-hi*)⟩

lemma *set-vt-subset-cons-False*:
  assumes ⟨*set-vt A* ⊆ *set-vt B*⟩
  shows ⟨(#) *False* ‘ *set-vt A* ⊆ *set-vt* (*VtNode b B B′*)⟩
proof −
  have ⟨(#) *False* ‘ *set-vt A* ⊆ (#) *False* ‘ *set-vt B*⟩
    by (*meson assms image-mono*)
  thus *?thesis*
    by (*metis* (*full-types*) *Un-commute le-supI1 set-vt.simps*(*2*, *3*))
qed

lemma *set-vt-subset-cons-True*:
  assumes ⟨*set-vt A* ⊆ *set-vt B*⟩
  shows ⟨(#) *True* ‘ *set-vt A* ⊆ *set-vt* (*VtNode b B′ B*)⟩
proof −
  have ⟨(#) *True* ‘ *set-vt A* ⊆ (#) *True* ‘ *set-vt B*⟩
    by (*meson assms image-mono*)
  thus *?thesis*
    by (*metis* (*full-types*) *Un-commute le-supI1 set-vt.simps*(*2*, *3*))
qed

lemma *set-vt-subset-is-subset-vt*:
  assumes ⟨*is-subset-vt A B*⟩
  shows ⟨*set-vt A* ⊆ *set-vt B*⟩
  using *assms*
  by (*induction A B rule*: *is-subset-vt.induct*;
      insert set-vt-subset-cons-False set-vt-subset-cons-True; *auto*)

fun *is-member-vt* :: ⟨*bool list* ⇒ *vect-trie* ⇒ *bool*⟩ **where**
  ⟨*is-member-vt - VtEmpty* = *False*⟩ |
  ⟨*is-member-vt* [] (*VtNode a - -*) = *a*⟩ |
  ⟨*is-member-vt* (*False* # *xs*) (*VtNode - a-lo -*) = *is-member-vt xs a-lo*⟩ |
  ⟨*is-member-vt* (*True* # *xs*) (*VtNode - - a-hi*) = *is-member-vt xs a-hi*⟩

lemma *is-member-vt-as-member-set-vt*:
  ⟨*is-member-vt xs A* = (*xs* ∈ *set-vt A*)⟩
proof (*induction xs A rule*: *is-member-vt.induct*; (*simp*; *fail*)*?*)
  case (*2 a uv uw*)
  then show *?case*
    by (*cases a*; *simp add*: *empty-list-nmember-cons-image*)
next

  case (*3 xs ux a-lo uy*)
  then show *?case*
    by (*cases ux*; *auto*)
next
  case (*4 xs uz va a-hi*)
  then show *?case*
    by (*cases uz*; *auto*)
qed

fun *list-to-vect* :: ‹*bool list* ⇒ *vect*› **where**
  ‹*list-to-vect* [] *i* = *True*› |
  ‹*list-to-vect* (*x* # *xs*) *0* = *x*› |
  ‹*list-to-vect* (*x* # *xs*) (*Suc i*) = *list-to-vect xs i*›

lemma *list-to-vect-as-nth*:
  ‹*list-to-vect xs* = (λ*i*. *if i* < *length xs then xs*!*i else True*)›
proof
  fix *i* **show** ‹*list-to-vect xs i* = (*if i* < *length xs then xs* ! *i else True*)›
  proof (*induction xs arbitrary*: *i*)
    case *Nil*
    thus *?case*
      by *simp*
  next
    case (*Cons a xs*)
    thus *?case*
      by (*cases i*; *simp*)
  qed
qed

lemma *list-to-vect-inj-on-same-length*:
  ‹*inj-on list-to-vect* {*xs*. *length xs* = *n*}›
  by (*rule*; *simp add*: *list-eq-iff-nth-eq list-to-vect-as-nth*; *metis*)

fun *apply-cmp-list* :: ‹*cmp* ⇒ *bool list* ⇒ *bool list*› **where**
  ‹*apply-cmp-list* (*a*, *b*) *xs* = (*let xa* = *xs*!*a*; *xb* = *xs*!*b in xs*[*a* := *xa* ∧ *xb*, *b* := *xa* ∨ *xb*])›

lemma *length-apply-cmp-list*:
  ‹*length* (*apply-cmp-list c xs*) = *length xs*›
  by (*metis apply-cmp-list.elims length-list-update*)

lemma *apply-cmp-list-to-vect*:
  assumes ‹*a* < *length xs*› ‹*b* < *length xs*›
  shows ‹*apply-cmp* (*a*, *b*) (*list-to-vect xs*) = *list-to-vect* (*apply-cmp-list* (*a*, *b*) *xs*)›
  by (*rule*; *simp add*: *assms list-to-vect-as-nth Let-def apply-cmp-def*)

fun *length-vt* :: ‹*vect-trie* ⇒ *nat*› **where**

⟨*length-vt VtEmpty* = *0*⟩ |
⟨*length-vt* (*VtNode True lo hi*) = *Suc* (*length-vt lo* + *length-vt hi*)⟩ |
⟨*length-vt* (*VtNode False lo hi*) = *length-vt lo* + *length-vt hi*⟩

lemma *length-vt-as-length-list-vt*:
⟨*length-vt A* = *length* (*list-vt A*)⟩
by (*induction A rule*: *list-vt.induct*; *simp*)

lemma *length-vt-as-card-set-vt*:
⟨*length-vt A* = *card* (*set-vt A*)⟩
proof −
have ⟨*length-vt A* = *card* (*set* (*list-vt A*))⟩
by (*simp add*: *length-vt-as-length-list-vt distinct-card distinct-list-vt*)
also have ⟨. . . = *card* (*set-vt A*)⟩
by (*simp add*: *set-list-vt*)
finally show *?thesis*.
qed

definition *is-unsorted-vt* :: ⟨*nat* ⇒ *vect-trie* ⇒ *bool*⟩ **where**
⟨*is-unsorted-vt n A* = (*length-vt A* > *Suc n*)⟩

lemma *fixed-width-vect-list-to-vect*:
⟨*length xs* = *n* ⟹ *fixed-width-vect n* (*list-to-vect xs*)⟩
unfolding *fixed-width-vect-def list-to-vect-as-nth*
by *simp*

lemma *is-unsorted-vt-bound*:
assumes ⟨*set-vt A* ⊆ {*xs*. *length xs* = *n*}⟩ ⟨*is-unsorted-vt n A*⟩
shows ⟨*pls-bound* (*list-to-vect* ‘ *set-vt A*) *1*⟩
proof (*rule unsorted-by-card-bound*)
fix *v* **assume** ⟨*v* ∈ *list-to-vect* ‘ *set-vt A*⟩
thus ⟨*fixed-width-vect n v*⟩
using *assms fixed-width-vect-list-to-vect*
by *blast*
next
show ⟨*n* + *1* < *card* (*list-to-vect* ‘ *set-vt A*)⟩
using *assms*
unfolding *is-unsorted-vt-def length-vt-as-card-set-vt*
by (*subst card-image*; *linarith?*; *insert inj-on-subset list-to-vect-inj-on-same-length*; *blast*)
qed

abbreviation *list-to-perm* :: ⟨*nat list* ⇒ (*nat* ⇒ *nat*)⟩ **where**
⟨*list-to-perm xs i* ≡ *if i* < *length xs then xs*!*i else i*⟩

lemma *bij-list-to-perm*:
assumes ⟨*distinct xs*⟩ ⟨*list-all* (λ*i*. *i* < *length xs*) *xs*⟩
shows ⟨*bij* (*list-to-perm xs*)⟩
proof −

```
  have *: ‹card (set xs) = length xs›
    by (simp add: assms(1) distinct-card)
  have ‹set xs ⊆ {..<length xs}›
    using assms distinct-Ex1 list-all-length by fastforce
  hence ‹set xs = {..<length xs}›
    using *
    by (simp add: card-subset-eq)
  hence ‹bij-betw (λi. xs!i) {..<length xs} {..<length xs}›
    by (simp add: assms(1) bij-betw-nth)
  hence ‹bij-betw (list-to-perm xs) {..<length xs} {..<length xs}›
    using bij-betw-cong[of - ‹λi. xs!i› ‹list-to-perm xs›]
    by (meson lessThan-iff)

  moreover have
    ‹bij-betw (list-to-perm xs) (−{..<length xs}) (−{..<length xs})›
    using bij-betw-cong[of ‹−{..<length xs}› ‹id› ‹list-to-perm xs›]
    by auto

  ultimately show ‹bij (list-to-perm xs)›
    using bij-betw-combine[of
        ‹list-to-perm xs›
        ‹{..<length xs}› ‹{..<length xs}›
        ‹−{..<length xs}› ‹−{..<length xs}›]
    by (metis Compl-partition inf-compl-bot)
qed

definition permute-list-vect :: ‹nat list ⇒ bool list ⇒ bool list› where
  ‹permute-list-vect ps xs = map (λi. xs!i) ps›

lemma list-to-vect-permute-list-vect-as-apply-perm:
  assumes ‹list-all (λx. x < length ps) ps› ‹length ps = length xs›
 shows ‹list-to-vect (permute-list-vect ps xs) = apply-perm (list-to-perm ps) (list-to-vect
xs)›
proof
  fix i
  show ‹list-to-vect (permute-list-vect ps xs) i =
    apply-perm (list-to-perm ps) (list-to-vect xs) i›
    unfolding permute-list-vect-def apply-perm-def list-to-vect-as-nth
    using assms
    by (simp add: list-all-length)
qed

lemma perm-list-to-vect-set:
  assumes ‹list-all (λx. x < length ps) ps› ‹length ps = n› ‹set-vt A ⊆ {xs. length
xs = n}›
  shows ‹list-to-vect ‘ set-vt (vt-list (map (permute-list-vect ps) (list-vt A))) =
    apply-perm (list-to-perm ps) ‘ list-to-vect ‘ set-vt A›
proof −
  have
```

‹*set-vt* (*vt-list* (*map* (*permute-list-vect ps*) (*list-vt A*)))
  = *set* (*map* (*permute-list-vect ps*) (*list-vt A*))›
  using *set-vt-list* **by** *blast*
also have ‹... = *permute-list-vect ps* ‘ *set-vt A*›
  by (*simp add*: *set-list-vt*)
finally have
  ‹*list-to-vect* ‘ *set-vt* (*vt-list* (*map* (*permute-list-vect ps*) (*list-vt A*))) =
    list-to-vect ‘ *permute-list-vect ps* ‘ *set-vt A*›
  by *simp*
also have ‹... = (λ*xs*. *list-to-vect* (*permute-list-vect ps xs*)) ‘ *set-vt A*›
  by *blast*
also have ‹... = (λ*xs*. *apply-perm* (*list-to-perm ps*) (*list-to-vect xs*)) ‘ *set-vt A*›
  by (*rule image-cong*; *simp?*; *subst list-to-vect-permute-list-vect-as-apply-perm*;
    (*simp add*: *assms*(*1*))?; *insert assms*; *auto*)
finally show *?thesis*
  by (*simp add*: *image-image*)
qed

lemma *list-to-vect-map-Not-as-invert-vect*:
  ‹*list-to-vect* (*map* (*Not*) *xs*) = *invert-vect* (*length xs*) (*list-to-vect xs*)›
  unfolding *list-to-vect-as-nth invert-vect-def*
  by *simp*

definition *permute-vt* :: ‹*nat list* ⇒ *vect-trie* ⇒ *vect-trie*› **where**
  ‹*permute-vt ps A* = *vt-list* (*map* (*permute-list-vect ps*) (*list-vt A*))›

lemma *permute-vt-bound*:
  assumes ‹*length ps* = *n*› ‹*distinct ps*› ‹*list-all* (λ*i*. *i* < *length ps*) *ps*›
    ‹*set-vt A* ⊆ {*xs*. *length xs* = *n*}›
    ‹*pls-bound* (*list-to-vect* ‘ *set-vt A*) *b*›
  shows ‹*pls-bound* (*list-to-vect* ‘ *set-vt* (*permute-vt ps A*)) *b*›
  using *perm-list-to-vect-set assms bij-list-to-perm permute-vt-def permuted-bound*
by *auto*

lemma *invert-list-to-vect-set*:
  assumes ‹*set-vt A* ⊆ {*xs*. *length xs* = *n*}›
  shows ‹*list-to-vect* ‘ *set-vt* (*vt-list* (*map* (*map Not*) (*list-vt A*))) =
    invert-vect n ‘ *list-to-vect* ‘ *set-vt A*›
proof –
  have
    ‹*set-vt* (*vt-list* (*map* (*map Not*) (*list-vt A*)))
      = *set* (*map* (*map Not*) (*list-vt A*))›
    using *set-vt-list* **by** *blast*
  also have ‹... = *map Not* ‘ *set-vt A*›
    by (*simp add*: *set-list-vt*)
  finally have
    ‹*list-to-vect* ‘ *set-vt* (*vt-list* (*map* (*map Not*) (*list-vt A*))) =
      list-to-vect ‘ *map Not* ‘ *set-vt A*›
    by *simp*

also have ‹... = (*λxs. list-to-vect* (*map Not xs*)) ‘ *set-vt A*›
by *blast*
also have ‹... = (*λxs. invert-vect n* (*list-to-vect xs*)) ‘ *set-vt A*›
by (*rule image-cong*; *simp?*; *subst list-to-vect-map-Not-as-invert-vect*; *insert assms*; *auto*)
finally show *?thesis*
by (*simp add*: *image-image*)
qed

definition *invert-vt* :: ‹*bool* ⇒ *vect-trie* ⇒ *vect-trie*› **where**
‹*invert-vt z A* = (*if z then vt-list* (*map* (*map Not*) (*list-vt A*)) *else A*)›

lemma *list-to-vect-set-vt-fixed-width*:
assumes ‹*set-vt A* ⊆ {*xs. length xs* = *n*}›
shows ‹*list-to-vect* ‘ *set-vt A* ⊆ {*v. fixed-width-vect n v*}›
by (*metis* (*mono-tags*, *lifting*) *Ball-Collect assms fixed-width-vect-list-to-vect image-subsetI mem-Collect-eq*)

lemma *invert-vt-bound*:
assumes ‹*set-vt A* ⊆ {*xs. length xs* = *n*}›
‹*pls-bound* (*list-to-vect* ‘ *set-vt A*) *b*›
shows ‹*pls-bound* (*list-to-vect* ‘ *set-vt* (*invert-vt z A*)) *b*›
by (*cases z*; *unfold invert-vt-def*; *simp add*: *assms*;
subst invert-list-to-vect-set[**where** *n*=*n*]; *simp add*: *assms*;
metis (*mono-tags*) *Ball-Collect list-to-vect-set-vt-fixed-width assms*(*1*, *2*) *inverted-bound*)

lemma *invert-vt-fixed-width*:
assumes ‹*set-vt A* ⊆ {*xs. length xs* = *n*}›
shows ‹*set-vt* (*invert-vt z A*) ⊆ {*xs. length xs* = *n*}›
using *assms*
by (*cases z*; *unfold invert-vt-def*; *auto simp add*: *set-vt-list set-list-vt*)

fun *get-bound*
:: ‹(*int* ⇒ *proof-step*) ⇒ *int* ⇒ *proof-witness option* ⇒ *nat* ⇒ *vect-trie* ⇒ *nat option*› **where**
‹*get-bound proof-steps step-limit None width A* = *Some* (*if is-unsorted-vt width A then 1 else 0*)› |
‹*get-bound proof-steps step-limit* (*Some witness*) *width A* = (
let
witness-id = *witness-step-id witness*;
perm = *map nat* (*witness-perm witness*);
step = *proof-steps witness-id*;
B-list = *step-vect-list step*;
B = *vt-list B-list*;
B′ = *permute-vt perm* (*invert-vt* (*witness-invert witness*) *B*)
in
if ¬(*0* ≤ *witness-id* ∧ *witness-id* < *step-limit*)
∨ ¬*list-all* (*λi. 0* ≤ *i* ∧ *i* < *width*) *perm*

∨ *length perm* ≠ *width*
∨ ¬*distinct perm*
∨ ¬*list-all* (*λxs. length xs* = *width*) *B-list*
∨ ¬*is-subset-vt B′ A*
then None
else Some (*nat* (*step-bound step*))
)›

definition *step-checked* :: ‹*proof-step* ⇒ *bool*› **where**
‹*step-checked step* = (
list-all (*λxs. length xs* = *nat* (*step-width step*)) (*step-vect-list step*) ∧
pls-bound (*list-to-vect* ‘ *set-vt* (*vt-list* (*step-vect-list step*))) (*nat* (*step-bound step*)))›

lemma *get-bound-bound*:
assumes ‹⋀*step*. *0* ≤ *step* ∧ *step* < *step-limit* ⟹ *step-checked* (*proof-steps step*)›
‹*set-vt A* ⊆ {*xs. length xs* = *n*}›
‹*get-bound proof-steps step-limit witness n A* = *Some b*›
shows ‹*pls-bound* (*list-to-vect* ‘ *set-vt A*) *b*›
proof (*cases witness*)
case *None*
thus *?thesis*
using *assms*(*2*, *3*) *is-unsorted-vt-bound trivial-bound* **by** *auto*
next
case (*Some some-witness*)

define *witness-id* **where** ‹*witness-id* = *witness-step-id some-witness*›
define *perm* **where** ‹*perm* = *map nat* (*witness-perm some-witness*)›
define *B* **where** ‹*B* = *vt-list* (*step-vect-list* (*proof-steps witness-id*))›
define *B′* **where** ‹*B′* = *permute-vt perm* (*invert-vt* (*witness-invert some-witness*) *B*)›

note *witness-id-def*[*simp*] *perm-def*[*simp*] *B-def*[*simp*] *B′-def*[*simp*]

have *c1*: ‹*0* ≤ *witness-id* ∧ *witness-id* < *step-limit*›
by (*rule ccontr*; *insert assms Some*; *simp*)

have *c2*: ‹*list-all* (*λi. 0* ≤ *i* ∧ *i* < *n*) *perm* ∧ *length perm* = *n* ∧ *distinct perm*›
by (*rule ccontr*; *insert assms Some*; *auto*)

have *c3*: ‹*is-subset-vt B′ A*›
by (*rule ccontr*; *insert assms Some*; *auto*)

have *c4*: ‹*list-all* (*λxs. length xs* = *n*) (*step-vect-list* (*proof-steps witness-id*))›
by (*rule ccontr*; *insert assms Some*; *auto*)

have *b*: ‹*nat* (*step-bound* (*proof-steps witness-id*)) = *b*›
using *c1 c2 c3 c4 assms Some*

by *simp*

have *B*: ‹*pls-bound* (*list-to-vect* ‘ *set-vt B*) *b*›
unfolding *B-def*
using *assms*(*1*) *b c1 step-checked-def*
by *blast*

have *c4'*: ‹*set* (*step-vect-list* (*proof-steps witness-id*)) ⊆ {*xs*. *length xs* = *n*}›
using *c4*
unfolding *Ball-Collect*[*symmetric*] *list.pred-set*
by *blast*

have *B-length*: ‹*set-vt B* ⊆ {*xs*. *length xs* = *n*}›
using *c4'*
by (*simp add*: *set-vt-list*)

have *invert-B*: ‹*pls-bound* (*list-to-vect* ‘ *set-vt* (*invert-vt* (*witness-invert some-witness*) *B*)) *b*›
by (*rule invert-vt-bound*; *insert B B-length*; *auto*)

have *B'*: ‹*pls-bound* (*list-to-vect* ‘ *set-vt B'*) *b*›
unfolding *B'-def*
by (*rule permute-vt-bound*; *insert c2 B-length invert-vt-fixed-width invert-B*; *simp*)

thus *?thesis*
by (*meson bound-mono-subset c3 image-mono set-vt-subset-is-subset-vt*)
qed

abbreviation *list-any* :: ‹('*a* ⇒ *bool*) ⇒ '*a list* ⇒ *bool*› **where**
‹*list-any p xs* ≡ ¬(*list-all* (λ*x*. ¬*p x*) *xs*)›

fun *is-redundant-cmp-vt* :: ‹*cmp* ⇒ *vect-trie* ⇒ *bool*› **where**
‹*is-redundant-cmp-vt* (*a*, *b*) *A* = (
let vs = *list-vt A in* ¬(*list-any* (λ*xs*. *xs*!*a* ∧ ¬*xs*!*b*) *vs* ∧ *list-any* (λ*xs*. ¬*xs*!*a* ∧ *xs*!*b*) *vs*))›

lemma *redundant-cmp-from-is-redundant-cmp-vt*:
assumes ‹*a* < *n*› ‹*b* < *n*›
‹*set-vt A* ⊆ {*xs*. *length xs* = *n*}›
shows ‹*is-redundant-cmp-vt* (*a*, *b*) *A* = *redundant-cmp* (*a*, *b*) (*list-to-vect* ‘ *set-vt A*)›
using *assms*
by (*simp add*: *redundant-cmp-def list-all-iff*; *insert list-to-vect-as-nth set-list-vt*; *auto*)

lemma *redundant-cmp-from-is-redundant-cmp-vt'*:
assumes ‹*fst c* < *n*› ‹*snd c* < *n*›

‹*set-vt A* ⊆ {*xs. length xs* = *n*}›
shows ‹*is-redundant-cmp-vt c A* = *redundant-cmp c* (*list-to-vect* ‘ *set-vt A*)›
using *redundant-cmp-from-is-redundant-cmp-vt*
by (*metis assms prod.collapse*)

definition *ocmp-list* :: ‹*nat* ⇒ *cmp list*› **where**
‹*ocmp-list n* = *concat* (*map* (λ*i. map* (λ*j.* (*j*, *i*)) [*0*..<*i*]) [*0*..<*n*])›

lemma *set-ocmp-list*:
‹*set* (*ocmp-list n*) = {*c. fst c* < *snd c* ∧ *snd c* < *n*}›
proof (*rule set-eqI*)
fix *c*
obtain *a b* :: *nat* **where** *ab*: ‹*c* = (*a*, *b*)›
by *fastforce*

have ‹*c* ∈ {*c. fst c* < *snd c* ∧ *snd c* < *n*} = (*a* < *b* ∧ *b* < *n*)›
using *ab* **by** *simp*
also have ‹. . . = (*c* ∈ *set* (*ocmp-list n*))›
unfolding *ab ocmp-list-def set-concat set-map image-image atLeast-upt*[*symmetric*]
by *blast*
finally show ‹*c* ∈ *set* (*ocmp-list n*) = (*c* ∈ {*c. fst c* < *snd c* ∧ *snd c* < *n*})›
..
qed

definition *check-successors* :: ‹(*int* ⇒ *proof-step*) ⇒ *int* ⇒ *proof-step* ⇒ *bool*›
where
‹*check-successors proof-steps step-limit step* = (*case step-witnesses step of*
HuffmanWitnesses - - ⇒ *False* |
SuccessorWitnesses witnesses ⇒
let
width = (*nat* (*step-width step*));
bound = (*nat* (*step-bound step*));
ocmps = *ocmp-list width*;
A-list = *step-vect-list step*;
A = *vt-list A-list*;
nrcmps = *filter* (λ*c.* ¬*is-redundant-cmp-vt c A*) *ocmps*;
Bs = *map* (λ*c. vt-list* (*map* (*apply-cmp-list c*) *A-list*)) *nrcmps*
in
bound ≠ *0* ∧
is-unsorted-vt width A ∧
length nrcmps = *length witnesses* ∧
list-all (λ*xs. length xs* = *width*) *A-list* ∧
list-all2 (λ*B w.*
case get-bound proof-steps step-limit w width B of
None ⇒ *False* | *Some b* ⇒ *Suc b* ≥ *bound*
) *Bs witnesses*
)›

lemma *list-all2-witness*:

assumes ‹*list-all2 P xs ys*›
shows ‹$\bigwedge x$. *x* ∈ *set xs* ⟹ ∃ *y*. *y* ∈ *set ys* ∧ *P x y*›
proof −
fix *x* **assume** ‹*x* ∈ *set xs*›
then obtain *i* **where** ‹*i* < *length xs*› **and** *x*: ‹*x* = *xs*!*i*›
by (*metis in-set-conv-nth*)
hence ‹*ys*!*i* ∈ *set ys* ∧ *P x* (*ys*!*i*)›
unfolding *x*
using *assms list-all2-conv-all-nth* **by** *auto*
thus ‹∃ *y*. *y* ∈ *set ys* ∧ *P x y*›
by *auto*
qed

lemma *apply-cmp-as-apply-cmp-list*:
assumes ‹*set-vt A* ⊆ {*xs*. *length xs* = *n*}› ‹*a* < *n*› ‹*b* < *n*›
shows ‹*apply-cmp* (*a*, *b*) ‘ *list-to-vect* ‘ *set-vt A* =
list-to-vect ‘ *set-vt* (*vt-list* (*map* (*apply-cmp-list* (*a*, *b*)) (*list-vt A*)))›
unfolding *set-vt-list set-map image-image set-list-vt*
by (*rule image-cong*; *insert assms apply-cmp-list-to-vect*; *blast*)

lemma *apply-cmp-as-apply-cmp-list′*:
assumes ‹*set-vt A* ⊆ {*xs*. *length xs* = *n*}› ‹*fst c* < *n*› ‹*snd c* < *n*›
shows ‹*apply-cmp c* ‘ *list-to-vect* ‘ *set-vt A* =
list-to-vect ‘ *set-vt* (*vt-list* (*map* (*apply-cmp-list c*) (*list-vt A*)))›
using *apply-cmp-as-apply-cmp-list*
by (*metis assms prod.collapse*)

lemma *check-successors-step-checked*:
assumes ‹*check-successors proof-steps step-limit step*›
‹$\bigwedge$*step*. *0* ≤ *step* ∧ *step* < *step-limit* ⟹ *step-checked* (*proof-steps step*)›
shows ‹*step-checked step*›
proof −

obtain *witnesses* **where** *witnesses*: ‹*step-witnesses step* = *SuccessorWitnesses witnesses*›
using *assms*(*1*) *check-successors-def*
by (*metis* (*no-types*, *lifting*) *proof-step-witnesses.case-eq-if proof-step-witnesses.collapse*(*2*))

define *width* **where** ‹*width* = *nat* (*step-width step*)›
define *bound* **where** ‹*bound* = *nat* (*step-bound step*)›
define *ocmps* **where** ‹*ocmps* = *ocmp-list width*›
define *A-list* **where** ‹*A-list* = *step-vect-list step*›
define *A* **where** ‹*A* = *vt-list A-list*›
define *nrcmps* **where** ‹*nrcmps* = *filter* (λ*c*. ¬*is-redundant-cmp-vt c A*) *ocmps*›
define *Bs* **where** ‹*Bs* = *map* (λ*c*. *vt-list* (*map* (*apply-cmp-list c*) *A-list*)) *nrcmps*›

note *width-def*[*simp*] *bound-def*[*simp*] *ocmps-def*[*simp*] *A-list-def*[*simp*] *A-def*[*simp*] *nrcmps-def*[*simp*]
note *Bs-def*[*simp*]

have *nonzero-bound*: ‹*bound* $\neq$ *0*›
  using *assms*(*1*)
  unfolding *witnesses check-successors-def*
  by *auto*
then obtain *suc-bound* **where** *suc-bound*: ‹*bound* = *Suc suc-bound*›
  using *not0-implies-Suc* **by** *blast*

have *A-list-lengths*: ‹*list-all* (λ*xs*. *length xs* = *width*) *A-list*›
  using *assms*(*1*)
  unfolding *witnesses check-successors-def*
  by (*simp*; *meson*)
hence *A-lengths*: ‹*set-vt A* $\subseteq$ {*xs*. *length xs* = *width*}›
  by (*simp add*: *list.pred-set set-vt-list subsetI*)

have ‹*is-unsorted-vt width A*›
  using *assms*(*1*)
  unfolding *witnesses check-successors-def*
  by (*simp*; *meson*)
hence *is-unsorted*: ‹*pls-bound* (*list-to-vect* ‘ *set-vt A*) *1*›
  using *is-unsorted-vt-bound A-lengths* **by** *simp*

have *checked-witnesses*:
  ‹*list-all2* (λ*B w*.
    case get-bound proof-steps step-limit w width B of
      None $\Rightarrow$ *False* | *Some b* $\Rightarrow$ *Suc b* $\geq$ *bound*
) *Bs witnesses*›
  unfolding *Bs-def bound-def*
  using *assms*(*1*)
  unfolding *witnesses check-successors-def*
  by (*simp*; *meson*)

have *A-suc-lengths*:
  ‹$\bigwedge$*x*. *set-vt* (*vt-list* (*map* (*apply-cmp-list x*) *A-list*)) $\subseteq$ {*xs*. *length xs* = *width*}›
  by (*metis* (*mono-tags*, *lifting*) *A-list-lengths image-subsetI length-apply-cmp-list*
    list.pred-set list.set-map mem-Collect-eq set-vt-list)

have *B-bound*: ‹$\bigwedge$*B*. *B* $\in$ *set Bs* $\Longrightarrow$ *pls-bound* (*list-to-vect* ‘ *set-vt B*) *suc-bound*›
proof −
  fix *B* **assume** *B*: ‹*B* $\in$ *set Bs*›

  hence *B-lengths*: ‹*set-vt B* $\subseteq$ {*xs*. *length xs* = *width*}›
    using *A-suc-lengths* **by** *auto*

  have ‹$\exists$ *w*. *case get-bound proof-steps step-limit w width B of*
      None $\Rightarrow$ *False* | *Some b* $\Rightarrow$ *Suc b* $\geq$ *Suc suc-bound*›
    using *B checked-witnesses*
    by (*fold suc-bound*; *meson list-all2-witness*)

hence ⟨∃ *w b*. *get-bound proof-steps step-limit w width B* = *Some b* ∧ *b* ≥ *suc-bound*⟩
by (*metis* (*no-types*, *lifting*) *Suc-le-mono option.case-eq-if option.collapse*)

hence ⟨∃ *b*. *pls-bound* (*list-to-vect* ' *set-vt B*) *b* ∧ *b* ≥ *suc-bound*⟩
by (*metis B-lengths assms*(*2*) *get-bound-bound*)

thus ⟨*pls-bound* (*list-to-vect* ' *set-vt B*) *suc-bound*⟩
using *pls-bound-def* **by** *auto*
qed

have *nrcmp-bounds*:
⟨⋀*c*. *c* ∈ *set nrcmps* ⟹
pls-bound (*apply-cmp c* ' *list-to-vect* ' *set-vt A*) *suc-bound*⟩
proof −
fix *c* **assume** *c*: ⟨*c* ∈ *set nrcmps*⟩
hence ⟨*fst c* < *snd c* ∧ *snd c* < *width*⟩
by (*simp add*: *set-ocmp-list*)
hence *c-range*: ⟨*fst c* < *width* ∧ *snd c* < *width*⟩
by *auto*

have ⟨*pls-bound* (*list-to-vect* ' *apply-cmp-list c* ' *set-vt A*) *suc-bound*⟩
using *c B-bound set-vt-list* **by** *fastforce*

thus ⟨*pls-bound* (*apply-cmp c* ' *list-to-vect* ' *set-vt A*) *suc-bound*⟩
by (*subst apply-cmp-as-apply-cmp-list'*[**where** *n*=*width*];
insert A-lengths c-range set-list-vt set-vt-list; *auto*)
qed

have *rcmp-bounds*:
⟨⋀*c*. *c* ∈ *set ocmps* − *set nrcmps* ⟹
pls-bound (*apply-cmp c* ' *list-to-vect* ' *set-vt A*) = *pls-bound* (*list-to-vect* ' *set-vt A*)⟩
proof −
fix *c* **assume** *c*: ⟨*c* ∈ *set ocmps* − *set nrcmps*⟩
hence ⟨*fst c* < *snd c* ∧ *snd c* < *width*⟩
by (*simp add*: *set-ocmp-list*)
hence *c-range*: ⟨*fst c* < *width* ∧ *snd c* < *width*⟩
by *auto*

have ⟨*is-redundant-cmp-vt c A*⟩
using *c* **by** *auto*
hence ⟨*redundant-cmp c* (*list-to-vect* ' *set-vt A*)⟩
using *redundant-cmp-from-is-redundant-cmp-vt' A-lengths c-range*
by *blast*

thus ⟨*pls-bound* (*apply-cmp c* ' *list-to-vect* ' *set-vt A*) = *pls-bound* (*list-to-vect* ' *set-vt A*)⟩
using *redundant-cmp-bound* **by** *blast*

```
  qed

  have ocmp-bounds:
    ‹⋀c. c ∈ set ocmps ⟹
      pls-bound (apply-cmp c ‘ list-to-vect ‘ set-vt A) suc-bound ∨
     pls-bound (apply-cmp c ‘ list-to-vect ‘ set-vt A) = pls-bound (list-to-vect ‘ set-vt
A)›
    using nrcmp-bounds rcmp-bounds
    by blast

  have ‹pls-bound (list-to-vect ‘ set-vt A) bound›
    unfolding suc-bound
  proof (rule ocmp-suc-bound)
    show ‹¬inj-on weight (list-to-vect ‘ set-vt A)›
      using is-unsorted bound-unsorted by blast
  next
    show ‹⋀v. v ∈ list-to-vect ‘ set-vt A ⟹ fixed-width-vect width v›
      by (metis (full-types) A-lengths Ball-Collect list-to-vect-set-vt-fixed-width)
  next
    show  ‹⋀c. fst c < snd c ∧ snd c < width ⟹
      pls-bound (apply-cmp c ‘ list-to-vect ‘ set-vt A) suc-bound ∨
     pls-bound (apply-cmp c ‘ list-to-vect ‘ set-vt A) = pls-bound (list-to-vect ‘ set-vt
A)›
      using ocmp-bounds
      by (simp add: set-ocmp-list)
  qed

  thus ?thesis
    using A-list-lengths step-checked-def by simp
qed



definition extremal-channels-vt :: ‹vect-trie ⇒ nat ⇒ bool ⇒ nat list› where
  ‹extremal-channels-vt A n pol =
    filter (λi. is-member-vt (map (λj. (j = i) ≠ pol) [0..<n]) A) [0..<n]›

lemma extremal-channels-vt-bound:
  assumes ‹i ∈ set (extremal-channels-vt A n pol)›
  shows ‹i < n›
proof -
  have ‹i ∈ set [0..<n]›
    using assms extremal-channels-vt-def by auto
  thus ?thesis
    by auto
qed

lemma extremal-channels-in-set:
  assumes ‹i ∈ set (extremal-channels-vt A n pol)›
```

shows ‹*is-member-vt* (*map* (λ*j*. (*j* = *i*) ≠ *pol*) [*0*..<*n*]) *A*›
using *assms* **unfolding** *extremal-channels-vt-def*
by *auto*

lemma *distinct-extremal-channels-vt*:
‹*distinct* (*extremal-channels-vt A n pol*)›
using *distinct-filter extremal-channels-vt-def* **by** *simp*

fun *prune-extremal-vt* :: ‹*bool* ⇒ *nat* ⇒ *vect-trie* ⇒ *vect-trie*› **where**
‹*prune-extremal-vt* - - *VtEmpty* = *VtEmpty*› |
‹*prune-extremal-vt True 0* (*VtNode* - *a-lo* -) = *a-lo*› |
‹*prune-extremal-vt False 0* (*VtNode* - - *a-hi*) = *a-hi*› |
‹*prune-extremal-vt pol* (*Suc i*) (*VtNode a a-lo a-hi*) =
VtNode a (*prune-extremal-vt pol i a-lo*) (*prune-extremal-vt pol i a-hi*)›

fun *remove-nth* :: ‹*nat* ⇒ *'a list* ⇒ *'a list*› **where**
‹*remove-nth* - [] = []› |
‹*remove-nth 0* (*x* # *xs*) = *xs*› |
‹*remove-nth* (*Suc i*) (*x* # *xs*) = *x* # *remove-nth i xs*›

lemma *remove-nth-as-take-drop*:
‹*remove-nth i xs* = *take i xs* @ *drop* (*Suc i*) *xs*›
by (*induction i xs rule*: *remove-nth.induct*; *simp*)

definition *prune-nth* :: ‹*nat* ⇒ *vect* ⇒ *vect*› **where**
‹*prune-nth n v i* = (*if i* ≥ *n then v* (*Suc i*) *else v i*)›

lemma *list-to-vect-remove-nth*:
assumes ‹*i* < *length xs*›
shows ‹*list-to-vect* (*remove-nth i xs*) = *prune-nth i* (*list-to-vect xs*)›
unfolding *prune-nth-def list-to-vect-as-nth remove-nth-as-take-drop*
using *assms*
by (*auto simp add*: *nth-append min-def*)

lemma *list-to-vect-remove-nth-image*:
assumes ‹*i* < *n*› ‹*V* ⊆ {*xs*. *length xs* = *n*}›
shows ‹*list-to-vect* ‘ *remove-nth i* ‘ *V* = *prune-nth i* ‘ *list-to-vect*‘ *V*›
by (*subst* (*1 2*) *image-image*; *rule image-cong*; *simp?*;
subst list-to-vect-remove-nth; *insert assms*; *force*)

lemma *filter-hd-Cons-image*:
‹(#) *x* ‘ *set-vt A* ∩ {*xs*. *P* (*xs* ! *0*)} = (*if P x then* (#) *x* ‘ *set-vt A else* {})›
by (*cases* ‹*P x*›; *auto simp add*: *Int-absorb2 image-subsetI*)

lemma *Cons-image-length*:
‹(#) *x* ‘ *set-vt A* ⊆ {*xs*. *length xs* = *Suc n*} = (*set-vt A* ⊆ {*xs*. *length xs* = *n*})›
using *image-subset-iff* **by** *auto*

lemma *distrib-collect-bounded*: ‹{*x* ∈ *A* ∪ *B*. *P x*} = (*A* ∩ {*x*. *P x*}) ∪ (*B* ∩ {*x*.

$P\ x\})$›
  by *blast*

lemma *image-Cons-shift-filter-ith*:
  ‹$(\#)\ x$ ‘ *set-vt a-lo* $\cap \{xs.\ P\ (xs\ !\ Suc\ i)\} = (\#)\ x$ ‘ (*set-vt a-lo* $\cap \{xs.\ P\ (xs\ !\ i)\})$›
  by *auto*

lemma *set-vt-prune-extremal-vt*:
  assumes ‹*set-vt* $A \subseteq \{xs.\ length\ xs = n\}$› ‹$i < n$›
  shows ‹*set-vt* (*prune-extremal-vt pol i A*) $= (\lambda xs.$ *remove-nth i xs*) ‘ $\{xs \in$ *set-vt* $A.\ xs!i \neq pol\}$›
  using *assms*
proof (*induction pol i A arbitrary*: *n rule*: *prune-extremal-vt.induct*)
  case (*1 uu uv*)
  then show *?case*
    by *simp*
next
  case (*2 uw a-lo ux*)
  then show *?case*
    by (*cases uw*; *simp add*: *remove-nth-as-take-drop drop-Suc Collect-conj-eq Collect-disj-eq*
      Int-Un-distrib2 image-Un filter-hd-Cons-image image-image)
next
  case (*3 uy uz a-hi*)
  then show *?case*
    by (*cases uy*; *simp add*: *remove-nth-as-take-drop drop-Suc Collect-conj-eq Collect-disj-eq*
      Int-Un-distrib2 image-Un filter-hd-Cons-image[**where** P=‹$\lambda x.\ x$›] *image-image*)
next
  case (*4 pol i a a-lo a-hi*)
  then obtain n' **where** n': ‹$n = Suc\ n'$›
    using *not0-implies-Suc* **by** *fastforce*
  have *a*: ‹$a = False$›
    using *4* **by** *auto*

  have *a-lo-length*: ‹*set-vt a-lo* $\subseteq \{xs.\ length\ xs = n'\}$›
    using *4* n' **unfolding** *a*
    by (*simp add*: *Cons-image-length*)
  have *a-hi-length*: ‹*set-vt a-hi* $\subseteq \{xs.\ length\ xs = n'\}$›
    using *4* n' **unfolding** *a*
    by (*simp add*: *Cons-image-length*)

  have *i*: ‹$i < n'$›
    using *4* n' **by** *simp*

  have *a-lo-IH*:
    ‹*set-vt* (*prune-extremal-vt pol i a-lo*) =
      $(\lambda xs.$ *remove-nth i xs*) ‘ $\{xs \in$ *set-vt a-lo*. $xs!i \neq pol\}$›
    using *4 a-lo-length i* **by** *blast*

have *a-hi-IH*:
  ‹*set-vt* (*prune-extremal-vt pol i a-hi*) =
    (λ*xs. remove-nth i xs*) ‘ {*xs* ∈ *set-vt a-hi. xs*!*i* ≠ *pol*}›
  using *4 a-hi-length i* **by** *blast*

have
  ‹*set-vt* (*prune-extremal-vt pol* (*Suc i*) (*VtNode False a-lo a-hi*)) =
    ((#) *False* ‘ *set-vt* (*prune-extremal-vt pol i a-lo*)) ∪
    ((#) *True* ‘ *set-vt* (*prune-extremal-vt pol i a-hi*))
  ›
  by *simp*

then show *?case*
  using *4 n′ a-lo-length a-hi-length i*
  unfolding *a prune-extremal-vt.simps set-vt.simps a-lo-IH a-hi-IH image-image*
    remove-nth.simps[*symmetric*] *distrib-collect-bounded image-Un*
    image-Cons-shift-filter-ith[**where** *P*=‹λ*x. x* ≠ *pol*›]
  by *blast*
qed

lemma *prune-extremal-vt-lengths*:
  assumes ‹*set-vt A* ⊆ {*xs. length xs* = *Suc n*}› ‹*i* < *Suc n*›
  shows ‹*set-vt* (*prune-extremal-vt pol i A*) ⊆ {*xs. length xs* = *n*}›
  using *assms*
proof (*induction pol i A arbitrary*: *n rule*: *prune-extremal-vt.induct*)
  case (*1 uu uv*)
  then show *?case* **by** *simp*
next
  case (*2 uw a-lo ux*)
  then show *?case*
    by (*cases uw*; *auto*)
next
  case (*3 uy uz a-hi*)
  then show *?case*
    by (*cases uy*; *auto*)
next
  case (*4 pol i a a-lo a-hi*)
  then obtain *n′* **where** *n′*: ‹*n* = *Suc n′*›
    using *not0-implies-Suc* **by** *fastforce*
  then show *?case*
    using *4*
    by (*cases pol*; *cases a*; *simp add*: *n′ Cons-image-length*)
qed

definition *check-huffman* :: ‹(*int* ⇒ *proof-step*) ⇒ *int* ⇒ *proof-step* ⇒ *bool*› **where**
  ‹*check-huffman proof-steps step-limit step* = (*case step-witnesses step of*
    SuccessorWitnesses - ⇒ *False* |

HuffmanWitnesses pol witnesses ⇒
let
width = (*nat* (*step-width step*));
width′ = *nat.pred width*;
bound = (*nat* (*step-bound step*));
A-list = *step-vect-list step*;
A = *vt-list A-list*;
extremal-channels = *extremal-channels-vt A width pol*;
Bs = *map* (λ*c*. *prune-extremal-vt pol c A*) *extremal-channels*;
bounds = *map2* (λ*B w*. *get-bound proof-steps step-limit w width′ B*) *Bs witnesses*;
huffman-bound = *sucmax-value-bound-huffman* (*mset* (*map the bounds*))
in
width ≠ *0* ∧
list-all (λ*xs*. *length xs* = *width*) *A-list* ∧
witnesses ≠ [] ∧
length extremal-channels = *length witnesses* ∧
list-all (λ*b*. *b* ≠ *None*) *bounds* ∧
huffman-bound ≥ *bound*
)›

lemma *mset-ran-map-of-zip*:
assumes ‹*distinct xs*› ‹*length xs* = *length ys*›
shows ‹*mset-ran* (*map-of* (*zip xs ys*)) = *mset ys*›
using *assms*
by (*induction ys arbitrary*: *xs*; *unfold mset-ran-def*; (*simp*; *fail*)?;
metis dom-map-of-zip image-mset-map-of map-fst-zip map-snd-zip mset-set-set)

lemma *list-to-vect-prune-push*:
assumes ‹*set-vt A* ⊆ {*xs*. *length xs* = *n*}› ‹*i* < *n*›
shows
‹*list-to-vect* ‘ {*xs* ∈ *set-vt A*. *xs* ! *i* ≠ *pol*} = {*v* ∈ *list-to-vect* ‘ *set-vt A*. (*v i*) ≠ *pol*}›
using *assms*(*1*) *assms*(*2*) *list-to-vect-as-nth* **by** *auto*

lemma *extremal-list-to-vect*:
assumes ‹*i* < *n*›
shows ‹*list-to-vect* (*map* (λ*j*. (*j* = *i*) ≠ *pol*) [*0*..<*n*]) = *apply-pol n pol* ((≠) *i*)›
using *assms* **unfolding** *invert-vect-def list-to-vect-as-nth*
by (*cases pol*; *auto*)

definition *shift-channels* :: ‹*nat* ⇒ *nat* ⇒ *nat* ⇒ *nat*› **where**
‹*shift-channels n i j* = (*if Suc j* = *n then i else if j* ≥ *i* ∧ *Suc j* < *n then Suc j else j*)›

definition *unshift-channels* :: ‹*nat* ⇒ *nat* ⇒ *nat* ⇒ *nat*› **where**
‹*unshift-channels n i j* =
(*if j* = *i then nat.pred n else if j* > *i* ∧ *j* < *n then nat.pred j else j*)›

lemma *unshift-shift-inverse*:
assumes ‹$i < n$›
shows ‹*unshift-channels* n i (*shift-channels* n i j) = j›
proof −
have ‹$j < i \Longrightarrow$ *unshift-channels* n i (*shift-channels* n i j) = j›
unfolding *unshift-channels-def shift-channels-def* **using** *assms* **by** *simp*
moreover have ‹$j \geq n \Longrightarrow$ *unshift-channels* n i (*shift-channels* n i j) = j›
unfolding *unshift-channels-def shift-channels-def* **using** *assms* **by** *simp*
moreover have ‹$j = i \Longrightarrow$ *unshift-channels* n i (*shift-channels* n i j) = j›
unfolding *unshift-channels-def shift-channels-def* **using** *assms* **by** *auto*
moreover have ‹*Suc* $j = n \Longrightarrow$ *unshift-channels* n i (*shift-channels* n i j) = j›
unfolding *unshift-channels-def shift-channels-def* **using** *assms* **by** *auto*
moreover have ‹$j > i \wedge$ *Suc* $j < n \Longrightarrow$ *unshift-channels* n i (*shift-channels* n i j) = j›
unfolding *unshift-channels-def shift-channels-def* **using** *assms* **by** *simp*
ultimately show *?thesis* **by** *linarith*
qed

lemma *shift-unshift-inverse*:
assumes ‹$i < n$›
shows ‹*shift-channels* n i (*unshift-channels* n i j) = j›
proof −
have ‹$j < i \Longrightarrow$ *shift-channels* n i (*unshift-channels* n i j) = j›
unfolding *unshift-channels-def shift-channels-def* **using** *assms* **by** *simp*
moreover have ‹$j \geq n \Longrightarrow$ *shift-channels* n i (*unshift-channels* n i j) = j›
unfolding *unshift-channels-def shift-channels-def* **using** *assms* **by** *simp*
moreover have ‹$j = i \Longrightarrow$ *shift-channels* n i (*unshift-channels* n i j) = j›
unfolding *unshift-channels-def shift-channels-def* **using** *assms* **by** *auto*
moreover have ‹*Suc* $j = n \Longrightarrow$ *shift-channels* n i (*unshift-channels* n i j) = j›
unfolding *unshift-channels-def shift-channels-def* **using** *assms* **by** (*cases* j; *simp*)
moreover have ‹$j > i \wedge$ *Suc* $j < n \Longrightarrow$ *shift-channels* n i (*unshift-channels* n i j) = j›
unfolding *unshift-channels-def shift-channels-def* **using** *assms* **by** (*cases* j; *simp*)
ultimately show *?thesis* **by** *linarith*
qed

lemma *bij-shift-channels*:
assumes ‹$i < n$›
shows ‹*bij* (*shift-channels* n i)›
by (*metis* (*full-types*) *assms bijI′ shift-unshift-inverse unshift-shift-inverse*)

lemma *shift-channels-permutes*:
assumes ‹$i < n$›
shows ‹*shift-channels* n i *permutes* $\{..<n\}$›
using *assms bij-shift-channels* **unfolding** *permutes-def*
by (*metis Suc-lessD lessI lessThan-iff shift-channels-def shift-unshift-inverse*

unshift-shift-inverse)

lemma *prune-nth-as-perm*:
assumes ‹*fixed-width-vect n v*› ‹*v i*› ‹$i < n$›
shows ‹*prune-nth i v* = *apply-perm* (*shift-channels n i*) *v*›
unfolding *prune-nth-def shift-channels-def apply-perm-def*
using *assms fixed-width-vect-def* **by** *auto*

lemma *inverted-prune-nth*:
assumes ‹$i < Suc\ n$›
shows ‹*invert-vect n* (*prune-nth i* (*invert-vect* (*Suc n*) *v*)) = *prune-nth i v*›
unfolding *invert-vect-def prune-nth-def*
using *assms* **by** *auto*

lemma *prune-nth-as-perm-inverted*:
assumes ‹*fixed-width-vect n v*› ‹¬*v i*› ‹$i < n$›
shows
‹*prune-nth i v* = *invert-vect* (*nat.pred n*) (*apply-perm* (*shift-channels n i*) (*invert-vect n v*))›
proof −
have ‹*fixed-width-vect n* (*invert-vect n v*)›
by (*simp add*: *assms*(*1*) *invert-vect-fixed-width*)
moreover have ‹(*invert-vect n v*) *i*›
by (*simp add*: *assms*(*2*) *assms*(*3*) *invert-vect-def*)
ultimately have
‹*prune-nth i* (*invert-vect n v*) = *apply-perm* (*shift-channels n i*) (*invert-vect n v*)›
by (*simp add*: *assms*(*3*) *prune-nth-as-perm*)
moreover have ‹*invert-vect* (*nat.pred n*) (*prune-nth i* (*invert-vect n v*)) = *prune-nth i v*›
using *inverted-prune-nth*[*of i* ‹*nat.pred n*› *v*]
by (*metis Suc-n-not-le-n assms*(*3*) *nat.split-sels*(*2*) *not-less0*)
ultimately show *?thesis*
by *simp*
qed

lemma *prune-nth-as-perm-pol-image*:
assumes ‹$\bigwedge v.\ v \in V \Longrightarrow$ *fixed-width-vect n v*› ‹$\bigwedge v.\ v \in V \Longrightarrow v\ i = pol$› ‹$i < n$›
shows
‹*prune-nth i* ‘ *V* = *apply-pol* (*nat.pred n*) *pol* ‘
apply-perm (*shift-channels n i*) ‘ *apply-pol n pol* ‘ *V*›
unfolding *image-image*
by (*intro image-cong*; *simp add*: *assms prune-nth-as-perm prune-nth-as-perm-inverted*)

lemma *pruned-bound-pol-using-prune-extremal-vt*:
assumes ‹*pls-bound* (*list-to-vect* ‘ *set-vt* (*prune-extremal-vt pol i A*)) *b*›
‹$i < n$›
‹*set-vt A* ⊆ {*xs. length xs* = *n*}›

‹is-member-vt (map (λj. (j = i) ≠ pol) [0..<n]) A›
  shows ‹pruned-bound-pol n pol (list-to-vect ‘ set-vt A) i b›
proof −
  obtain n′ **where** n′: ‹n = Suc n′›
    using assms less-imp-Suc-add **by** blast

  have f1: ‹⋀v. v ∈ list-to-vect ‘ {xs ∈ set-vt A. xs ! i ≠ pol} ⟹ fixed-width-vect n v›
    by (metis (mono-tags) Collect-subset assms(3) image-mono in-mono
      list-to-vect-set-vt-fixed-width mem-Collect-eq)

  hence ‹⋀v.
    v ∈ apply-pol n (¬ pol) ‘ list-to-vect ‘ {xs ∈ set-vt A. xs ! i ≠ pol}
    ⟹ fixed-width-vect n v›
    using invert-vect-fixed-width **by** auto

  hence ‹⋀v.
    v ∈ apply-perm (shift-channels n i) ‘ apply-pol n (¬ pol) ‘
      list-to-vect ‘ {xs ∈ set-vt A. xs ! i ≠ pol}
    ⟹ fixed-width-vect n v›
    by (meson apply-perm-fixed-width-image assms(2) shift-channels-permutes)

  moreover have ‹⋀v. v ∈ set-vt A ∧ v ! i ≠ pol ⟹
    apply-perm (shift-channels n i) (apply-pol n (¬ pol) (
      list-to-vect v)) n′›
    unfolding invert-vect-def list-to-vect-as-nth apply-perm-def shift-channels-def
    using n′ assms(3) **by** force

  hence ‹⋀v.
    v ∈ apply-perm (shift-channels n i) ‘ apply-pol n (¬ pol) ‘
      list-to-vect ‘ {xs ∈ set-vt A. xs ! i ≠ pol}
    ⟹ v n′›
    unfolding image-image **by** blast

  ultimately have f2: ‹⋀v.
    v ∈ apply-perm (shift-channels n i) ‘ apply-pol n (¬ pol) ‘
      list-to-vect ‘ {xs ∈ set-vt A. xs ! i ≠ pol}
    ⟹ fixed-width-vect n′ v›
    unfolding n′ fixed-width-vect-def
    by (metis (no-types, lifting) Suc-leI le-imp-less-or-eq)

  have ‹set-vt (prune-extremal-vt pol i A) = remove-nth i ‘ {xs ∈ set-vt A. xs ! i ≠ pol}›
    using assms
    by (subst set-vt-prune-extremal-vt[**where** n=n]; simp)
  hence ‹pls-bound (list-to-vect ‘ remove-nth i ‘ {xs ∈ set-vt A. xs ! i ≠ pol}) b›
    using assms **by** simp
  hence ‹pls-bound (prune-nth i ‘ list-to-vect ‘ {xs ∈ set-vt A. xs ! i ≠ pol}) b›

```
  by (subst (asm) list-to-vect-remove-nth-image[where n=n]; insert Ball-Collect
assms; auto)
 hence
  ‹pls-bound (apply-pol n' (¬pol) `
    apply-perm (shift-channels n i) `
    apply-pol n (¬pol) `
    list-to-vect ` {xs ∈ set-vt A. xs ! i ≠ pol}) b›
  by (subst (asm) prune-nth-as-perm-pol-image[where n=n and pol=‹¬pol›];
     insert assms(2, 3) fixed-width-vect-list-to-vect list-to-vect-as-nth; auto simp
add: n')
 hence
  ‹pls-bound (
    apply-perm (shift-channels n i) `
    apply-pol n (¬pol) `
    list-to-vect ` {xs ∈ set-vt A. xs ! i ≠ pol}) b›
  by (subst (asm) apply-pol-bound-iff; simp add: f2)
 hence
  ‹pls-bound (
    apply-pol n (¬pol) `
    list-to-vect ` {xs ∈ set-vt A. xs ! i ≠ pol}) b›
  by (subst (asm) permuted-bounds-iff[symmetric]; simp add: assms(2) bij-shift-channels)
 hence
  ‹pls-bound (list-to-vect ` {xs ∈ set-vt A. xs ! i ≠ pol}) b›
  by (subst (asm) apply-pol-bound-iff; simp add: f1)
 hence *: ‹pls-bound {v ∈ list-to-vect ` set-vt A. (v i) ≠ pol} b›
  using list-to-vect-prune-push assms by simp

 have ‹apply-pol n pol ((≠) i) ∈ list-to-vect ` set-vt A›
  using assms is-member-vt-as-member-set-vt
  by (subst extremal-list-to-vect[symmetric]; simp)
 hence extremal: ‹(≠) i ∈ apply-pol n pol ` list-to-vect ` set-vt A›
  by (subst apply-pol-invol[where n=n and pol=pol and v=‹(≠) i›, symmetric];
blast)


 have ‹pls-bound {v ∈ apply-pol n pol ` list-to-vect ` set-vt A. ¬ v i} b›
 proof (cases pol)
  case True
  then show ?thesis
   using * by auto
 next
  case False
  have ‹pls-bound {v ∈ list-to-vect ` set-vt A. v i} b›
   using * False by auto
  hence ‹pls-bound (invert-vect n ` {v ∈ list-to-vect ` set-vt A. v i}) b›
   by (intro inverted-bound; (simp; fail)?;
      metis (mono-tags) Ball-Collect Collect-subset assms(3) in-mono
      list-to-vect-set-vt-fixed-width)
```

moreover have
‹*invert-vect n* ‘ {v ∈ *list-to-vect* ‘ *set-vt A*. v i} =
{v ∈ *invert-vect n* ‘ *list-to-vect* ‘ *set-vt A*. ¬ v i}›
by (*intro set-eqI iffI*; *unfold invert-vect-def*; *simp*; *insert assms*(*2*); *blast*)

finally show *?thesis*
by (*cases pol*; *simp add*: *False*)
qed

hence ‹*pruned-bound* (*apply-pol n pol* ‘ *list-to-vect* ‘ *set-vt A*) *i b*›
unfolding *pruned-bound-def*
using *extremal* **by** *simp*

thus *?thesis*
using *pruned-bound-pol-def*
by *simp*
qed

lemma *check-huffman-step-checked*:
assumes ‹*check-huffman proof-steps step-limit step*›
‹⋀*step*. *0* ≤ *step* ∧ *step* < *step-limit* ⟹ *step-checked* (*proof-steps step*)›
shows ‹*step-checked step*›
proof −

obtain *witnesses pol* **where** *witnesses-pol*: ‹*step-witnesses step* = *HuffmanWitnesses pol witnesses*›
using *assms*(*1*) *check-huffman-def*
by (*metis* (*no-types*, *lifting*) *proof-step-witnesses.case-eq-if proof-step-witnesses.collapse*(*1*))

define *width* **where** ‹*width* = *nat* (*step-width step*)›
define *width′* **where** ‹*width′* = *nat.pred width*›
define *bound* **where** ‹*bound* = *nat* (*step-bound step*)›
define *A-list* **where** ‹*A-list* = *step-vect-list step*›
define *A* **where** ‹*A* = *vt-list A-list*›
define *extremal-channels* **where** ‹*extremal-channels* = *extremal-channels-vt A width pol*›
define *Bs* **where** ‹*Bs* = *map* (λ*c*. *prune-extremal-vt pol c A*) *extremal-channels*›
define *bounds* **where**
‹*bounds* = *map2* (λ*B w*. *get-bound proof-steps step-limit w width′ B*) *Bs witnesses*›
define *huffman-bound* **where** ‹*huffman-bound* = *sucmax-value-bound-huffman* (*mset* (*map the bounds*))›

define *D* **where** ‹*D* = *map-of* (*zip extremal-channels* (*map the bounds*))›

note *width-def*[*simp*] *width′-def*[*simp*] *bound-def*[*simp*] *A-list-def*[*simp*] *A-def*[*simp*]

note *extremal-channels-def*[*simp*] *Bs-def*[*simp*] *bounds-def*[*simp*] *huffman-bound-def*[*simp*] *D-def*[*simp*]

have *checked*: ‹
width ≠ *0* ∧
list-all (λ*xs*. *length xs* = *width*) *A-list* ∧
witnesses ≠ [] ∧
length extremal-channels = *length witnesses* ∧
list-all (λ*b*. *b* ≠ *None*) *bounds* ∧
huffman-bound ≥ *bound*›
using *assms*(*1*)
unfolding *witnesses-pol check-huffman-def*
unfolding *width-def bound-def A-list-def A-def extremal-channels-def Bs-def bounds-def*
huffman-bound-def
unfolding *Let-def*
by *auto*

have *A-list-lengths*: ‹*list-all* (λ*xs*. *length xs* = *width*) *A-list*›
using *checked* **by** *simp*
hence *A-lengths*: ‹*set-vt A* ⊆ {*xs*. *length xs* = *width*}›
by (*simp add*: *list.pred-set set-vt-list subsetI*)

have *witnesses-length*: ‹*length extremal-channels* = *length witnesses*›
using *checked* **by** *simp*

have *nonempty-witnesses*: ‹*witnesses* ≠ []›
using *checked* **by** *simp*

have *huffman-bound-bound*: ‹*huffman-bound* ≥ *bound*›
using *checked* **by** *auto*

have *Bs-length*: ‹*length Bs* = *length witnesses*›
by (*simp add*: *witnesses-length del*: *extremal-channels-def*)
hence *bounds-length*: ‹*length bounds* = *length witnesses*›
by *simp*
hence ‹*length* (*zip extremal-channels* (*map the bounds*)) = *length witnesses*›
by *auto*
hence *card-dom-D*: ‹*card* (*dom D*) = *length witnesses*›
unfolding *D-def*
by (*subst dom-map-of-zip*; (*insert witnesses-length*; *auto*)?;
subst distinct-card; *insert distinct-extremal-channels-vt witnesses-length*; *simp*)

have *width'*: ‹*width* = *Suc width'*›
by (*metis Suc-n-not-le-n checked nat.split-sels*(*2*) *width'-def*)

have ‹*pls-bound* (*list-to-vect* ‘ *set-vt A*) (*sucmax.value-bound-mset* (*mset-ran D*))›
proof (*rule pls-bound-from-pruned-bounds-pol*[**where** *n*=*width* **and** *pol*=*pol* **and** *B*=*D*])
fix *v* **assume** *v*: ‹*v* ∈ *list-to-vect* ‘ *set-vt A*›

thus ‹*fixed-width-vect width v*›
by (*metis A-lengths Ball-Collect v list-to-vect-set-vt-fixed-width*)
next
show ‹*finite* (*dom D*)›
unfolding *D-def*
using *finite-dom-map-of* **by** *blast*
next
show ‹*dom D* $\neq$ {}›
using *card-dom-D nonempty-witnesses* **by** *auto*
next
have ‹$\forall$ *c* $\in$ *dom D*. *pruned-bound-pol width pol* (*list-to-vect* ‘ *set-vt A*) *c* (*the* (*D c*))›
proof
fix *c* **assume** ‹*c* $\in$ *dom D*›
hence *c-set*: ‹*c* $\in$ *set extremal-channels*›
using *witnesses-length* **by** *auto*
then obtain *i* **where** *c-def*: ‹*c* = *extremal-channels*!*i*›
and *i-bound*: ‹*i* < *length extremal-channels*›
by (*metis in-set-conv-nth*)

have *c-bound*: ‹*c* < *width*›
using *extremal-channels-vt-bound c-set extremal-channels-def* **by** *blast*

define *w* **where** ‹*w* = *witnesses*!*i*›
define *B* **where** ‹*B* = *prune-extremal-vt pol c A*›

have *B-alt*: ‹*B* = *Bs*!*i*›
unfolding *Bs-def B-def c-def*
using *witnesses-length i-bound nth-map*[*of i extremal-channels*]
by *simp*

have *length-zip*: ‹*length* (*zip Bs witnesses*) = *length witnesses*›
using *Bs-length* **by** *auto*

have
‹*bounds*!*i* =
(λ(*x*, *y*). *get-bound proof-steps step-limit y width′ x*) (*zip Bs witnesses* ! *i*)›
unfolding *bounds-def*
using *nth-map*[*of i* ‹*zip Bs witnesses*›] *length-zip witnesses-length i-bound*
by *simp*
also have ‹. . . = *get-bound proof-steps step-limit w width′ B*›
unfolding *B-alt*
using *nth-zip*[*of i Bs witnesses*]
using *i-bound w-def witnesses-length* **by** *auto*
ultimately have *get-bound-i*: ‹*bounds*!*i* = *get-bound proof-steps step-limit w width′ B*›
by *simp*

have ‹*bounds*!*i* $\neq$ *None*›

```
      by (metis (full-types) bounds-length checked i-bound list-all-length)
    then obtain b where b-def: ‹bounds!i = Some b›
      by blast
    hence b-alt: ‹b = the (get-bound proof-steps step-limit w width' B)›
      using get-bound-i
      by (metis option.sel)

    have ‹D c = Some b›
      unfolding c-def D-def b-def[symmetric]
      by (subst map-of-zip-nth;
       insert b-def i-bound witnesses-length; auto simp add: distinct-extremal-channels-vt)
    hence the-D-c: ‹the (D c) = b›
      using b-alt by simp

    have B-lengths: ‹set-vt B ⊆ {xs. length xs = width'}›
      unfolding B-def using A-lengths width' c-bound
      by (intro prune-extremal-vt-lengths; auto)

    have ‹pls-bound (list-to-vect ` set-vt B) b›
      using b-def assms(2) B-lengths
      unfolding get-bound-i
      by (subst get-bound-bound[
            where step-limit=step-limit and proof-steps=proof-steps
             and witness=w and n=width']; simp)

    hence ‹pruned-bound-pol width pol (list-to-vect ` set-vt A) c b›
    using B-def c-bound A-lengths extremal-channels-in-set c-set extremal-channels-def
    by (intro pruned-bound-pol-using-prune-extremal-vt[where i=c and n=width];
auto)

    thus ‹pruned-bound-pol width pol (list-to-vect ` set-vt A) c (the (D c))›
      using the-D-c by simp
  qed
  thus ‹pruned-bounds-pol width pol (list-to-vect ` set-vt A) D›
    unfolding pruned-bounds-pol-def by auto
qed

moreover have ‹mset-ran D = mset (map the bounds)›
  by (simp del: bounds-def; subst mset-ran-map-of-zip;
      insert distinct-extremal-channels-vt witnesses-length bounds-length;
      auto simp del: bounds-def)
hence ‹bound ≤ sucmax-value-bound-huffman (mset-ran D)›
  using huffman-bound-bound
  by simp
hence ‹bound ≤ sucmax.value-bound-mset (mset-ran D)›
  by (simp add: sucmax.value-bound-huffman-mset)
ultimately have ‹pls-bound (list-to-vect ` set-vt A) bound›
  by (meson dual-order.trans pls-bound-def)
```

thus *?thesis*
using *A-list-lengths step-checked-def* **by** *simp*
qed

definition *check-step* :: ‹(*int* ⇒ *proof-step*) ⇒ *int* ⇒ *proof-step* ⇒ *bool*› **where**
‹*check-step proof-steps step-limit step* = (*case step-witnesses step of*
SuccessorWitnesses - ⇒ *check-successors proof-steps step-limit step* |
HuffmanWitnesses - - ⇒ *check-huffman proof-steps step-limit step*)›

lemma *check-step-step-checked*:
assumes ‹*check-step proof-steps step-limit step*›
‹$\bigwedge$*step*. *0* ≤ *step* ∧ *step* < *step-limit* ⟹ *step-checked* (*proof-steps step*)›
shows ‹*step-checked step*›
proof (*cases* ‹*step-witnesses step*›)
case (*HuffmanWitnesses* - -)
hence ‹*check-huffman proof-steps step-limit step*›
using *assms*(*1*) *check-step-def* **by** *auto*
then show *?thesis*
using *assms*(*2*) *check-huffman-step-checked* **by** *blast*
next
case (*SuccessorWitnesses x2*)
hence ‹*check-successors proof-steps step-limit step*›
using *assms*(*1*) *check-step-def* **by** *auto*
then show *?thesis*
using *assms*(*2*) *check-successors-step-checked* **by** *blast*
qed

lemma *check-induct*:
assumes ‹*list-all* (λ*i*. *check-step proof-steps* (*int i*) (*proof-steps* (*int i*))) [*0*..<*n*]›
shows ‹$\bigwedge$*step*. *0* ≤ *step* ∧ *step* < *int n* ⟹ *step-checked* (*proof-steps step*)›
using *assms*
proof (*induction n*)
case *0*
then show *?case*
by *linarith*
next
case (*Suc n*)
have ‹*list-all* (λ*i*. *check-step proof-steps* (*int i*) (*proof-steps* (*int i*))) [*0*..<*n*]›
using *Suc.prems*(*2*) **by** *auto*
hence ∗: ‹$\bigwedge$*step*. *0* ≤ *step* ∧ *step* < *int n* ⟹ *step-checked* (*proof-steps step*)›
using *Suc.IH* **by** *blast*

have ‹*check-step proof-steps* (*int n*) (*proof-steps* (*int n*))›
using *Suc* **by** *simp*
hence ‹*step-checked* (*proof-steps* (*int n*))›

```
  by (rule check-step-step-checked[where step-limit=‹int n› and proof-steps=proof-steps];
      simp add: ∗)
 thus ‹⋀step. 0 ≤ step ∧ step < int (Suc n) ⟹ step-checked (proof-steps step)›
   using ∗ nat-less-iff not-less-less-Suc-eq by fastforce
qed

definition par :: ‹'a ⇒ 'b ⇒ 'b› where
 ‹par a b = b›

fun par-range-all :: ‹(nat ⇒ bool) ⇒ nat ⇒ nat ⇒ bool› where
 ‹par-range-all f lo n = (
   if n < 1000 then list-all f [lo..<lo + n] else
     let n' = n div 2;
         a = par-range-all f lo n';
         b = par-range-all f (lo + n') (n − n')
     in (par b a) ∧ b)›

declare par-range-all.simps[simp del]

lemma par-range-all-iff-list-all:
 ‹par-range-all f lo n = list-all f [lo..<lo + n]›
proof (induction f lo n rule: par-range-all.induct)
 case (1 f lo n)
 define n' where ‹n' = n div 2›
 hence n'-range: ‹n' ≤ n›
   by simp

 show ?case
 proof (cases ‹n < 1000›)
   case True
   then show ?thesis
     using 1 by (simp add: par-range-all.simps)
 next
   case False

   have ‹par-range-all f lo n' = list-all f [lo..<lo + n']›
     using 1 False n'-def
     by simp

   moreover have ‹par-range-all f (lo + n') (n − n') = list-all f [lo + n'..<lo
+ n' + (n − n')]›
     using 1 False n'-def
     by simp

   ultimately have ‹par-range-all f lo n = list-all f [lo..<lo + n' + (n − n')]›
     using False
     by (subst par-range-all.simps; simp;
         metis le-add1 list-all-append n'-def par-def upt-add-eq-append)
```

then show *?thesis*
using *n′-range* **by** *simp*
qed
qed

lemma *par-range-all-iff-list-all′*:
⟨*par-range-all f 0 n* = *list-all f* [*0*..<*n*]⟩
using *par-range-all-iff-list-all* **by** *simp*

definition *check-proof* :: ⟨*proof-cert* ⇒ *bool*⟩ **where**
⟨*check-proof cert* = (
let
steps = *cert-step cert*;
n = *cert-length cert*
in n ≥ *0* ∧ *par-range-all* (λ*i*. *check-step steps* (*int i*) (*steps* (*int i*))) *0* (*nat n*))⟩

lemma *check-proof-spec*:
assumes ⟨*check-proof cert*⟩
shows ⟨⋀*step*. *0* ≤ *step* ∧ *step* < *cert-length cert* ⟹ *step-checked* (*cert-step cert step*)⟩
using *assms* **unfolding** *check-proof-def par-range-all-iff-list-all′*
by (*cases* ⟨*cert-length cert* ≥ *0*⟩; *linarith?*; *metis check-induct int-nat-eq*)

definition *check-proof-get-bound* :: ⟨*proof-cert* ⇒ (*int* × *int*) *option*⟩ **where**
⟨*check-proof-get-bound cert* = (
if check-proof cert ∧ *cert-length cert* > *0*
then
let last-step = *cert-step cert* (*cert-length cert* − *1*)
in Some (*step-width last-step*, *step-bound last-step*)
else None
)⟩

lemma *step-checked-bound*:
assumes ⟨*step-checked step*⟩
shows ⟨*lower-size-bound* {*v*. *fixed-width-vect* (*nat* (*step-width step*)) *v*} (*nat* (*step-bound step*))⟩
proof −
define *A* **where** ⟨*A* = *list-to-vect* ‘ *set-vt* (*vt-list* (*step-vect-list step*))⟩
hence ⟨*pls-bound A* (*nat* (*step-bound step*))⟩
using *assms step-checked-def* **by** *auto*
moreover have ⟨*A* ⊆ {*v*. *fixed-width-vect* (*nat* (*step-width step*)) *v*}⟩
by (*metis* (*no-types*, *lifting*) *A-def Ball-Collect assms list.pred-set list-to-vect-set-vt-fixed-width set-vt-list step-checked-def*)
ultimately have
⟨*pls-bound* {*v*. *fixed-width-vect* (*nat* (*step-width step*)) *v*} (*nat* (*step-bound step*))⟩
using *bound-mono-subset* **by** *blast*
thus *?thesis*

    by (*metis* (*full-types*) *mem-Collect-eq pls-bound-implies-lower-size-bound*)
qed

lemma *check-proof-get-bound-spec*:
  assumes ‹*check-proof-get-bound cert* = *Some* (*width*, *bound*)›
  shows ‹*lower-size-bound-for-width* (*nat width*) (*nat bound*)›
proof −
  have *checked*: ‹*check-proof cert* ∧ *cert-length cert* > *0*›
    using *assms* **unfolding** *check-proof-get-bound-def*
    by (*meson option.distinct*(*1*))

  define *last-step* **where** ‹*last-step* = *cert-step cert* (*cert-length cert* − *1*)›
  hence ‹*step-checked last-step*›
    using *checked check-proof-spec*
    by *simp*

  thus *?thesis*
    by (*metis Pair-inject assms check-proof-get-bound-def checked last-step-def*
        option.inject step-checked-bound lower-size-bound-for-width-def)
qed

end
theory *Checker-Codegen*
  imports *Main Sorting-Network Checker HOL−Library.Code-Target-Numeral*
begin

lemma *check-proof-get-bound-spec*:
  assumes ‹*check-proof-get-bound cert* = *Some* (*width*, *bound*)›
  shows ‹*lower-size-bound-for-width* (*nat width*) (*nat bound*)›
  using *assms* **by** (*rule Checker.check-proof-get-bound-spec*)

definition *nat-pred-code* :: ‹*nat* ⇒ *nat*› **where**
  ‹*nat-pred-code n* = (*case n of 0* ⇒ *nat.pred 0* | *Suc n′* ⇒ *n′*)›

lemma *nat-pred-code*[*code*]: ‹*nat.pred* = *nat-pred-code*›
  by (*rule*; *metis nat-pred-code-def old.nat.simps*(*4*) *pred-def*)

export-code
    check-proof-get-bound integer-of-int int-of-integer
    ProofCert ProofStep HuffmanWitnesses SuccessorWitnesses ProofWitness
  in *Haskell* **module-name** *Verified.Checker* **file-prefix** *checker*

end